NTNU

Norwegian University of
Science and Technology

Automatic Detection and Correction of Flaws in Service Specifications

Vidar Slåtten

Master of Science in Communication Technology
Submission date: June 2008
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Frank Alexander Kraemer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

The SPACE approach uses UML collaborations and activities to express service specifications as compositions from reusable building blocks. In previous work, a transformation from these specifications to TLA was implemented, so that the specifications may be analyzed using the model checker TLC.

While this transformation made it easier for developers to use formal techniques, knowledge from this domain was still necessary when interpreting the feedback from TLC in the form of error traces. In this work, the present approach should be taken one step further towards a fully automated approach, in which the system not only derives the theorems automatically from the UML models, but also interprets any error traces to find design flaws and in some cases even suggests improvements of the design.

Assignment given: 29. October 2007
Supervisor: Peter Herrmann, ITEM

Automatic Detection and Correction of Flaws in Service Specifications

Vidar Slåtten

June 2008

Abstract

While rigorous, mathematical techniques are helpful for improving the quality of software engineering, the threshold of learning and adapting formal methods keep many practitioners from embracing these kinds of approaches. We present the *Arctis Analyzer*, a tool for supporting a developer by formal methods, without the developer having to understand any formal language. To realize this tool, we developed an extensible analysis framework that is used by the analyzer to assist the user when problems are encountered. We combine model checking with syntactic analysis so as to provide a developer with not only symptoms, but also diagnoses and fixes for the underlying flaws in the specifications.

Our work is based in SPACE, an approach for specifying reactive and distributed systems. In this approach, systems are composed of service specifications as opposed to traditional component specifications. Services are expressed by UML collaborations and activities representing their structure and behavior, respectively. This work focuses on analyzing the behavior of the services, hence on the UML activities.

The analyzer transforms the UML models into TLA$^+$, the language of the Temporal Logic of Actions, TLA. It also generates TLA theorems for a number of properties that should hold. In order to detect property violations, the tool uses the model checker TLC to check the entire state space of the formal specifications. The analyzer can visualize any error traces from TLC in terms of the graphical model that the user is working on.

This thesis presents the analyzer and how it is implemented. It also presents the analysis framework detailing twelve theorems, eleven symptoms, seven diagnoses and nine fixes. Out of these, ten theorems, six symptoms, three diagnoses and two fixes, have been implemented in the analyzer as proof of concept. The thesis also contains a number of examples showing the use of the analyzer within Arctis.

Preface

This master's thesis is submitted for my Master of Science degree in Communication Technology at the Norwegian University of Science and Technology (NTNU). The work has been carried out at the Department of Telematics during winter of 2007 and spring of 2008. My subject teacher has been Professor Peter Herrmann and my supervisor has been Ph.D. candidate Frank Alexander Kraemer.

I would like to thank Peter Herrmann and Frank Alexander Kraemer for all the guidance and feedback I have been given while working on the thesis.

München, June 17^{th} 2008

Vidar Slåtten

Contents

Chapter 1

Introduction

In the SPACE approach [KH06, KBH07, KSH07], reactive systems are specified by UML collaborations and activities.[1] Systems are hereby decomposed alongside a horizontal axis [Mik99], so that services crossing the boundaries of their participating components can be represented explicitly. A (sub-)service is encapsulated in a building block that has a behavioral interface description to allow for easy reuse inside other specifications.

Figure 1.1 shows a graphic representation of the approach. Services are composed of building blocks to form the system. This is transformed to a component-oriented specification so as to be able to generate the executable code for each deployable component.

The Arctis tool suite, shown in Fig. 1.2, gives support for creating and storing the service specifications. It also gives access to existing building blocks through a library and provides automated checks (so-called *inspectors*) that make sure the specifications are syntactically correct.

While syntactic inspection can detect and supply fixes for many types of errors, there are other properties that can only be verified by exploring the behavior of the specifications. For example, we cannot check that transmission queues will be bounded just by looking at the syntax of the activities.

We want to use formal methods to verify behavioral properties of our specifications. Unfortunately, all such methods require the user to be experienced in their use for them to give a valuable contribution to the development process. This threshold is seen as too big by many, and they avoid formal methods all together. Rushby argues in [Rus00] that the best way to increase their use in

[1] All UML elements in this work are from the UML 2 specification.

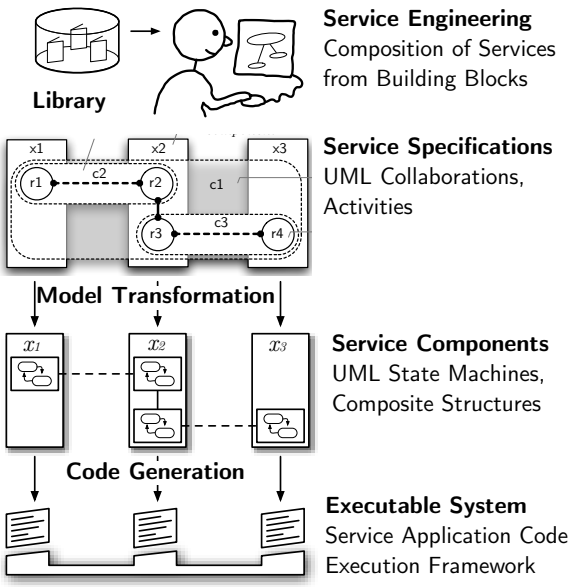


Figure 1.1: The SPACE approach. Figure from [KBH07]

software engineering, is to hide them inside development tools that automatically apply them and use them to support the user in the development process.

To accomplish what Rushby suggests, we must overcome several obstacles. We have to automatically generate a formal specification from the specification that the user is developing, the UML collaborations and activities. In addition we need to automatically generate a formal expression for the properties that should hold for the specification. Then we have to formally analyze the specification without requiring any manual intervention. Finally, the results of the analysis must be automatically interpreted so that meaningful feedback can be given to the user.

To summarize the task at hand: We aim to create a tool that brings users the benefits of formal methods, when developing reactive systems using the Arctis tool, without requiring any knowledge in the field of formal methods from them. In addition to the benefit of automatic detection of problems, we also aim to demonstrate that automatic correction is possible.

For this task, we first need a suitable formalism and to transform the model, that the user is working on, into a corresponding language. The semantics of our service-oriented models is already expressed in the form of cTLA [HK00, KH07a], which is based on the linear-time temporal logic TLA [Lam94]. TLA$^+$ is the language of TLA and also the input language for the model checker TLC [YML99]. Moreover, a tool for transforming our UML models to TLA$^+$, the *formulator*, was

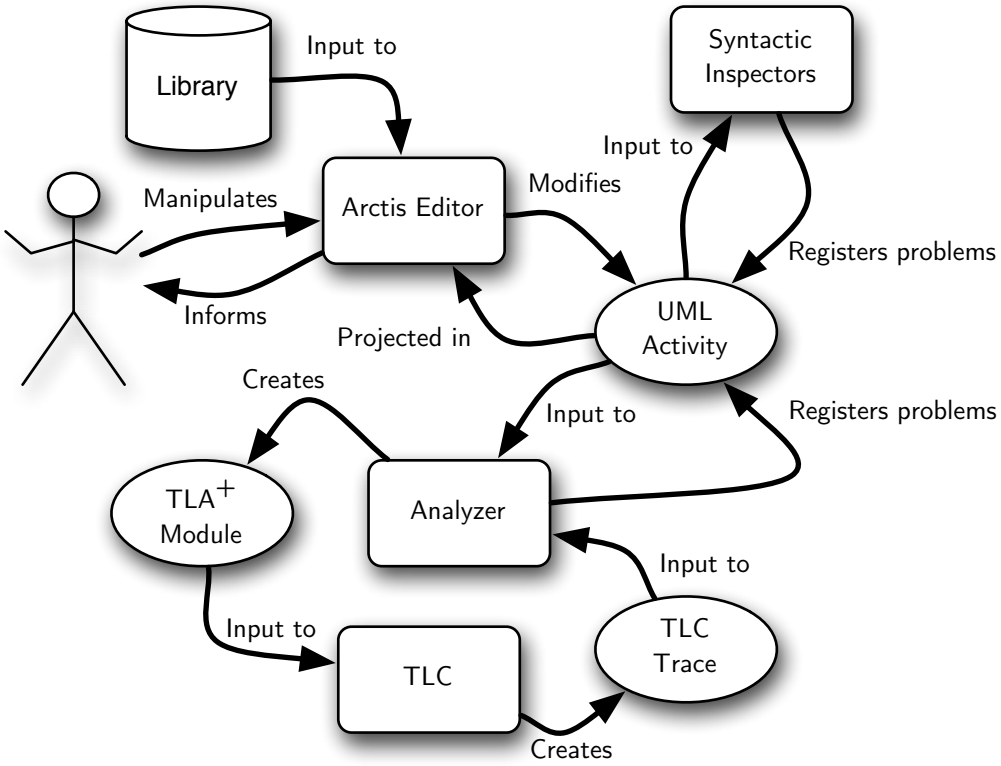


Figure 1.2: The Analyzer in the context of the Arctis tool suite

created in [Slå07]. While we make some alterations in this work, the main features can be reused.

Once we have transformed our models into a formal language, the next obstacle is to express the properties that we want to verify. These are either default properties that should hold for all specifications, or they are given as assertions on the UML models, by the user.

Once the formal specification and the properties are written, the newly developed tool runs TLC and parses the returned output. If a property is violated, TLC returns an *error trace* of states and transitions that lead to the violating state. Such a trace can be very helpful in debugging a specification and is a major benefit from using a model checker. Unfortunately, the trace from TLC is in a textual format that is hard to read and impossible to interpret without also looking at the formal specification. Hence the trace in its original form is useless to our intended users. We therefore parse the trace and transform it back into states and steps of the UML activity, so that we can visualize the trace in the activity editor (see Fig. 1.3 for a screenshot of the editor).

In addition to the states and transitions leading up to the violating state, the

trace also gives information about what specific property is violated. We take advantage of this to do specialized analysis on the activity and the trace to look for clues as to what caused a specific violation.

Both the expression of properties and the analysis of a violation is handled by an analysis framework that has been developed in this work. It contains elements of four types: theorems, symptoms, diagnoses and fixes. It is easy to expand the framework by adding more elements.

We are able to automate the analysis because we claim that the majority of mistakes made in specifying reactive systems can be put in a limited number of categories. Hence we focus on providing automated support for the most common problems, leaving the user more time to deal with the more specific ones. For example, many problems stem from the fact that the user is developing a distributed system. The user can easily forget to take into account the communication delays of the asynchronous channels as well as the fact that in-order delivery of messages is not guaranteed.

We refer to the tool that carries out the transformation, model checking and analysis as the *Arctis Analyzer*, or *the analyzer* for short. It is implemented as a plug-in to the Arctis tool suite. Figure 1.2 shows the analyzer in the context of the Arctis tool suite.

We have chosen to limit our scope by not focusing on the performance of the analysis; Every example that is presented in this thesis, is analyzed within one or two seconds[2]. In other words, performance is not yet a practical problem.

Introductory Example

We will briefly demonstrate the use of the analyzer through an example that we will revisit later (in Chapter 7), the *Location Tracker*. Figure 1.3 shows the *Location Tracker* as it appears in the Arctis activity editor. It is a building block that serves to track the location of a mobile client of some sorts. It is instantiated with a target location which is sent to the location server. The client then sends its position to the location server at certain intervals. If the client is within some given distance of the target, the building block will first send a notification on the server side and then to the client, terminating itself.

The building block has an External State Machine, ESM, that acts as a behavioral interface. The ESM of the *Location Tracker* is shown in Fig. 1.4. The syntax will be explained in more detail later, see Sect. 2.1. For now, we will just state

[2]Running Eclipse 3.3 with JVM 1.5.0 on a MacBook Pro with 2 GB memory and an Intel Core 2 Duo processor at 2.2 GHz.

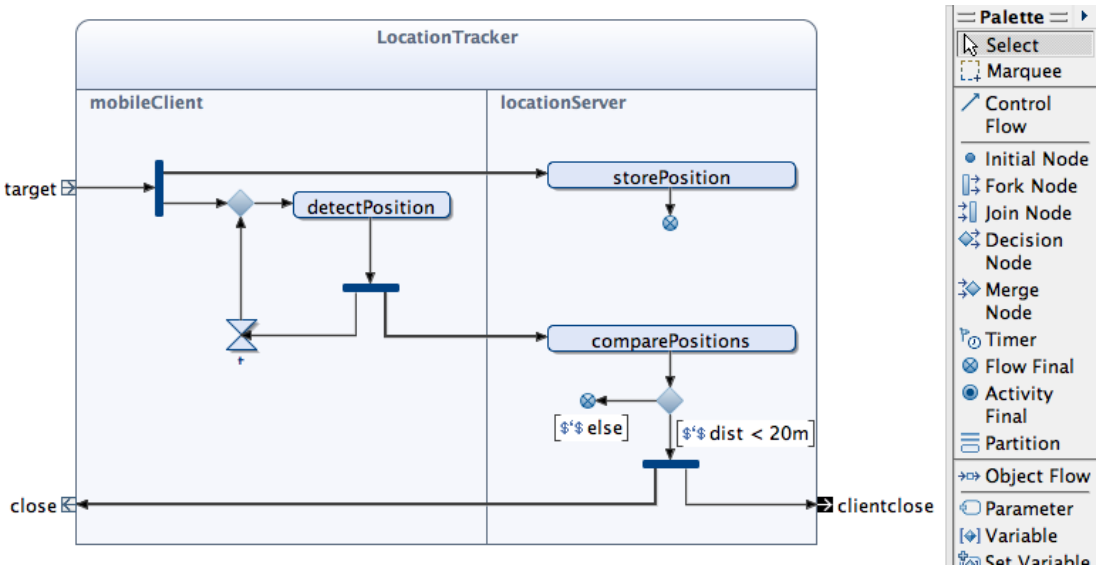


Figure 1.3: The Location Tracker in the Arctis editor

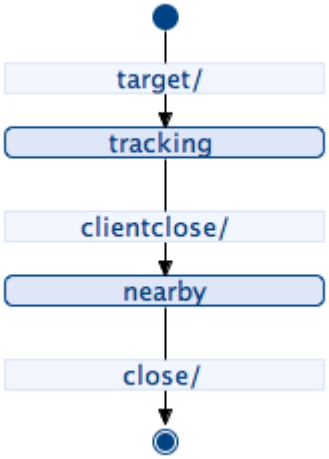


Figure 1.4: The ESM of the Location Tracker

that the labels on the arrows correspond to parameter nodes (the boxes on the outside of the activity). A token arriving or leaving through one of these causes a transition of the ESM.

We will explain the semantics of activities properly in Chapter 2. For now, we say that their semantics is based on tokens flows, like in Petri nets.

The behavior of the *Location Tracker* can be understood by imagining a token entering the activity through the activity parameter node *target*, and following the directed edges. It first encounters a fork node where it is split into two tokens. One will stop as the edge crosses from the *mobileClient* to the *locationServer*, and the other token will pass through a merge node and a call operation action, *detectPosition*, before reaching a new fork node. At this point, one duplicate will stop at the timer, while the other one will stop at the partition boundary. This is because timers and queues between partitions are stateful, whereas merge and fork nodes are not.

After this first step, there are three tokens in the activity. One in the timer, and one in each of the queues that the edges crossing from *mobileClient* to *locationServer* represent. There are now three more steps that are enabled:

- If the token in the top queue arrives at the *locationServer*, it will pass through the call operation action *storePosition* and then be removed by the flow final node.
- If the token in the bottom queue arrives at the *locationServer*, it will pass through the call operation action *comparePositions* and then a decision node. Depending on the result of *comparePositions*, the decision node will either send the token to the flow final node to be removed, or it will let it continue to the fork node. In the latter case, one token will exit through the *clientclose* parameter node, while the other duplicate ends up in the queue from *locationServer* to *mobileClient*.
  - When the token in the queue from *locationServer* to *mobileClient* arrives, it will exit through the *close* parameter node and terminate the activity.
- If the timer expires, the token that was waiting inside will pass through the merge node and behave just like the token that passed through it in the first step.

Arctis provides inspectors that make sure that the specification is syntactically consistent. Once any syntactic problems have been resolved, we run the analyzer to see if there are any problems with the behavior of the specification.

The analyzer creates a TLA$^+$ specification, containing both the behavior as well as the properties that should hold, from the UML activity (and its ESM) representing the *Location Tracker*. The analyzer then runs TLC with the TLA$^+$ specification as input to see if any of the properties are violated. It parses the output from TLC and notifies the user of any problems. All of this is done automatically.

For every property violated, a symptom will be reported. In some cases, the analyzer will also be able to set or at least suggest a more detailed diagnosis explaining not only what is wrong, but what the underlying cause is. Here we use a combination of model checking and syntactic analysis. Since the model checker gives us a violated theorem as well as a trace, we can syntactically analyze the activity starting with the relevant elements and/or we can analyze the trace. Hence the syntactic analysis does not have to be completely general, but can utilize the information about what kind of symptom is found and what elements are affected, increasing both performance and accuracy.

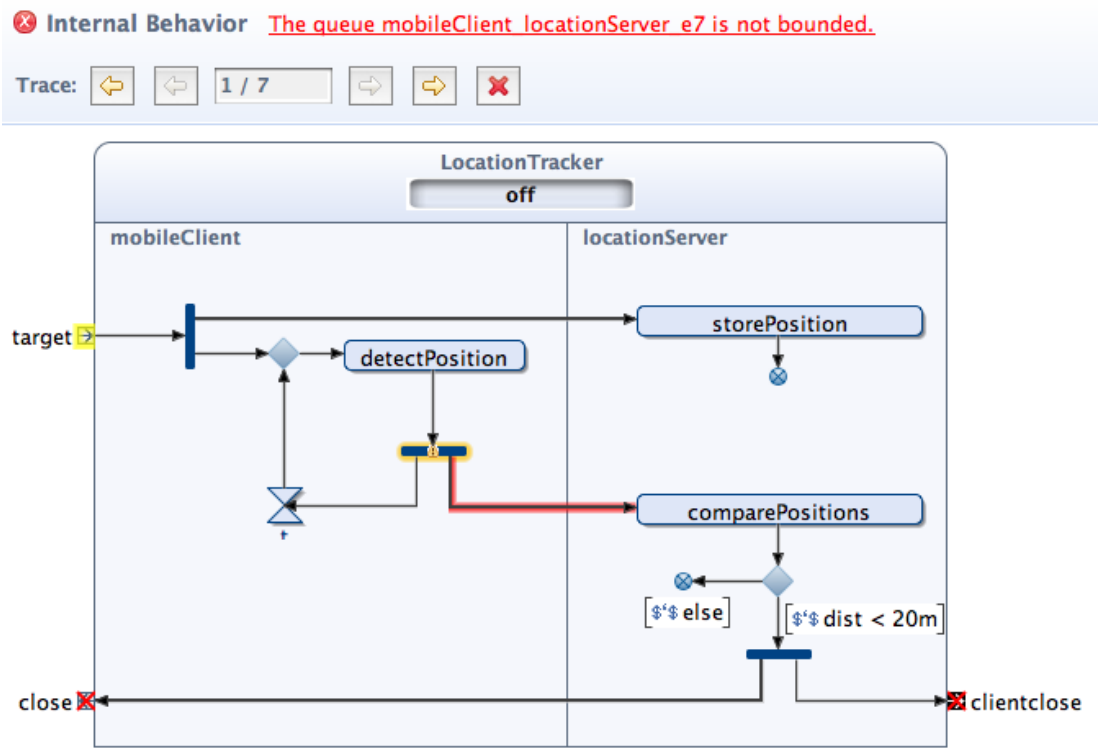


Figure 1.5: The Location Tracker in state 1

The resulting view in the activity editor, once this analysis has been run on the example, is shown in Fig. 1.5. Specifically, it shows the first state of the error trace. There are two problems reported, as seen in Fig. 1.6:

- An error saying the queue between mobileClient and locationServer, *e7*, is not bounded
- and a warning saying an unrestrained producer might be the problem.

The unbounded queue is a symptom. There is only general advice on what to do or look for. The unrestrained producer is a more detailed diagnosis and even supplies an automated fix as shown in Fig. 1.7.

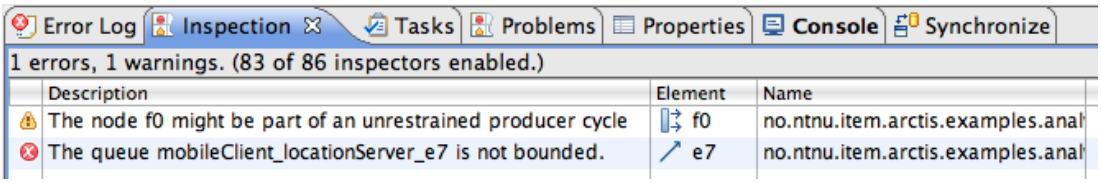


Figure 1.6: Semantic errors of the Location Tracker

Selecting an error or warning, we can open a diagnosis view, Fig. 1.7. In this view, an explanation of what has happened is given as well as instructions for what to do.[3] It may also provide automated fixes that modify the activity to resolve the problem.

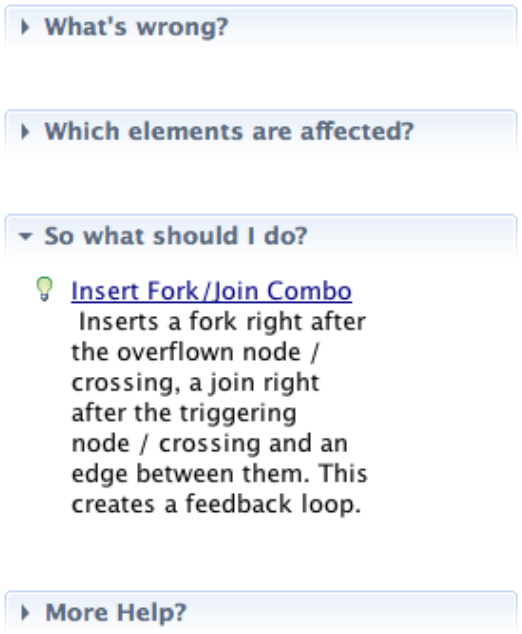
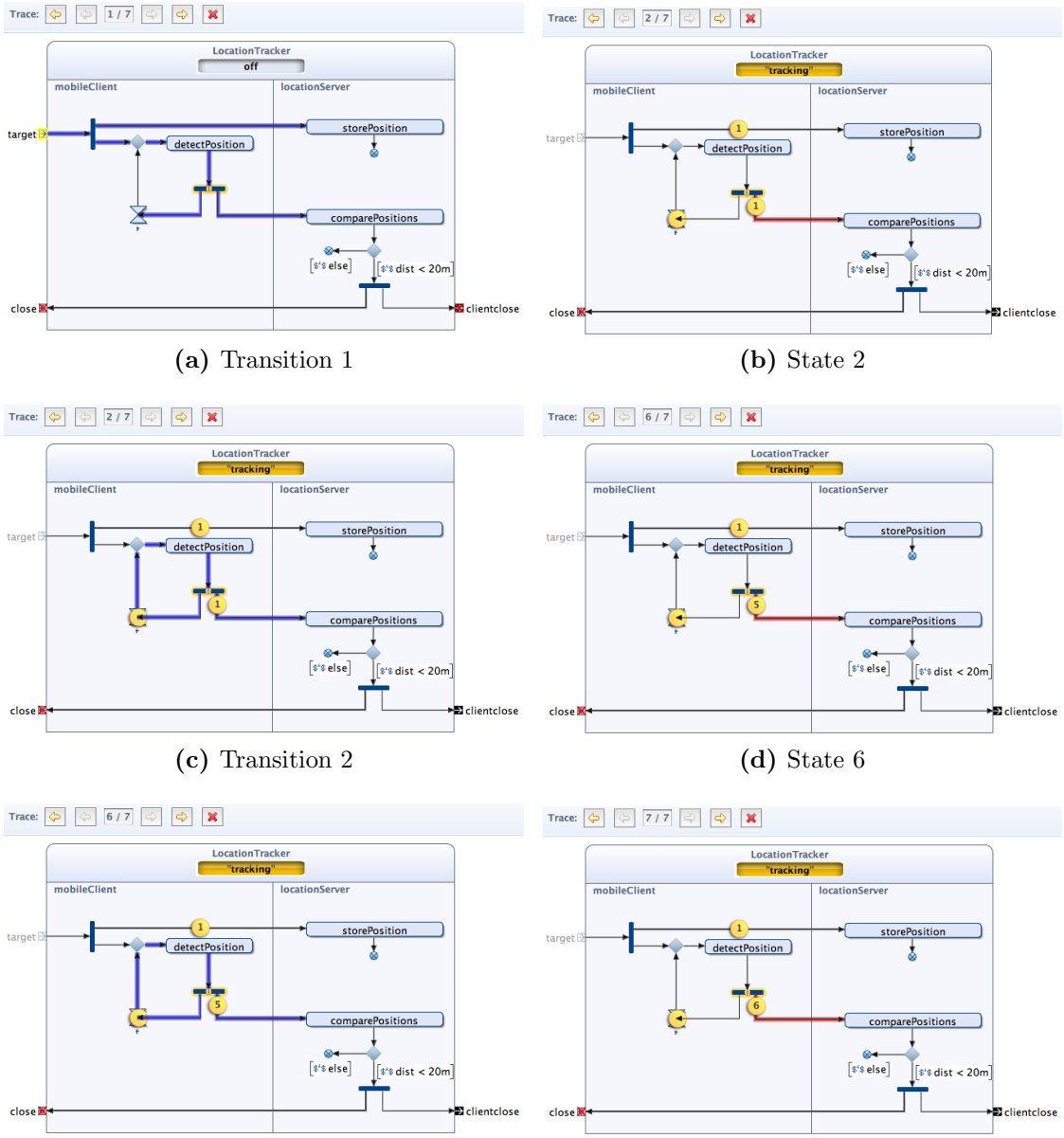


Figure 1.7: Diagnosis view showing the fix of a possible unrestrained producer

[3]In Fig. 1.7, only the "So what should I do?" part is expanded, so as to keep the screenshot somewhat small.



(a) Transition 1 **(b)** State 2

(c) Transition 2 **(d)** State 6

(e) Transition 6 **(f)** State 7

Figure 1.8: Trace for the Location Tracker

We decide to have a look at the trace to see what happened for ourselves. Figure 1.8 shows an excerpt of the visualized trace.[4] A state is visualized as tokens filling various token places in the activity and labels showing the name of the

[4]The visualization is at an early stage of development. The queue tokens are therefore not placed on the partition border. Also, when showing transitions, the border crossing edges should only be highlighted up to the border, whereas now the entire edge is highlighted.

current state of the involves ESMs. The steps are visualized by highlighting the edges that the tokens will travel to reach the next state.

The trace shows how tokens can build up in one of the queues. In state 7, this queue contains six tokens. This violates the property of maximum five tokens in a queue, which is the default threshold set by the analyzer to check for unbounded queues. We can also spot the unrestrained producer that was diagnosed; In the same step as the timer expires, a token is duplicated in a fork node and one of the resulting tokens flows back into the timer, effectively activating it again in the same step. Hence there is nothing to stop this process from repeating itself over and over.

Having reviewed the trace in the activity editor and even spotted the unrestrained producer manually, we decide to apply the fix suggested by the analyzer. The fix creates a feedback loop forcing the producer to wait for the queue to be emptied before inserting another token, as shown in Fig. 1.9.

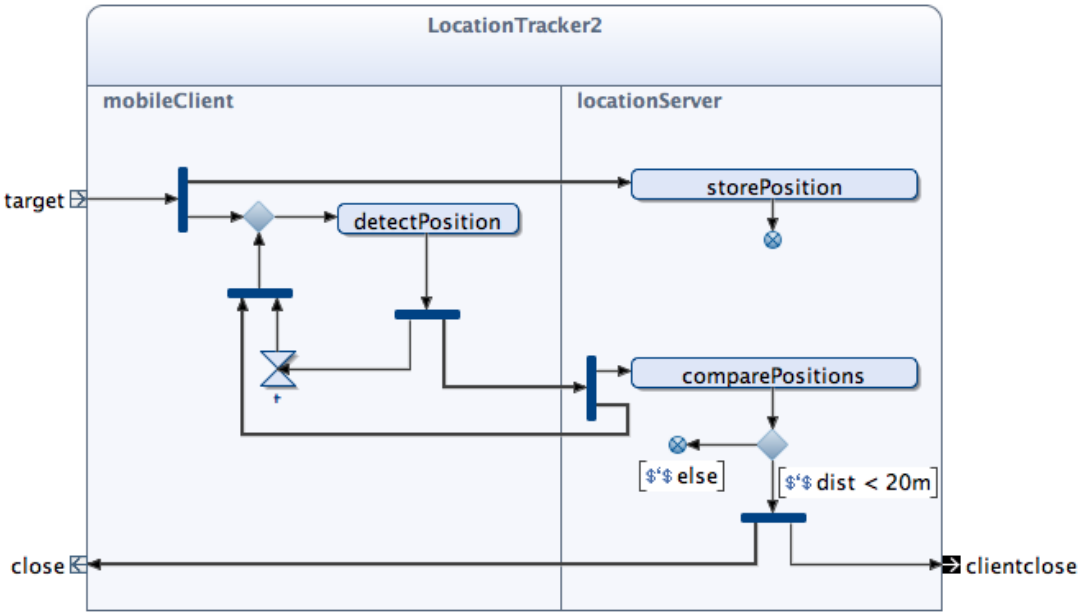


Figure 1.9: The Location Tracker after the fix has been applied

This fixes the current symptom, but will reveal others. More on this in Sect. 7.3.

Overview

There are several advantages to having the analyzer as part of the Arctis tool. The user neither has to learn a new language (TLA$^+$) nor learn to use a model checker (TLC). Hence knowledge of the formal methods domain is removed from the list of prerequisites. Intuitive graphical feedback directly in the model editor shortens the time to find and correct design mistakes. The analysis framework has potential for aiding the user beyond a visualized trace by diagnosing the underlying flaws and even giving automated fixes.

According to the "rush to code" syndrome [SGW94], developers often hurry through the early stages of development, like modeling, so that they can start creating something "tangible", as well as have good tool support. We believe that having tool support for not only syntactic, but also behavioral analysis of our service specifications, will give developers more confidence in their work and motivate to do the early stages of the development properly.

In the next chapter, we will give an overview of the background material that is relevant for the work presented in this thesis. Based on this, Chapter 3 will introduce some new concepts and develop an analysis framework for finding flaws in the specifications. Chapter 4 examines the architecture of the implemented tool, the analyzer, and also points out the changes done to the existing part, the formulator. Chapters 5 to 9 detail the various elements of the analysis framework and provide examples of their use. Chapters 5 to 8 each present a specific category of framework elements, while Chapter 9 presents the remaining ones. Chapter 10 discusses several ideas for future work, and finally, Chapter 11 presents the conclusion.

Chapter 2

Background

Before we go into the details of what has been done in this master's thesis, we will present some background information. To aid us in this, we will use an example, the Hotel Wakeup System, taken from [KSH07][1]. The outermost UML activity of the system is shown in Fig. 2.1.

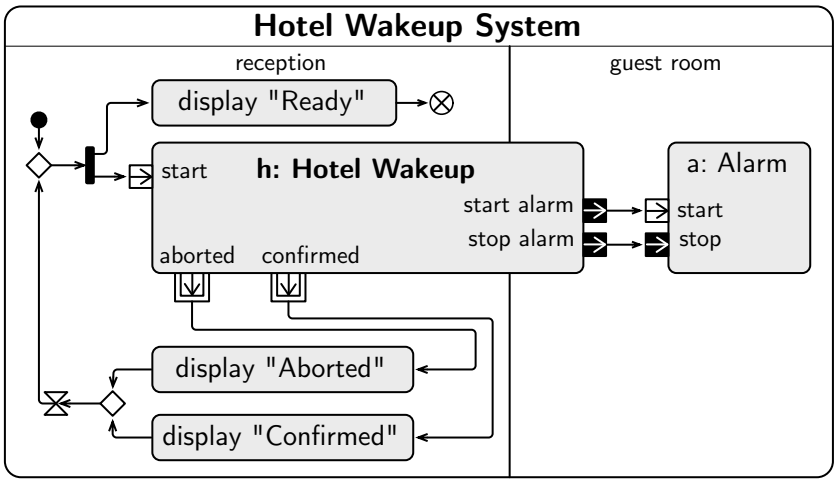


Figure 2.1: The Hotel Wakeup system. Figure from [KSH07].

The system models a hotel that has a semi-automatic wakeup system installed. When the system is started, the message "Ready" will be displayed at the reception. The receptionist manually triggers the alarm in the guest room by pressing a button at the reception. The guest can confirm the alarm by pressing a button in the room. If the receptionist receives no confirmation within reasonable time, he or she can abort the alarm and proceed to manually check whether the room

[1] An earlier version of this example is presented in [Slå07]

is already empty or the guest is simply a heavy sleeper. Whether the alarm was confirmed or aborted will be shown on a display a the reception for a short while, and then the system will be reset to allow for another wakeup.

2.1 The SPACE Approach

The work done within this thesis is based on the approach for modeling reactive software systems, SPACE [KH06, KBH07, KSH07], depicted in Fig. 1.1. In this approach, services are used as reusable building blocks to compose reactive systems. Service specifications are expressed by UML collaborations, focusing on their structure, and UML activities describing their detailed behavior. This service-oriented model can be transformed into a component-oriented model which is again used to generate the code of the executable system.

The structural view of services consists of the participating components and the various collaborations they take part in. A collaboration is denoted by a dotted oval and a title. Collaborations may be composed of other collaborations. This composition is achieved by *collaboration uses* that reference other collaborations. A collaboration which is not composed of others, is called an elementary collaboration.

The structure of the Hotel Wakeup System is shown in Fig. 2.2. The *reception* role of the Hotel Wakeup collaboration, represented by the collaboration use *h*, is assigned to the component by the same name. The guest room is assigned the role of the *room*. In addition, the guest room has an alarm device, represented by a one-part collaboration *Alarm*.

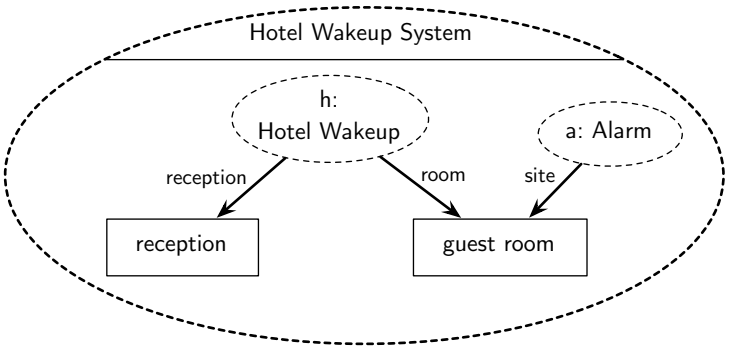


Figure 2.2: Structure of the Hotel Wakeup system.

The UML activities describing the service behavior show the components as activity partitions and the collaboration uses as call behavior actions, which are references to sub-activities [Obj07]. In addition, activities contain logic nodes

like join, fork, merge and decision nodes to synchronize the flows from the various sub-activities.

In Fig. 2.1, we easily recognize the activity partitions *reception* and *guest room* as the components shown in Fig. 2.2. The collaboration uses h of type Hotel Wakeup and a of type Alarm are also represented by correspondingly named call behavior actions.

Activities have a semantics like Petri nets [Obj07]. Tokens flow along the directed edges of the graph and interact with the activity nodes along the way. The activity describes a state transition system where every token movement is a transition from one state to another. We call the path a token can travel in a single transition a *step*.

Figure 2.3 explains some common nodes found in activity diagrams. The semantics of SPACE style activities is a specialization of the semantics of activities as described in the UML Superstructure Specification [Obj07]. For example, a decision node where all outgoing edges are connected to join nodes, is called a waiting decision node and is represented as a filled diamond. Its semantics is that it represents a shared token place between the join nodes. Also, any edge crossing a partition boundary is considered a queue, to represent the channels between the physically distributed components. Currently, we do not model any data. Hence, a decision node is always non-deterministic since we do not know the value of the data the decision is based on.

Technically, both timers and receive signal actions are really accept event actions that refer to a time event or a signal event respectively. We simplify by treating them as different nodes.

Figure 2.4 shows an initial proposal for the activity of the Hotel Wakeup referenced by call behavior action h in Fig. 2.1. The buttons *alert* and *abort* are used by the receptionist to start and stop the alarm while the *confirm* button is used by the hotel guest to confirm that he or she is awake. Note that the dotted lines in this example are not standard for UML activities, but a way of outlining the parts of the specification that will later be replaced as we find flaws in this version.

The activity is started by a token arriving on the parameter node *start*, activating the *alert* button. Once the *alert* button is pushed, it emits a token which is duplicated in the following fork node. One of the tokens activates the *abort* button while the other heads off towards the *room* partition. Here it will make a stop at the partition border, as it cannot reach another partition in a single step. This stop at the border models the time it takes for signal transmission between physically distributed components. Next, the token can arrive in the

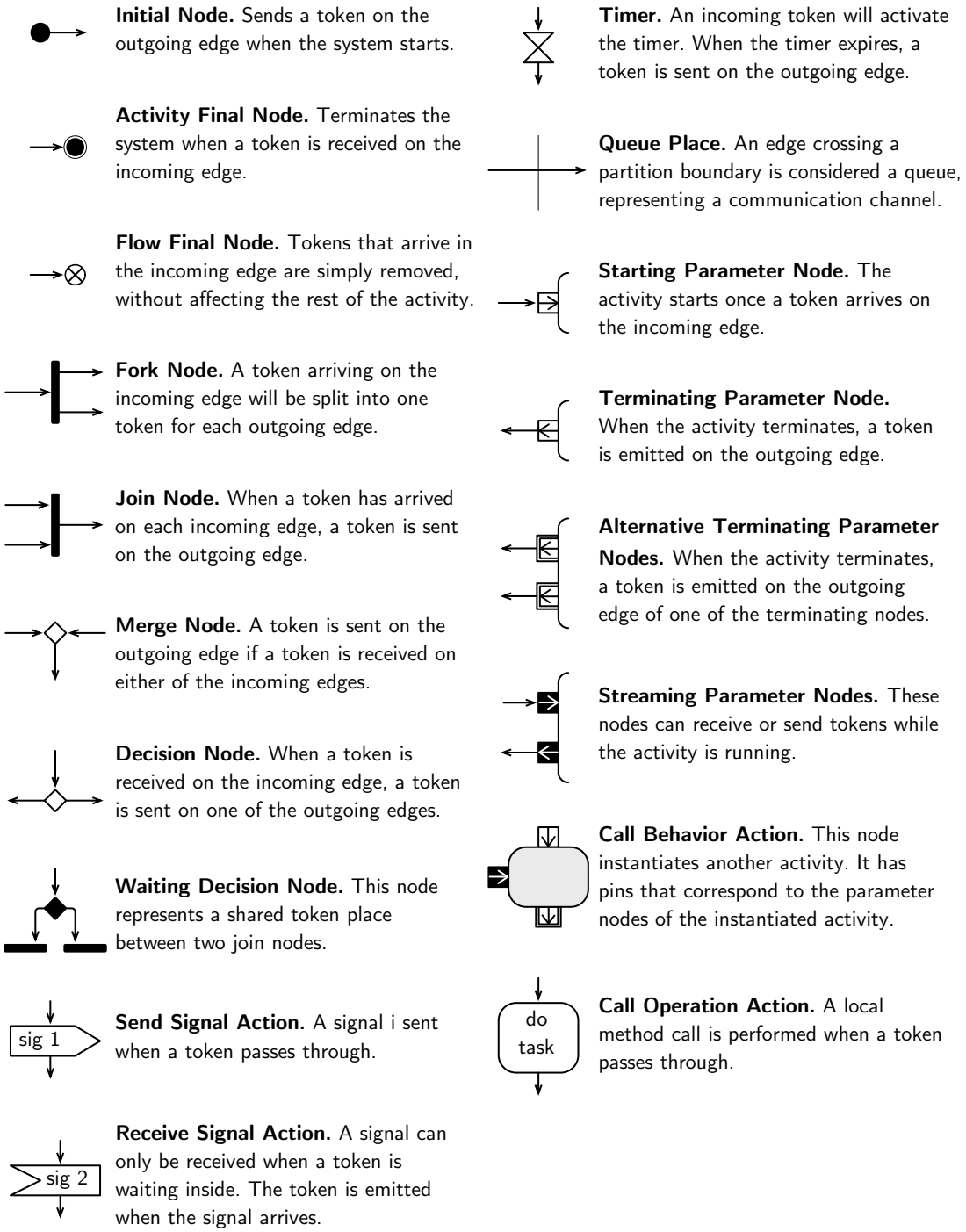


Figure 2.3: Activity node explanations. Figure adapted from [KH06].

room partition to activate the *confirm* button and start the alarm, or the *abort* button may be pushed. Once both buttons are active, the first one that is pressed will deactivate the other as well as stop the alarm.

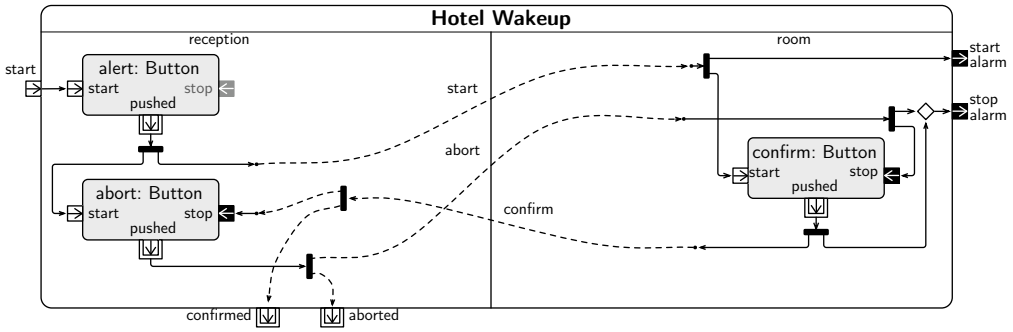


Figure 2.4: Hotel Wakeup, solution 1. Figure from [KSH07].

The nodes *start*, *start alarm*, *stop alarm*, *confirmed* and *aborted* on the outside of the Hotel Wakeup activity are called activity parameter nodes or *parameter nodes* for short. They have their counterparts in the pins that are located on the outside of the call behavior action h in the Hotel Wakeup System activity, Fig. 2.1. It can be easy to mix the concepts of parameter nodes and pins. Put simply, a pin is to a parameter node what a call behavior action is to an activity, a reference to an instance of that type.

We distinguish between system activities and sub-activities. A system activity is an activity without any parameter nodes. Instead it will contain at least one initial node and may contain several activity final nodes as well. Sub-activities are the opposite in that they do have parameter nodes and are hence supposed to be instantiated in other activities through call behavior actions. Sub-activities always have an ESM.

An External State Machine, ESM, is a state machine that acts as a behavioral interface for a sub-activity. The ESMs relieve us from having to look into the implementation details of every sub-activity and also mitigates the state explosion problem when we later model check the specification.

The ESM of a sub-activity tells when a token can be emitted or received on the various parameter nodes. Each transition is labeled with the name of one or two parameter nodes separated by a "/". The parameter node before the slash is called the *trigger* parameter node. This means that the transition is triggered by a token traveling through this parameter node. There may be an *effect* parameter node, in which case it means that a token will travel through this parameter node as a consequence of the transition being triggered. The states of an ESM are simply those states of the sub-activity that can be told apart by an

external observer from looking at which parameter nodes send or receive tokens.

Figure 2.5 shows the ESMs of the sub-activities *Hotel Wakeup*, *Alarm* and *Button*. We see that the ESMs describe the behavior of the sub-activities as it can be observed from the outside.

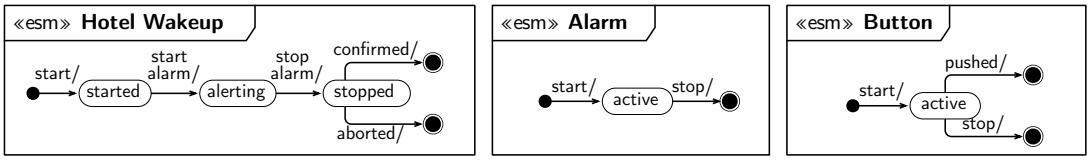


Figure 2.5: ESMs of Hotel Wakeup, Alarm and Button. Figure adapted from [KSH07].

Sometimes, we want to express a reusable piece of local behavior without specifying the details using inner nodes. We do this by a *shallow building block*, a building block that is completely specified by its ESM, and hence has no UML activity [Kra08]. Shallow building blocks do not necessarily need a starting parameter node, as they can be said to be implicitly activated when the surrounding activity partition becomes active. An example is the *Sequencer*, which is introduced in Fig. 2.11.

Note that the general term *building block* encompasses both system activities, sub-activities and shallow building blocks [Kra08].

2.1.1 Properties of Building Blocks

We assume a number of general properties that should hold for all building blocks.

Bounded queues. All queues must be of finite length so they will be implementable.

Respect ESM. When composing building blocks from others, the activity of the surrounding building block must not send tokens into the sub-activity when its ESM is not ready for it.

One-boundedness of inner places. An activity cannot have more than one token in a single inner place.

In addition, we allow the user to specify a range of allowed executions for an action element. For example, that a timer should expire at least once in a single run of an activity.

We will discuss these properties, as well as some new ones developed through this work, in greater detail later in the thesis.

2.1.2 Tool Support

There exists a tool, Arctis (see Fig. 1.2), for editing the UML collaborations and activities. The tool is a plug-in to the Eclipse development platform [Ecl08b], and it takes advantage of the UML2 projects's implementation of the UML metamodel [Ecl08a]. The implementation of the work presented in this thesis, runs as a plug-in to Arctis.

Arctis checks that the specifications are syntactically sound by a number of inspectors. Each inspector checks for a certain kind of problem. Examples are that every node belongs to at least one activity partition and that all nodes, with the exception of initial nodes, have at least one incoming edge. Arctis also provides a framework that makes it easy to add new syntactic inspectors and present their results to the user.

Through a plug-in, the formulator, Arctis can transform UML activities into TLA$^+$. See Sect. 2.2 for information on TLA$^+$ and Sect. 2.3 for information on the plug-in.

Arctis also provides the transformation from collaborations and activities to components and state machines [KH07b]. From this model, code generators [Kra03] can create executable code [KHB06].

2.2 Temporal Logic (of Actions)

In order to do model checking on the specifications created by the Arctis tool, we need a formalism. In [KH07a], the semantics of the UML activities are defined in cTLA [HK00], which is based on Lamport's TLA [Lam02, Lam94], the Temporal Logic of Actions. In this section, we will give an introduction to TLA and its specification language TLA$^+$.

Temporal logic is a variant of modal logic. Modal logics are formal logics that deal with degrees of truth. There are two basic modal operators

- $\Box$ – necessarily
- $\Diamond$ – possibly

In temporal logic these modalities are used to represent aspects of time

$\Box$ – always

$\Diamond$ – eventually

The modal operators of temporal logic can also be combined. Let P be a proposition like "The deadline is approaching".

$\Box\Diamond P$ – "always eventually P", that means, P will be true infinitely often.

$\Diamond\Box P$ – "eventually always P", that means, P will at one point become true and stay that way forever.

The two operators are related as follows: $\Diamond P \equiv \neg\Box\neg P$. Eventually P is equivalent to "not always not P", or "not never P".

TLA is a *linear-time* temporal logic [Pnu77, MP92], meaning that for a property to be true, it must hold in all possible variants of the future. This is unlike *branching-time* temporal logic [CE82] where there are additional operators for "at least one" or "all" possible futures.

In TLA, behavior is expressed as sequences of states, and a state is an assignment of values to all variables. We use the modal operators to express properties about these sequences. When we express a property that should hold in all states, we call it an *invariant*.

Figures 2.6, 2.7 and 2.8 show a TLA$^+$ specification of the Hotel Wakeup System activity shown in Fig. 2.1, as generated by the formulator tool. In TLA$^+$, specifications are structured into *modules*.

Line 1 tells the name of the module, *HotelWakeupSystem*. Line 2 tells that this TLA$^+$ module imports another module that contains the axioms and operators necessary to model the natural numbers and do arithmetic on them.

Next comes a list of variables declared by the VARIABLE keyword. The formulator has generated comments for each of them to explain what they are. Note that in the case that a line is too long, the comment belonging to a variable will be located on the following line. The *_status* variable is a bit special in that it does not represent any particular element, but rather keeps track of whether the entire system is active or not.

Lines 16 and 17 contain the declaration *vars* which is simply a short hand way of referring to all the variables.

```
1  ──────────── MODULE HotelWakeupSystem ────────────
2  EXTENDS Naturals
3  VARIABLE _status
4     Represents the state of the entire activity (active/inactive)
5  VARIABLE display_Ready  Represents a CallOperationAction
6  VARIABLE h  Represents the state of the ESM of a CBA
7  VARIABLE h_counter
8     This variable counts the number of times a CBA is started
9  VARIABLE display_Aborted  Represents a CallOperationAction
10 VARIABLE t  The state of a timer
11 VARIABLE t_counter  This variable counts the number of times a timer expires
12 VARIABLE display_Confirmed  Represents a CallOperationAction
13 VARIABLE a  Represents the state of the ESM of a CBA
14 VARIABLE a_counter
15    This variable counts the number of times a CBA is started
16 vars ≜ ⟨_status, display_Ready, h, h_counter, display_Aborted, t, t_counter,
17         display_Confirmed, a, a_counter⟩

19 Init ≜
20   ∧ _status = "pre_execution"
21   ∧ display_Ready = 0
22   ∧ h = "_initial"
23   ∧ h_counter = 0
24   ∧ display_Aborted = 0
25   ∧ t = 0
26   ∧ t_counter = 0
27   ∧ display_Confirmed = 0
28   ∧ a = "_initial"
29   ∧ a_counter = 0

31 _initialaction ≜
32   ∧ _status = "pre_execution"
33   ∧ _status′ = "executing"  Setting activity to active
34   ∧ display_Ready′ = display_Ready + 1
35      Increasing counter for CallOperationAction
36   ∧ h = "_initial"
37   ∧ h′ = "started"
38   ∧ h_counter′ = h_counter + 1  Incrementing counter for CBA
39   ∧ UNCHANGED ⟨display_Confirmed, a_counter, a, t, display_Aborted,
40                t_counter⟩
```

Figure 2.6: TLA$^+$ specification for Hotel Wakeup System, part 1

```
42 e9 ≜
43   ∧ _status = "executing"
44   ∧ h = "stopped"
45   ∧ h′ = "_initial"
46   ∧ display_Aborted′ = display_Aborted + 1
47       Increasing counter for CallOperationAction
48   ∧ t = 0
49   ∧ t′ = 1
50   ∧ UNCHANGED ⟨_status, display_Confirmed, a_counter, a, display_Ready,
51                t_counter, h_counter⟩

53 e4 ≜
54   ∧ _status = "executing"
55   ∧ h = "stopped"
56   ∧ h′ = "_initial"
57   ∧ display_Confirmed′ = display_Confirmed + 1
58       Increasing counter for CallOperationAction
59   ∧ t = 0
60   ∧ t′ = 1
61   ∧ UNCHANGED ⟨_status, a_counter, a, display_Ready, display_Aborted,
62                t_counter, h_counter⟩

64 e11 ≜
65   ∧ _status = "executing"
66   ∧ h = "started"
67   ∧ h′ = "alerting"
68   ∧ a = "_initial"
69   ∧ a′ = "active"
70   ∧ a_counter′ = a_counter + 1  Incrementing counter for CBA
71   ∧ UNCHANGED ⟨_status, display_Confirmed, t, display_Ready,
72                display_Aborted, t_counter, h_counter⟩

74 e12 ≜
75   ∧ _status = "executing"
76   ∧ h = "alerting"
77   ∧ h′ = "stopped"
78   ∧ a = "active"
79   ∧ a′ = "_initial"
80   ∧ UNCHANGED ⟨_status, display_Confirmed, a_counter, t, display_Ready,
81                display_Aborted, t_counter, h_counter⟩
```

Figure 2.7: TLA$^+$ specification for Hotel Wakeup System, part 2

$e8 \triangleq$
  $\land \_status = \text{“executing”}$
  $\land t = 1$
  $\land t' = 0$
  $\land t\_counter' = t\_counter + 1$ Incrementing counter for timer
  $\land display\_Ready' = display\_Ready + 1$
    Increasing counter for *CallOperationAction*
  $\land h = \text{“\_initial”}$
  $\land h' = \text{“started”}$
  $\land h\_counter' = h\_counter + 1$ Incrementing counter for *CBA*
  $\land$ UNCHANGED $\langle \_status,\ display\_Confirmed,\ a\_counter,\ a,\ display\_Aborted \rangle$

$Liveness \triangleq$
  $\land \mathrm{WF}_{vars}(\_initialaction)$
  $\land \mathrm{WF}_{vars}(e9)$
  $\land \mathrm{WF}_{vars}(e4)$
  $\land \mathrm{WF}_{vars}(e11)$
  $\land \mathrm{WF}_{vars}(e12)$
  $\land \mathrm{WF}_{vars}(e8)$

$Next \triangleq$
  $\lor \_initialaction$
  $\lor e9$
  $\lor e4$
  $\lor e11$
  $\lor e12$
  $\lor e8$

$Spec \triangleq Init \land \Box[Next]_{vars} \land Liveness$

$theorem\_\_status\_h \triangleq \Box((\_status = \text{“pre\_execution”}) \Rightarrow (h = \text{“\_initial”}))$
$theorem\_h\_t \triangleq \Box((h = \text{“stopped”}) \Rightarrow (t = 0))$
$theorem\_h\_a \triangleq \Box((h = \text{“started”}) \Rightarrow (a = \text{“\_initial”}))$
$theorem\_h\_a\_2 \triangleq \Box((h = \text{“alerting”}) \Rightarrow (a = \text{“active”}))$
$theorem\_t\_h \triangleq \Box((t = 1) \Rightarrow (h = \text{“\_initial”}))$

Figure 2.8: TLA$^+$ specification for Hotel Wakeup System, part 3

Lines 19 to 29 contain the *Init* statement. This statement tells what the values of all the variables will be in the initial state. All the inner places and their counters are set to "0" as they are empty to begin with. The call behavior actions h and a are given the special value "*_initial*" to represent the initial node of their ESMs. The *_status* variable is given the value "pre_execution" to show that the activity has not yet started.[2]

The following part contains the TLA actions. Each one describes a transition between two states. We can separate a TLA action into two parts, the preconditions and the effects. The preconditions are all the variable – value pairs where the variables are not primed. Together they describe the set of states where this action is enabled. The effects are the variable – value pairs where the variables are primed (with a $'$). Each effect pair tells the value of that variable in the next state.

We use the ENABLED keyword to refer to just the preconditions of an action. This is useful when writing theorems that are naturally connected. For example, instead of writing $\wedge$ *_status* = "executing" $\wedge\, h$ = "alerting" $\wedge\, a$ = "active", we can just write ENABLED $(e12)$.

The first TLA action[3], on lines 31 to 40, is named _initialaction. This is a special name given to the action that represents a token leaving from all initial nodes in the activity. In our example, there is only one initial node, so the action represents the step where a token leaves the initial node, passes through the merge node and is duplicated in the fork node. One token then executes the *display "Ready"* call operation action before stopping in a flow final node. The other token enters the call behavior action h through the *start* pin, changing its state to "started".

Lines 42 to 51 contain an action that maps to the step of a token leaving call behavior action h through the *aborted* pin, changing its state back to "_initial", executing *display "Aborted"*, passing through a merge node and setting the timer. The action is simply named after the edge that it starts with, *e9*, but the edges are not annotated with names in Fig. 2.1. The order in which the actions are executed has no bearing on the order in which they appear in the TLA module.

Lines 53 to 62 contain an action like the one above. It exits h through *confirmed* and executes *display "Confirmed"* instead.

Lines 64 to 72 contain the action derived from the step of a token leaving h through *start alarm* , changing its state from "started" to "alerting", and entering a through *start*, changing its state from "_initial" to "active".

Lines 74 to 81 contain an action similar to the one above, only it leaves h through

[2]The starting underscores are there to make sure that there are no name collisions, as we do not allow any UML elements to have a name starting with underscore.

[3]In the context of a TLA$^+$ specification, we will often just refer to TLA actions as *actions*.

stop alarm and stops the alarm.

Lines 83 to 93 contain the last action. It is derived from the step where the timer expires, emitting a token. The token passes through the merge node and then does exactly like in the step starting from the initial node.

Having specified all that is allowed to happen is enough to check that something which is not allowed will not happen, so called *safety* properties. However, it is not sufficient to check that something eventually will happen, so called *liveness* properties. There are two types of liveness that we can give to an action A, in TLA$^+$.

- Weak fairness – $\mathrm{WF}_{vars}(A)$.[4] If action A is always enabled from some point on, it will eventually be executed.
- Strong fairness – $\mathrm{SF}_{vars}(A)$. If action A is infinitely often enabled, it will eventually be executed.

By default, the formulator gives each action weak fairness. The *Liveness* statement is shown in lines 95 to 101.

Lines 103 to 109 give a disjunction of all the actions that are valid transitions[5] of the specified behavior. This is used in line 111 where the entire specification is given as a single temporal formula. First there is the *Init* statement. As said before, this gives the initial state(s) of the system. Then comes the so called next-state relation as a second conjunct. The syntax $\Box[A]_{vars}$ means that every transition that change any of the variables in *vars*, must be a transition caused by A. The implication of this is that it is also allowed to have two subsequent states that are the same. This is called a stuttering step. So the next-state relation really says that any two subsequent states must either be identical or the difference between them must correspond to one of the actions listed in the *Next* statement. Finally there is the *Liveness* conjunct to ensure that the behavior will not just be a list of identical states, but that TLA actions will actually be executed.

We express specification properties in the form of TLA theorems. Lines 113 to 117 contain some theorems that the formulator generates for the specification. They are understood in the context of Fig. 2.1 and Fig. 2.5.

The first theorem states that when the activity has not yet started, the call behavior action h must not be started either. This is since when the activity is

[4] Where *vars* is the tuple of all declared variables

[5] TLA transitions are often referred to as *steps*, but as we also use this word to describe token movements in activities in this work, we just call them transitions here.

started, a token will travel from the initial node into the *start* pin of h and this is only allowed when h is not yet started.

The theorem on line 114 states that whenever h is in state "stopped", the timer t must not already contain a token. This is because in this state, h may output a token through pin *confirmed* or *aborted* (see Fig 2.5), both which lead to the timer t. And due to the one-boundedness of inner places, t cannot receive a token when it already has one.

The following three theorems are similar to the above in that they state what the state of the receiving call behavior action must be if the sending node is ready to emit a token.

2.2.1 Refinement

In addition to writing theorems for checking safety and liveness properties, TLA allows us to check whether one specification is a *refinement* of another. We can also say that we check if Spec1 *implements* Spec2. Spec1 is the more detailed specification whose states are mapped onto the state space of Spec2. Implementation is expressed using the implication operator, $\Rightarrow$.

To check refinement, one needs to provide a refinement mapping. This is a mapping that expresses the variables of the abstract specification in terms of the detailed one. For a detailed introduction to refinement, see [Lam96].

Refinement is useful when one wishes to check whether a detailed implementation conforms to a more abstract interface specification. It is also useful if one needs to check more complex properties, as one can express such properties as abstract specifications and then check if the detailed specification implements them.

2.2.2 TLC

The purpose of creating a TLA$^+$ specification is so we can model check it. For this we use the model checker TLC [YML99, Lam02]. TLC takes two files as input, A *spec.tla* file containing the specification and a *spec.cfg* file containing some configuration parameters for the model checking.

TLC will default to checking the entire state space of the specification, checking that every property is satisfied. If any state (or sequence of states) violates a property, it returns a textual trace of all the states leading up to the violation. Such a trace specifies the value of every variable at every state as well as which TLA action triggered the transition between every pair of successive states.

TLC will also give the name of the violated theorem, but only if it is an invariant. That is, any theorem expressing a liveness property, usually by means of the $\Diamond$ (eventually) operator, is not identified by name. In this case, the message "*Error: Temporal properties were violated*" will be given instead of the theorem name. Later in the thesis, we describe how to work around this issue.

A deadlock is a state where none of the actions specified in the next-state relation are enabled. TLC defaults to reporting deadlocks as violations and showing a trace to them. We turn this off by use of the -deadlock parameter, since many of our specifications are supposed to terminate at some point. Instead, we introduce our own deadlock theorem (Sect. 9.3).

TLC also has a parameter -continue to make it continue model checking even though it encounters a violation. It then reports all found violations in the end. A problem with using this functionality is that the behavior after the first violation may be irrelevant, as fixing the first violation may change the following behavior.

2.3 The Formulator

The architecture of the original formulator is shown in Fig. 2.9. The job of the formulator is to transform a UML activity into a TLA$^+$ specification. The traverser class traverses the activity and calls corresponding methods in the TLA module class as it encounters various nodes and edges. Theorems are created directly in the TLA module. The TLA module consists of several lists and maps containing text strings. It also contains TLA action objects that represent TLA actions. Once the traversal is complete, the formulator tells the TLA module to return the entire TLA$^+$ specification as a text string. This is then written to a file and the user can use this file as input when running TLC.

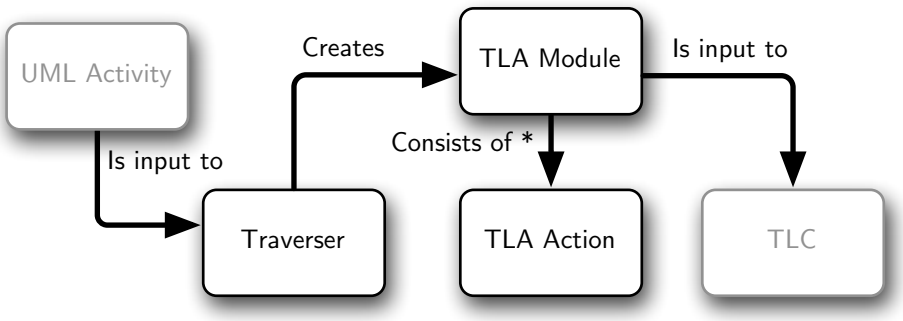


Figure 2.9: Formulator architecture

The formulator can only transform complete system activities. Sub-activities are encapsulated in an environment activity that contains initial and activity final nodes. Sub-activities are then checked by adding a stereotype ≪specimen≫ to the referencing call behavior action, to tell the formulator to traverse into the sub-activity and create a TLA$^+$ specification for both the environment activity and the sub-activity at once.

When transforming and model checking the Hotel Wakeup System in Fig. 2.1 with the ≪specimen≫ stereotype applied to call behavior action *h*, we get a textual trace leading up to a state where a theorem is violated. We present a visualized version of this trace in Fig. 2.10.

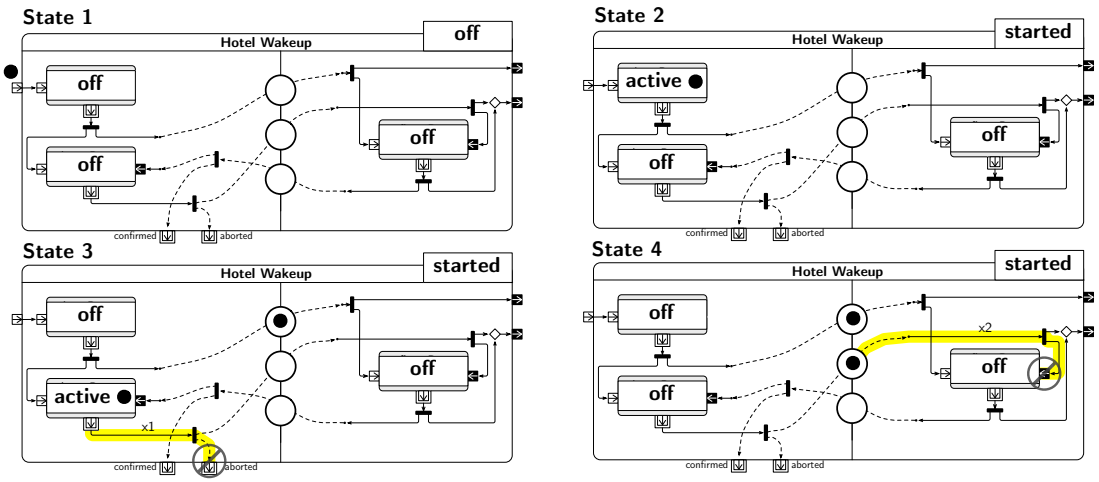


Figure 2.10: Trace for solution 1 of Hotel Wakeup. Figure from [KSH07]

In state 1, the activity is not yet active. The token has yet to arrive through the *start* parameter node. Note that in visualizing the trace, we use the string "off" to denote the initial state whereas in TLA$^+$ it is denoted by "_initial".

In state 2, the activity has been started and its ESM has changed state to "started". The *alert* button is also started and has changed state to "active".

State 3 shows how the *alert* button has been pushed, activating the *abort* button. A token is also waiting to cross over to the *room* partition. In this state, a theorem is violated. If the *abort* button is pushed, a token will leave through the *aborted* parameter node. Yet, the ESM of *Hotel Wakeup* does not allow this when its state is "started".

For the sake of brevity, we also show the next state, state 4, in Fig. 2.10. The violation shown here would occur even if we first fixed the violation in state 3. Here, the *abort* button has been pushed and a second token is waiting to cross

over into the *room* partition. Since we do not guarantee in-order delivery of tokens traveling from one partition to another, this last token may arrive before the first. It will then reach the *confirm* button on the *stop* pin, before the *confirm* button is ever started.

Figure 2.11 shows a second attempt at a solution for the *Hotel Wakeup* activity. We fix the violation in state 3 by moving the fork node that creates an extra token to be sent to the *aborted* parameter node, to the room partition. This makes sure a token will leave through the *stop alarm* parameter node first. The second violation is fixed by introducing a sequencer. This is a shallow building block that takes two inputs and simply makes sure that they are sent on in a specified order. In this case, *o1* before *o2*.

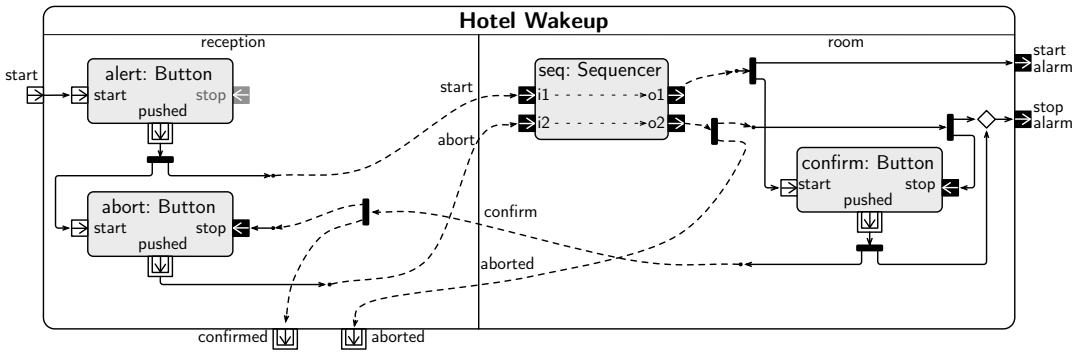


Figure 2.11: Solution 2 of Hotel Wakeup. Figure from [KSH07]

We run the transformation and TLC again. The result is another violation. The trace is visualized in Fig. 2.12. We start from state 3 as states 1 and 2 correspond to states 1 and 2 of the first attempt. We name the queues q1 to q4, starting from the top of the diagram.

State 3 shows a token waiting in q1 and the *abort* button waiting for a push or a stop. State 4 shows that the token has been consumed from q1, passed through the sequencer and been duplicated. One token has then left the activity through parameter node *start alarm*, changing the state of the ESM to "alerting", while the other has started the *confirm* button.

State 5 shows that the *confirm* button has been pushed and a token is hence waiting in q3, to stop the *abort* button and terminate the activity through parameter node *confirmed*. A duplicate of the token emitted from the button also left the activity through the *stop alarm* parameter node, changing the ESM state to "stopped".

State 6 shows that the *abort* button is pushed before the token is consumed from

q3. This puts a token in q2, and it also makes it illegal for the token in q3 to now arrive at the *stop* pin of the *abort* button. It is also illegal for the token in q2 to try to stop the *confirm* button which is already stopped, as well as leaving through *stop alarm* now that the ESM is in state "stopped".

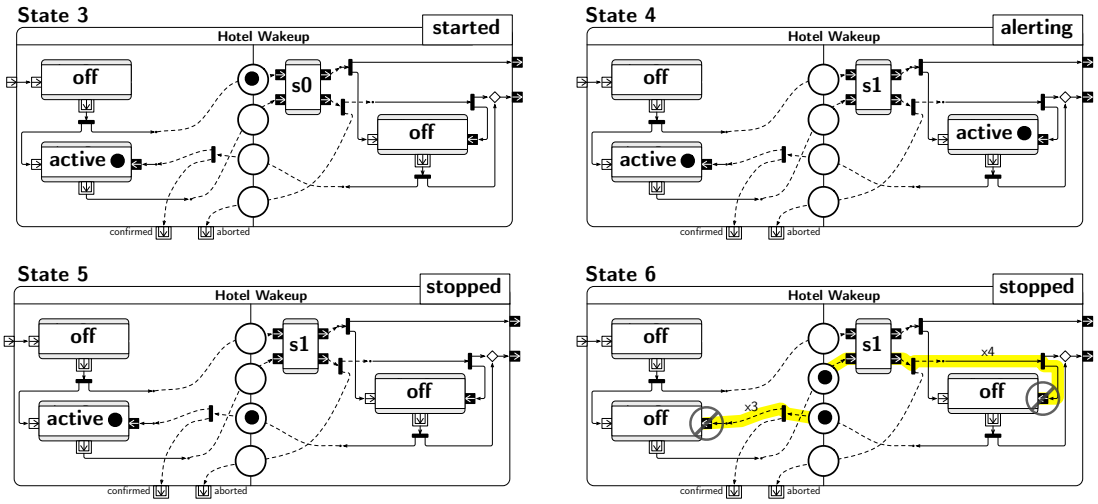


Figure 2.12: Trace for solution 2 of Hotel Wakeup. Figure from [KSH07]

Looking at the trace, we see that we have a situation where both sides may take initiative before being stopped by the other, due to the delay of the communication channels. We call this a mixed initiative situation. Looking through the library of building blocks, we find a Mixed Initiative Secondary Starter (MISS) block. Its purpose is to solve exactly this kind of problem, by giving one side priority over the other. The non-priority side must then be able to handle that it could be overruled even after taking initiative itself. The ESM of the MISS building block is shown in Fig. 2.13.

Figure 2.14 shows the *Hotel Wakeup* activity after we have applied the MISS block. The primary taking initiative alone is distinguished from the secondary being overruled by how we must stop the *abort* button in the first, but not in the latter case. We generate another TLA$^+$ specification and run TLC on it. This time, there are no violations reported.

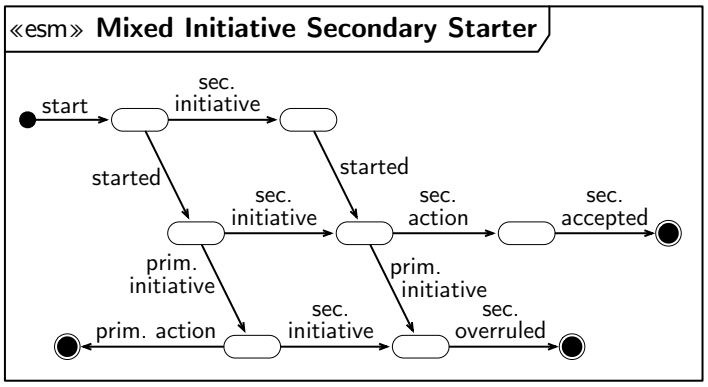


Figure 2.13: The ESM of the MISS building block. Figure from [KSH07]

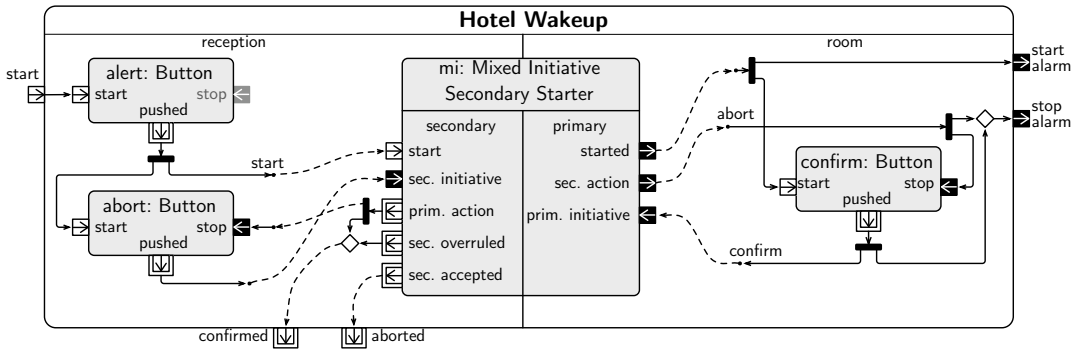


Figure 2.14: Solution 3 of Hotel Wakeup. Figure from [KSH07]

2.4 Related Work

The application of formal methods in combination with tools is also done with OMEGA [Hoo02], FUJABA [BGHS04] and HUGO[BBK+04]. However, they do not focus on UML activities such as we do, rather state machines and sequence diagrams. Also, they do not present the error trace on the UML model or attempt at making any automatic interpretations.

vUML [LP99] automatically creates PROMELA specifications from UML state charts and can create a sequence diagram from the error trace provided by the SPIN model checker. Like us, they mostly check general properties that the users do not have to specify themselves, but they also allow for marking a state as an error state or a goal state.

Theseus [GCKK06] visualizes error traces from the SPIN and SMV model checkers onto UML state diagrams, and also generates UML sequence diagrams from the trace. Theseus is one component in a round trip engineering tool suite that

also supports transforming UML to PROMELA and formulating properties as LTL or CTL formulas. Theseus is independent of a certain UML editor, but is currently only implemented as a plug-in to the ArgoUML CASE tool.

There are approaches where UML activities are used. In [Esh06], the NuSMV, a symbolic model checker, is used to check the consistency of activity diagrams. In [GM05], UML activities are checked by the SPIN model checker [Hol03]. In [DS03], they transform UML activities into the π-calculus and use modal mu-calculus to express safety and liveness properties. This is then checked by the MWB tool [VM94]. In [Stö05], UML 2.0 activities are transformed into Colored Petri Nets, hence enabling analysis by Petri net tools.

In [FF06], a method is proposed for visualizing soundness violations of workflow Petri nets [Aal98] in the WoPeD tool [Wop08]. The violations themselves are detected by the Woflan tool [Aal99]. Soundness violation is separated into five violation classes and a list of eleven error reasons is presented. Some of these *error reasons* map to our *symptoms* while others map to our *diagnoses*. In the case of a violation, the violating node(s) is (are) highlighted with the violation class and the error reason. If a violation is caused by a certain firing sequence of the net, an animation can be shown.

The above approaches all use UML activities or Petri net (derivatives), but not for distributed reactive systems with asynchronous communication. Rather it is applied for business processes and they assume synchronous communication or a central clock.

In [SGK+05], activity charts (their version of activity diagrams) are animated, showing how tokens flow when a system is running. However, there is no formal verification, just a simulation engine. There is also no concept of distributed components, everything is local to the same machine.

There are some approaches that utilize multiple traces to locate the cause of an error. In [GV03], Java PathFinder [VHBP00] is used to find both traces that lead to a violation of a Java program and traces that do not. Three ways of computing the most relevant differences between the violating and the non-violating traces are presented. One example is that they locate any transitions that exist in all violating traces and in none of the non-violating traces. They cannot, at the time of writing, handle infinite traces (traces with loops), so they can only check for safety properties.

Similarly, [BNR03] uses the SLAM toolkit [BR01] to verify software programs written in C. They exclude the violating transition of each model checking run from the next, hence collecting several different traces for an error. They then report which lines were executed in the error traces compared to successful traces. Like [GV03], they only check for safety properties.

Naturally, these approaches do not visualize the trace, as they are working directly with code. They simply give a list of code lines that should be inspected closer. They also differ in that they are using several traces instead of combining one trace with syntactic analysis, like we do.

The focus of this work is to interpret error traces and find flaws automatically. To the best of our knowledge, there are no other approaches that offer animation and interpretation of error traces in terms of UML activities. And with the exception of the WoPeD tool, we have not found anyone that try to give the user an answer as to what exactly the problem is. Finally, we have found no tools that can suggest automatic fixes based on the feedback from a model checker. To the best of our knowledge, we are unique in this respect.

Chapter 3

Identifying Specification Flaws: Towards a Framework

In this chapter we will look at how we find flaws in the specifications we create in the SPACE approach, and how we create a framework for dealing with them.

3.1 What is a Flaw?

A *flaw* is the part of a specification that causes undesired behavior. Said in another way, it is what causes a specification not to behave as intended. So what is desired behavior? What is the intention of the user?

A computer cannot reason about what the users really wants. All it can do is to compare two behaviors and tell if they are consistent. Hence, the first step is to decide how to express desired behavior in a practical manner. We could write a complete specification that behaves in the way we want the system we are currently specifying to behave. But how do we know that this one is any more correct than the first one? We need to be able to express an abstract version of the intended behavior that we can easily understand and then check that the detailed version is actually doing the same. In fact, we want to be able to give just subsets of this abstract behavior, so we do not have to think of everything at once. We solve this by giving a set of properties that must hold for the specification.

So what properties must hold for a specification to give an end result with the intended behavior? There are two ways to determine this.

General properties. Since the user is using the Arctis tool to specify a system or service, we can assume that the user intends for it to comply with

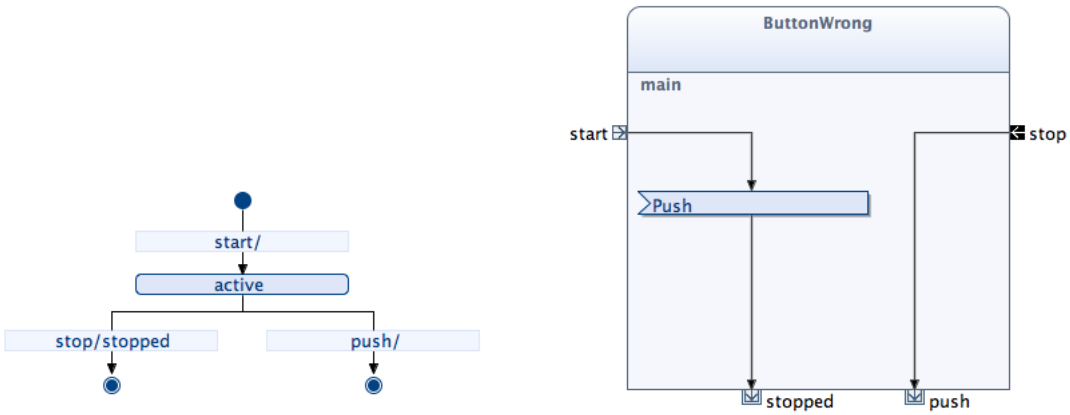
some general properties that apply to all SPACE specifications and that are assumed to hold for the transformation to component models [KH07b]. Examples are: one-boundedness of inner places, bounded queues and freedom from deadlocks.

Assertions. Assertions are explicit redundant information that the user can append to the UML model in the form of UML stereotypes. These give information like how many times an action should be executed or whether two actions are supposed to be mutually exclusive.

It is hard to decide just how many and what assertions a specification should have. One extreme would be to assert every property, but this would just be another way of creating the specification. Instead we limit the assertions to those that are most abstract. These are useful as they provide a way to check that the detailed specification is consistent with the intention of the user, expressed in the assertion.

3.2 Towards an Analysis Framework

We now know that we can express properties that should hold for the specifications we create. This section will introduce some new concepts and outline a framework for identifying and correcting flaws. To illustrate the following concepts, we look at the example of the *Button* building block. Figure 3.1 shows both its ESM and an (erroneous) activity as displayed in the Arctis editor.



(a) ESM **(b)** Activity

Figure 3.1: Button Building Block (with a flaw)

The Button is a very simple building block designed to listen for an external signal, a push. A button can also be stopped meaning that it will stop listening for the push.

Figure 3.2 shows a flowchart of how the analyzer works. It may be useful to compare this to the architecture presented in Sect. 4.1. We will now discuss the process from start to end, introducing new concepts and giving examples as we go.

3.2.1 Phase 1: Transformation and Detection

The first phase of the analyzer's work is to transform the UML specification into TLA$^+$, generate theorems and detect any violations. We will now describe these three tasks in more detail.

Creating the TLA$^+$ Specification

The process of checking a specification for flaws begins by transforming the UML activity of the Arctis specification into a TLA$^+$ specification. The result is a TLA module. An excerpt from the TLA module of the Button specification is shown in Fig. 3.3[1].

The specification starts with a list of variables:

r0 represents the receive signal action that is waiting for the push.

r0_counter is a counter that is increased every time *r0* receives a signal.

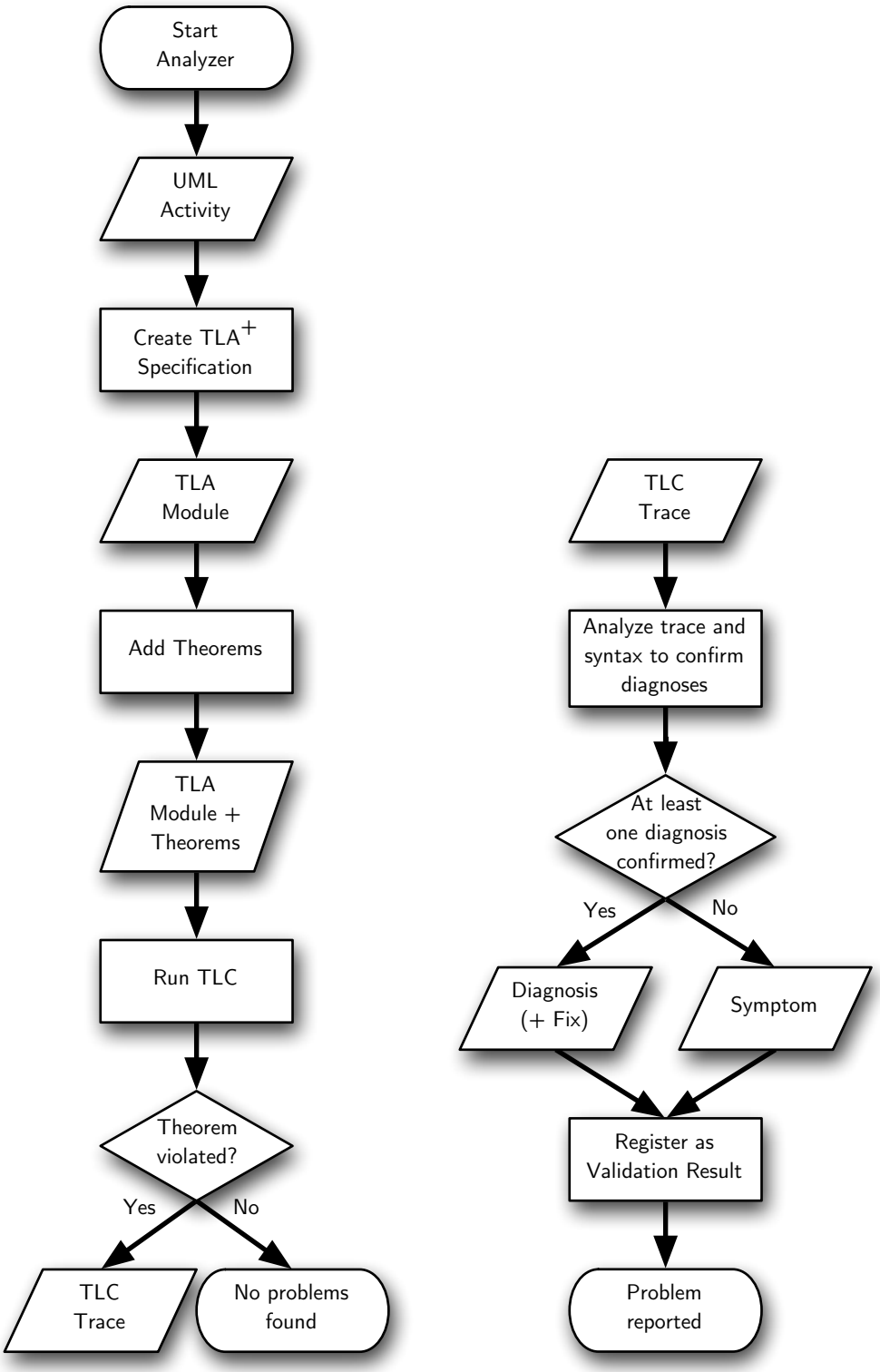
ButtonWrong represents the state of the ESM as seen in Fig. 3.1(a).

_status is a variable that we add to keep track of whether the activity is about to start executing, is executing or has terminated (is done executing).

vars is not really a variable, but a shortcut for writing all the variable names later on in the specification. However, its use is not shown further in the excerpt in Fig. 3.3.

After the variables comes the *Init* statement which allocates the initial values of all the variables. The receive signal action *r0* is empty and its counter is 0. The

[1]There is a mistake in this version. This is why it is named *ButtonWrong*



(a) Phase 1 **(b)** Phase 2

Figure 3.2: Work flow of the Analyzer

MODULE *ButtonWrong*

EXTENDS *Naturals*
VARIABLE $r0$ The state of a receive signal action
VARIABLE $r0\_counter$ Counter for receive signal action
VARIABLE *ButtonWrong* Represents the state of the *ESM* of an Activity
VARIABLE *_status*
Represents the state of the entire activity (active/inactive)
$vars \triangleq \langle r0,\ r0\_counter,\ ButtonWrong,\ \_status \rangle$

$Init \triangleq$
$\land\ r0 = 0$
$\land\ r0\_counter = 0$
$\land\ ButtonWrong =$ "_initial"
$\land\ \_status =$ "pre_execution"

$start\_\_initial\_active \triangleq$
$\land\ r0 = 0$
$\land\ ButtonWrong =$ "_initial"
$\land\ \_status =$ "pre_execution"
$\land\ \_status' =$ "executing" Setting activity to active
$\land\ r0' = 1$ Getting ready to receive signal
$\land\ ButtonWrong' =$ "active" Token arrived through start
$\land$ UNCHANGED $\langle r0\_counter \rangle$

$e1 \triangleq$
$\land\ \_status =$ "executing"
$\land\ r0 = 1$ Ready to receive signal
$\land\ r0\_counter' =$ IF $r0\_counter < 2$ THEN $r0\_counter + 1$ ELSE 2
Incrementing counter
$\land\ r0' = 0$ Signal received
$\land$ UNCHANGED $\langle \_status,\ ButtonWrong \rangle$

$stop\_active \triangleq$
$\land\ \_status =$ "executing"
$\land\ ButtonWrong =$ "active"
$\land$ UNCHANGED $\langle \_status,\ r0\_counter,\ r0,\ ButtonWrong \rangle$

Figure 3.3: Excerpt of TLA$^+$ specification of erroneous Button building block

starting node of the ESM is represented by the string "_initial" and the *_status* variable is "pre_execution" which means it is not yet started.

After the Init statement, three TLA actions follow:

start__initial_active represents a token arriving through the parameter node *start* and a token filling the receive signal action *r0*. This TLA action also changes the state of the ESM from "_initial" to "active".

e1 represent the signal being received by *r0* and a token leaving through *stopped*. However, since there are no ESM transitions that match this behavior, this action does not alter the state of the ESM.

stop_active represents a token being received on parameter node *stop*, something which the ESM only allows when in state "active". But since the token exits through parameter node *push*, there is no match found in the ESM for this TLA action either.

Adding Theorems

Every property of a specification can be transformed into a TLA *theorem* which is then added to the TLA module. We give a few examples of the theorems created for the ButtonWrong activity.

The TLA action *stop_active* must never be enabled for execution. This is because there is not a single transition in the Button ESM that matches it. (A matching transition would have to have the label *stop/push*). If it were to be enabled, the activity would not be consistent with its ESM. To detect this particular inconsistency, we write the theorem

$$theorem\_stop\_active\_neverEnabled \triangleq \Box\neg \text{ENABLED } (stop\_active) \tag{3.1}$$

Whenever the ESM is ready to accept a token through parameter node *start*, the TLA action implementing the transition from the initial state to "active" will be enabled. This is also necessary for the ESM and the activity to be consistent and is expressed by the theorem

$$\begin{aligned} &theorem\_enabled\_start\_\_initial\_active \triangleq \\ &\Box((\_status = \text{“pre\_execution”} \wedge ButtonWrong = \text{“\_initial”} \Rightarrow \\ &\text{ENABLED } (start\_\_initial\_active)) \end{aligned} \tag{3.2}$$

Whenever the ESM is in a state where it accepts a token on *start*, the receive signal action *r0* must be empty. This is expressed by the theorem

$$\begin{array}{l} theorem\_\_status\_ButtonWrong\_r0 \triangleq \\ \Box((\_status = \text{“pre\_execution”} \wedge ButtonWrong = \text{“\_initial”}) \Rightarrow (r0 = 0)) \end{array} \tag{3.3}$$

A violation of the theorem would be a violation of the one-boundedness property, meaning that the activity cannot be correctly transformed to state machines as the next step towards a deployable system (see Fig. 1.1).

Running TLC

The TLA module and the theorems are fed to TLC which either returns “no errors found” or a trace. The trace is returned in the event of a theorem violation and also has information on which theorem was violated (with the exceptions noted in Sect. 2.2.2).

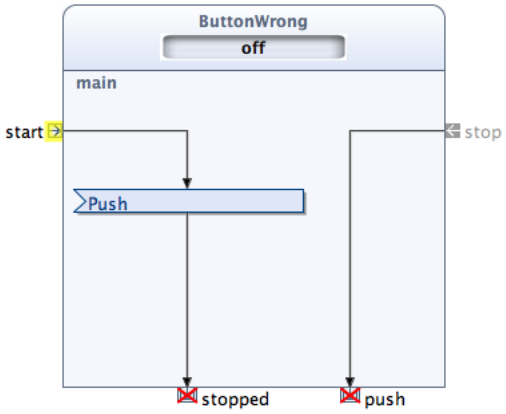
Running TLC on our button example, gives the trace shown in Fig. 3.4. The analyzer converts the textual trace, which is expressed in terms of the TLA$^+$ variables, to a graphical representation in terms of the UML elements that constitute the activity. Being such a simple example, the trace is not very extensive, but it shows how we arrive in the state where *r0* is ready to emit a token. In this state, the TLA action *stop_active* is enabled even though it has no matching ESM transitions.

3.2.2 Phase 2: Determining the Problem

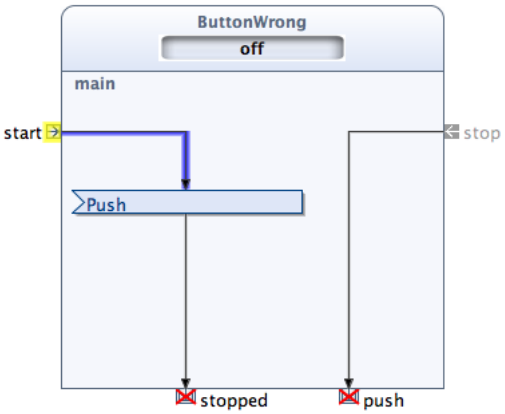
If TLC finds a theorem to be violated, we enter the second phase of the analysis. In this phase, we analyze the activity and the trace from TLC looking for properties that are relevant to the violated theorem.

When a theorem is violated, we can always present the user with a *symptom*. A symptom describes what the problem detected by the violated theorem is, but not the cause of it.

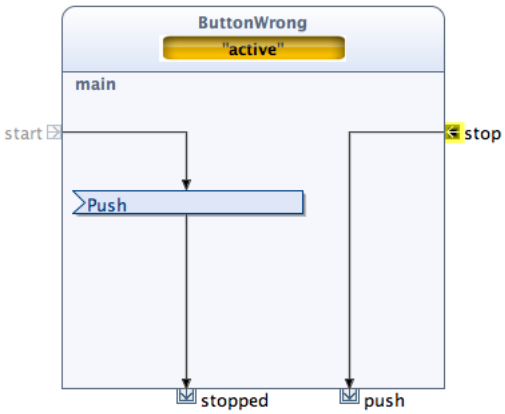
Once we have a specific symptom, we can analyze the trace and the syntax of the activity to try to pinpoint what is the underlying reason for it. Based on the results, we may be able to find one or more *diagnoses*. A diagnosis represents a candidate for what the cause of the problem may be. We say that it is confirmed, if all the criteria for setting the diagnosis are met. There can be several diagnoses



(a) State 1



(b) Transition 1



(c) State 2

Figure 3.4: Visual Trace of Button Building Block

for a single flaw, yet only one of these will be the correct one. Since what is correct depends on the user's intentions, the correct diagnosis is simply the one that the user chooses to act upon.

In the button example, a *Missing ESM Transition* diagnosis is set. Whenever the activity wants to output a token and the ESM does not allow it, we can solve it by adding a new ESM transition. Hence this is a candidate for the flaw, that an ESM transition was simply missing. We will discuss this diagnosis further in Sect. 5.3.2.

Each symptom can define its own logic for what to report depending on what diagnoses are confirmed. Often, the symptom is withheld if an accurate diagnosis can be set. This is to avoid overloading the user with too much redundant information. The symptom and/or diagnosis is wrapped in a validation result

which is an object that the Arctis editor can present to the user.

Both symptoms and diagnoses, may contain textual advice for what the user should do. An example could be "Make sure to have feedback from the consuming side to the producing side of this queue". If a diagnosis is set, we can sometimes suggest a correction that can be applied automatically to the specification. We call such corrections *fixes*. A fix corresponding to the above advice might be something like "Automatically create a feedback loop from consumer to producer".

There is a variant of fixes that give *grants*. A grant is a piece of information that is added to alter the way the model is interpreted. The only existing example so far, is that we can grant that one partition will consume tokens from a queue faster than the other partition will put tokens into the queue.

In our current example, the mere presence of the symptom and diagnosis is enough to cause us to have a closer look at the Button(Wrong) activity. The parameter nodes *push* and *stopped* have been swapped. After the flaw is corrected, a second run of the analyzer reports no problems.

3.3 Preview: Analysis of SPACE Specifications

The proposed analysis framework is independent of a certain specification style. For example, we could also use it on state machine models. However, the focus of this work has been on SPACE specifications, and we will now give a preview of the contents of the framework specifically developed for the SPACE approach.

Figures 3.5, 3.6 and 3.7 shows the theorems, symptoms, diagnoses and fixes we found in our work so far. The figures demonstrate that the relationship between theorems, symptoms, diagnoses and fixes is not only one-to-one. For example, there are several theorems that if violated, will give an *Activity Violates ESM* symptom, and several symptoms can be caused by an *Unrestrained Producer*.

After Chapter 4 which deals with the analyzer from an implementation perspective, Chapter 5 will cover the theorems, symptoms, diagnoses and fixes shown in Fig. 3.5. In this chapter, we will also present a detailed example where we show part of the TLA^+ specification as well as the mapping between theorems, activity steps and ESM transitions.

Chapter 6 will cover the theorems, symptoms, diagnoses and fixes shown in Fig. 3.6. The greyed out boxes represent elements that are still in the early

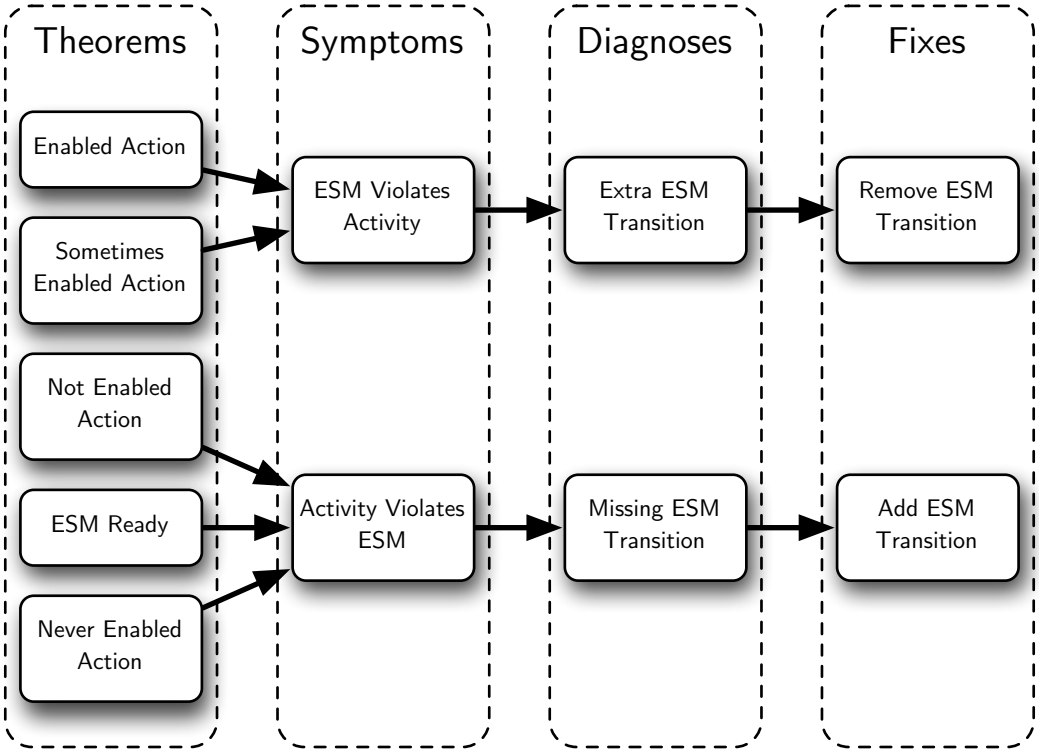


Figure 3.5: Current theorems, symptoms, diagnoses and fixes, part 1

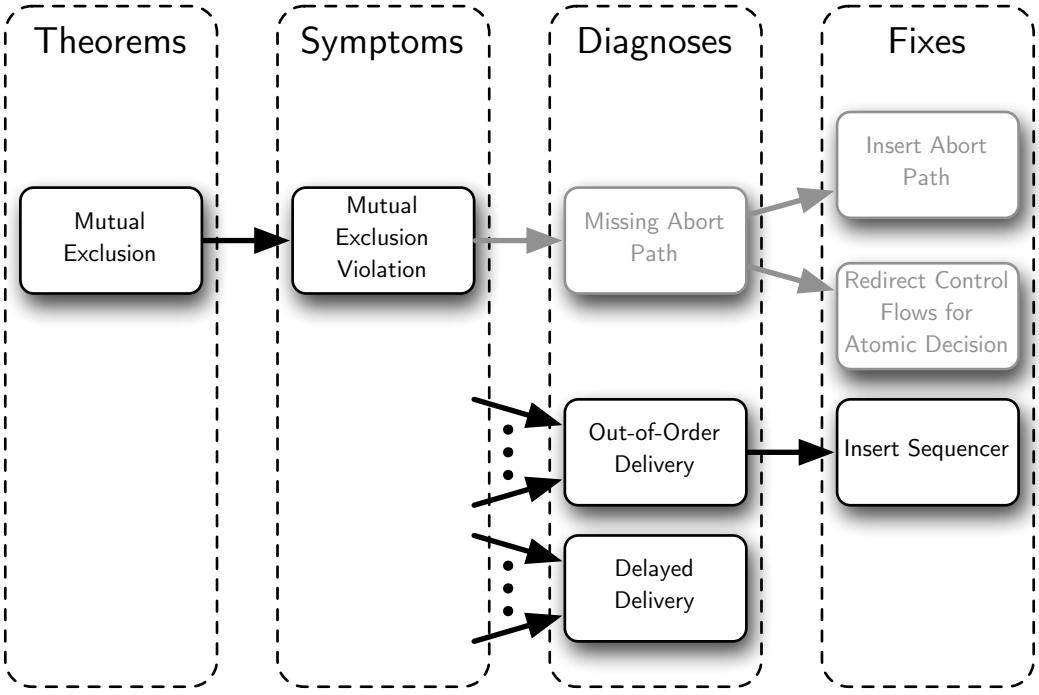


Figure 3.6: Current theorems, symptoms, diagnoses and fixes, part 2

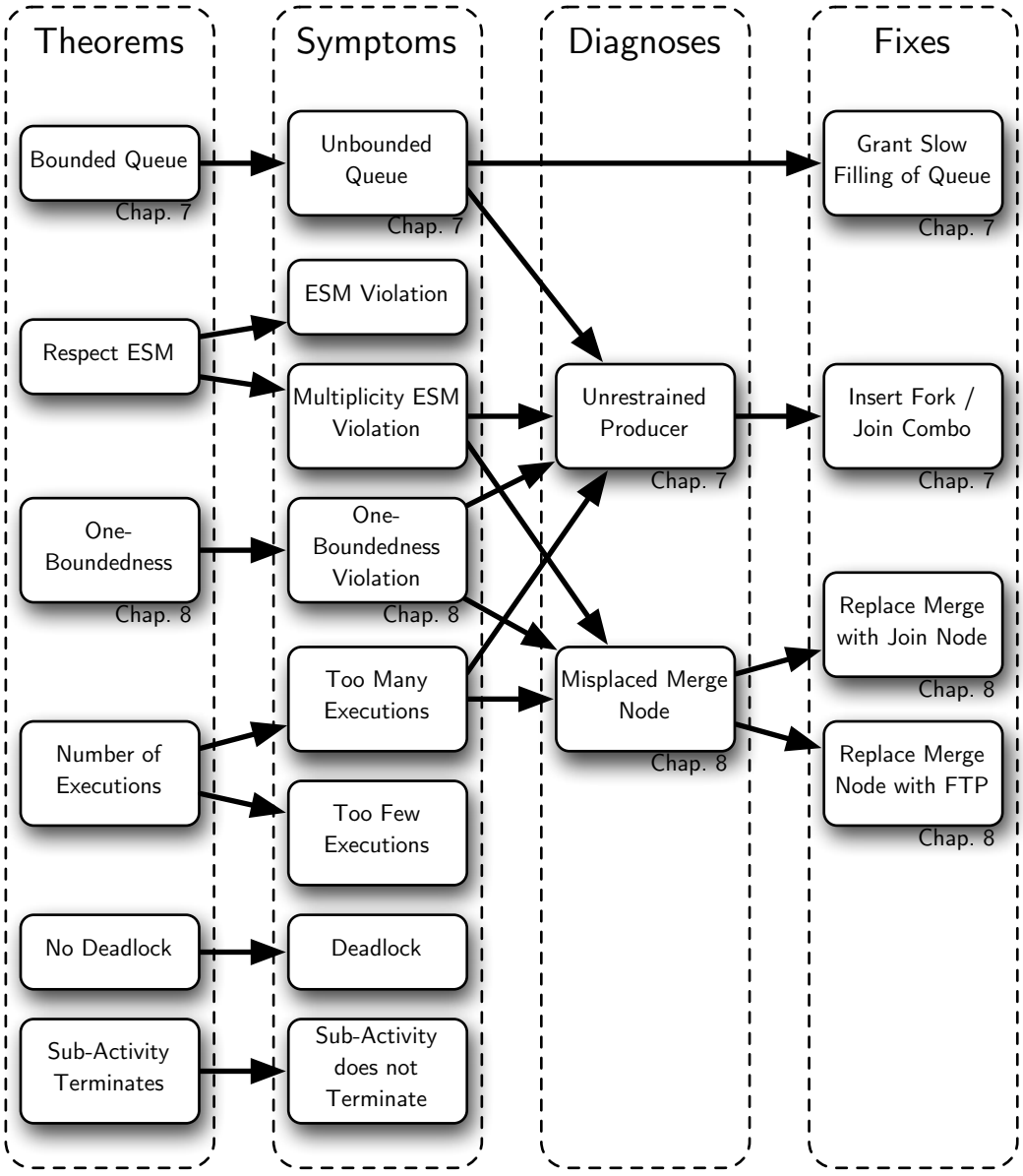


Figure 3.7: Current theorems, symptoms, diagnoses and fixes, part 3

stages of development. The arrows going into *Out-of-Order Delivery* and *Delayed Delivery* signify that every symptom we have so far, attempts to set these diagnoses.

Chapter 7 will discuss the *Bounded Queue* theorem and the symptoms, diagnoses and fixes related to it.

Chapter 8 will discuss *One-Boundedness* theorem and the symptoms, diagnoses and fixes related to it.

The remaining theorems and symptoms will be covered by Chapter 9.

3.3.1 What is Currently Implemented?

To show that the proposed framework works well in practice as well as theory, we have implemented several of the elements in Java, as part of the analyzer tool. Figure 3.8 gives an overview over those that are currently implemented.

3.4 Completeness of Analysis Framework

It is not always clear what the intention of a developer is. Therefore, we may not find all flaws, and the path from symptom to fix may not be clear. As a conceptual aid for discussing how well a flaw can be handled, we identify the following categories.

Cat. A Flaws that we are sure to detect and give a single accurate diagnosis for.

Cat. B Flaws that we are sure to detect, but we can only guarantee that at least one of the proposed diagnoses is the correct one.

Cat. C Flaws that we are sure to detect, but we may not be able to give a diagnosis for. (And even if we do, it may not be correct.) However, we will be able to give a visual trace up to the violating state.

Cat. D - E Same as above, only we do not have all the theorems required to detect the flaw every time.

While category A is obviously the ideal, being able to place a flaw in any category is a contribution, and it helps the user to create specifications that behave as intended. Even if we only have a theorem and a corresponding symptom, it is

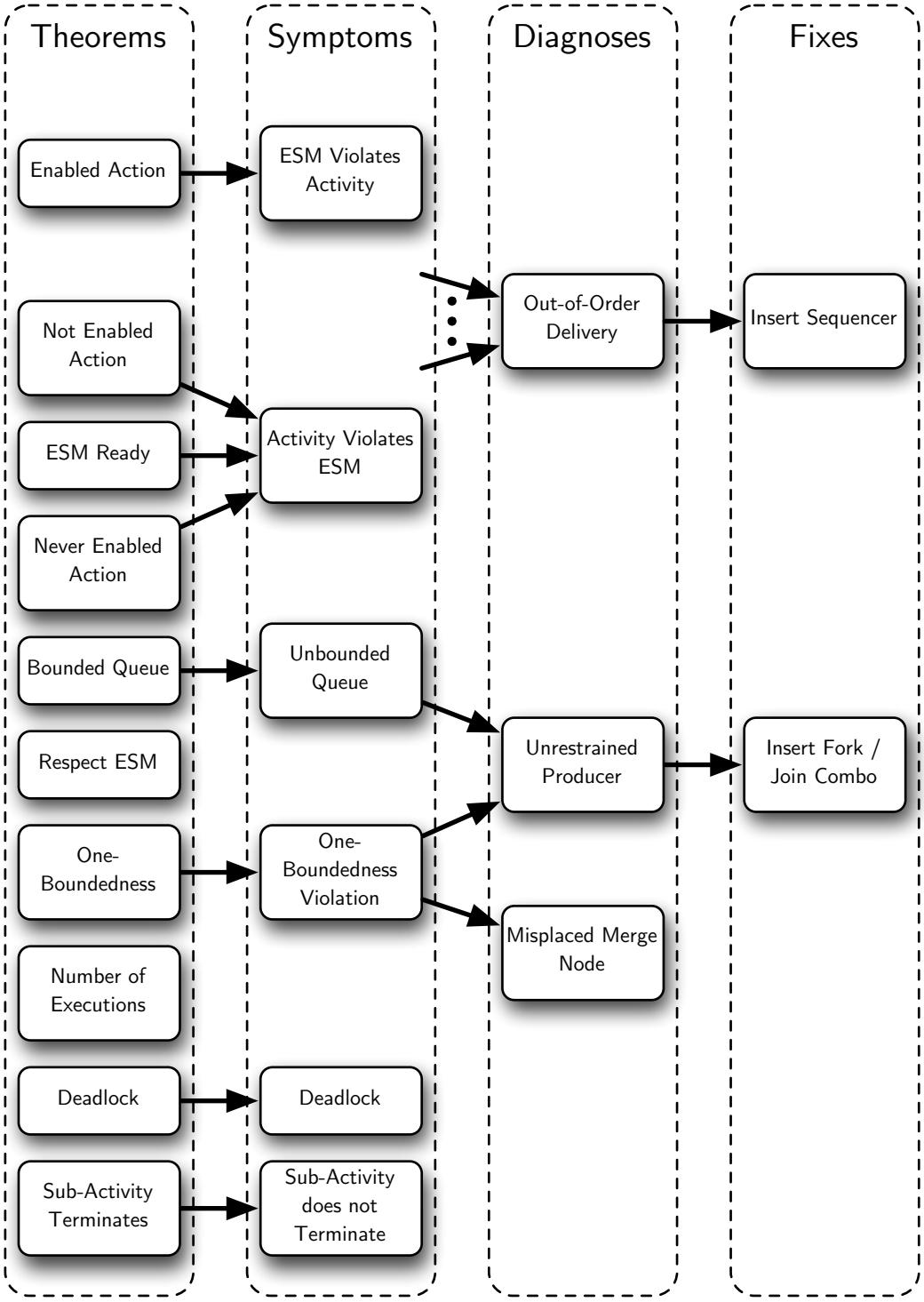


Figure 3.8: Currently implemented theorems, symptoms, diagnoses and fixes

still better than not having any support for that flaw at all. In any case, we can present a trace and that will often be enough for the user to figure out what the flaw is.

This thesis is the start of creating a more complete solution for the analysis of specification behavior. As our experience grows, we can incrementally improve the quality of the tool by adding more elements to the framework. The idea is that we add a theorem and a symptom when we find that a specification is not behaving as intended. Then we add a diagnosis when we see a pattern for what is causing the symptom. Finally we may be able to add an automated fix when we see a pattern for how to fix the flaw. With the addition of new elements to the framework, some flaws will travel further up the categories towards category A. And so, quality will improve.

It is hard, if not impossible to determine exactly what category a flaw should be placed in at a given time. For example, we may think that we can always detect a missing ESM transition, only to later find that it can manifest itself in yet another way, meaning we need yet another theorem (and perhaps symptom) to detect it. Hence a great deal of experience with the tool would be required to at least form an opinion. For example, if no new theorems or diagnoses are added for a symptom for a long time, one could perhaps consider the underlying flaw to be in one of the higher categories.

3.5 Alternative Approach: Graph Analysis

We are currently using TLC to verify all the behavioral properties of our specifications. There are however, other ways of checking the behavior of an Arctis specification. During the process of this thesis, some time was spent investigating how graph analysis might be used as a compliment to TLC.

The idea to use graph analysis came from working on diagnosing an *unrestrained producer*, see Sect. 7.2. Here we ideally want to state a property that a queue or other element cannot be filled with a token twice without being emptied once in-between. This is not possible to express as a standard TLA theorem. We later found that it could be manually expressed as a refinement proof [Lam96], but we did not have the time to investigate whether this could be automated.

The idea we came up with was to transform the UML activity into a semantic graph. But instead of the conventional way of creating a vertex for every state and an edge for the transitions between them, we thought of an alternative solution: What is now a TLA action would be a vertex and a directed edge from one vertex to the other would mean that the downstream action would be enabled once the upstream one had taken place.

The idea was that we could get a smaller graph since we would not have to express every state, just the ones that differed with respect to enabling the following actions. Since no actions have, for example, a counter being a certain value as their precondition, we could ignore counters completely.

Once we had a graph, we could then use existing java graph frameworks like JGraphT [JGr08], JDSL [JDS08] or JUNG [JUN08] for checking some properties. Diagnosing an unrestrained producer would then be as simple of asking for all cycles containing the filling action and see if there are any of these that do not contain the emptying action of the violated queue or node.

However, in order to ensure that an action would be enabled after another one, each action would explicitly have to state the resulting values of any variables that could appear as a precondition for another action. Hence we could no longer have a single action for removing a token from a queue, but one action for removing a token and leaving 4 tokens, another one for leaving 3 tokens, and so on.

At this point we decided to stop investigating this approach any further. It does not seem to be a suitable method for checking general behavioral properties. Graph analysis of a semantic graph, may, however, show some promise for efficiently checking very specific properties that may not be possible to check automatically in TLC. Further work is needed to determine this.

Chapter 4

Implementation

In this chapter, we will look at the analyzer from a technical point of view. The first section discusses the architecture of the analyzer, while the second gives an insight in the changes from the previous formulator tool and how the new transformation of sub-activities works.

4.1 Architecture of the Analyzer

The original formulator focuses on expressing the semantics of the SPACE UML specifications in TLA$^+$. It also generates some theorems that can be violated, but does not offer any assistance in interpreting them.

The analyzer aims to build on this and hide the details of the model checking from the user. In order to do this, we had to substantially expand the original architecture. The analyzer consists of many entities that all are related to one or more other entities. Figure 4.1 shows the architecture of the analyzer including the relationships between the different entities[1]. We will now introduce each of them from an implementation point of view.

The *Controller* is given a reference to the UML activity that is to be transformed, when it is started from Arctis. This reference is simply passed on to the formulator which then returns a reference to the TLA module when done. The controller can then later invoke a TLC process to run TLC on the TLA$^+$ specification.

The *Formulator* entity traverses the nodes and edges of a UML activity and its ESM (if it has one) to slice it into steps. During the traversal, it calls methods in the TLA module, gradually constructing it.

[1]Where there is no cardinality specified, a cardinality of "1" is implied.

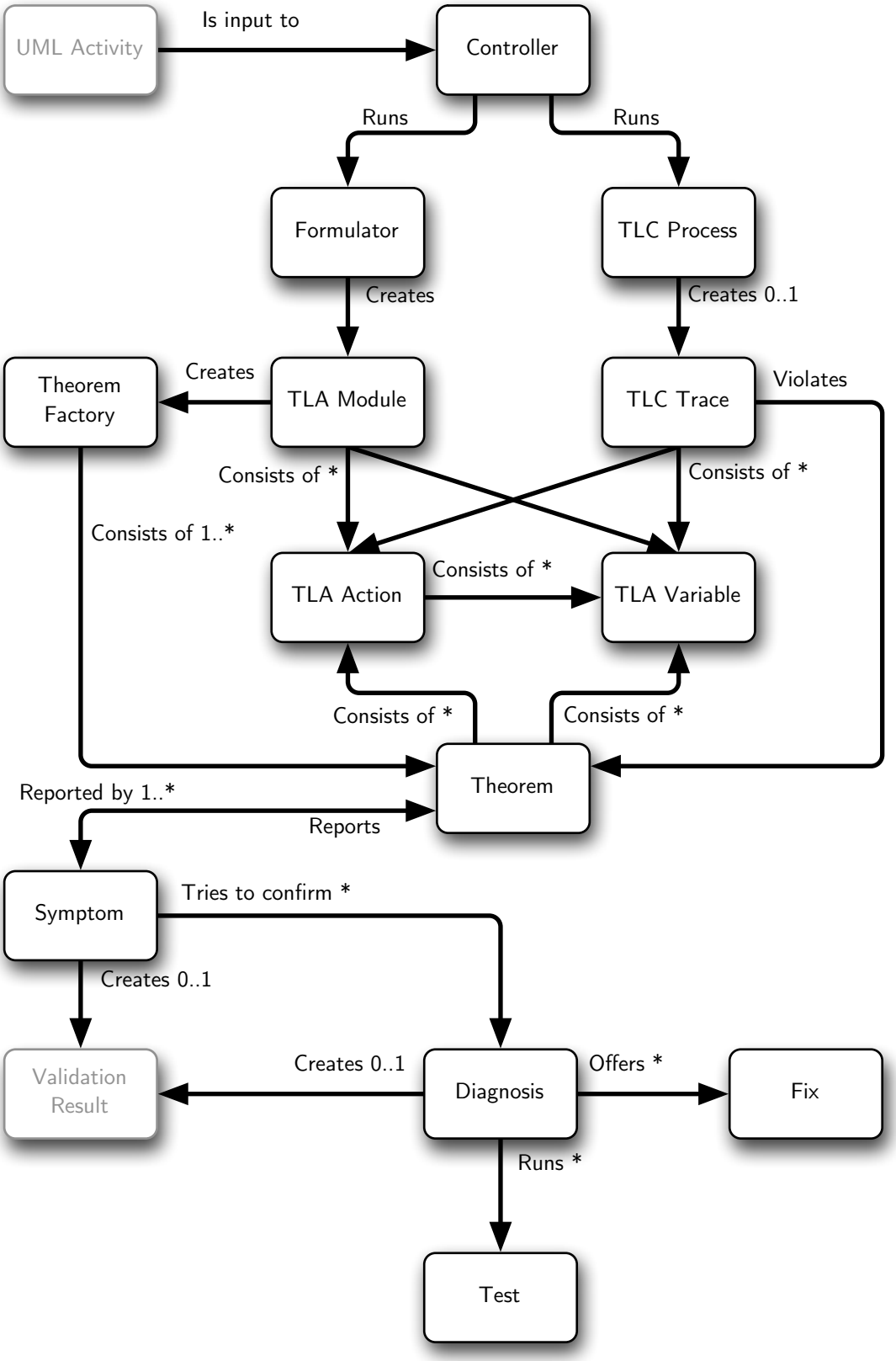


Figure 4.1: Relationships of Analyzer entities

The *TLA Module* contains the data structures and methods necessary to create a TLA$^+$ specification. We also need to use this information after the TLA$^+$ specification has been written to file, and so the TLA module consists of *TLA Variable* and *TLA Action* objects. The TLA variables contain references to the UML elements they were derived from. This allows us to annotate these UML elements later, if a theorem is violated. TLA actions contain the TLA variables that they alter or have as preconditions. TLA actions also contain references to all edges they traverse allowing the TLA actions to be visualized when showing the TLC trace.

The *Theorem Factory* handles the creation of each theorem and keeps a register of them for later lookup. Any new theorem types have to be registered here to be included when the theorems are generated.

The *TLC Process* is basically a wrapper around TLC that allows us to run TLC as a separate process from within the analyzer and parse the output TLC produces. If any of the theorems are violated, the TLC process will create a TLC trace.

A *TLC Trace* consists of TLA actions and TLA variables, just like the TLA module. The difference being that these are contained in lists ordering them according to the states of the trace from TLC. Hence we can get iterate through the trace getting state 1, action 1, state 2, action 2, state 3 etc. A TLC trace object also implements the interface *IAnimationProvider* so that it can be given to the Arctis editor to visualize itself onto the UML activity.

A *Theorem* not only produces a text string to be inserted into the TLA$^+$ specification, but has a checkup method that is run if the theorem is violated. In this method it will determine the correct symptom to report and instantiate this symptom with the violating TLC trace as input.

A *Symptom* will try to confirm one or more diagnoses or simply register itself. Each symptom type is free to choose an algorithm for how it does this. For example, some symptoms will only register themselves if none of the attempted diagnoses are confirmed. A symptom is reported by creating a validation result. This is then registered in Arctis and annotated to a UML element as an error or a warning.

The purpose of a *Diagnosis* is to point out what may be causing a symptom. In other words, a diagnosis represents a typical flaw in a UML activity. The diagnosis checks to see if it is applicable by running one or more tests on the trace and/or the activity. If all tests give the right result, we say that the diagnosis is confirmed. It then registers itself in Arctis by creating a validation result. Note that a validation result is not a part of the analyzer plug-in, but part of the Arctis inspection framework (see Sect. 2.1.2).

Some diagnoses have references to one or more *Fixes*. A fix takes one or more

UML elements as input and alters them (and their surrounding elements). A fix is not run directly from the analyzer, but given as a result to the editor where the user is presented with the option to run it.

A *Test* is a method for answering a specific question about a TLC trace or a UML activity. The tests are basically a toolbox that various diagnoses can reuse.

4.2 Changes from the Original Formulator

The work in this thesis focuses on the interpretation of the model checking results, phase 2 of Fig. 3.2. But we changed some parts that have to do with the transformation from UML to TLA$^+$ as well, like mapping TLA$^+$ variables to UML elements. Also, we changed the way sub-activities are transformed into TLA$^+$. This section describes these changes from the original formulator of [Slå07], to the corresponding software entities of the analyzer.

4.2.1 Improved Mapping Between TLA$^+$ and UML

In the original formulator, TLA$^+$ modules are constructed as a structure of strings, directly derived from the UML building block. This is sufficient to create the file containing the TLA$^+$ specification, so we can run TLC on it. When creating the analyzer, we found that we needed a richer representation where we could go from TLA$^+$ back to the UML model as well.

Hence the TLA module class was enhanced to be able to access data after the TLA$^+$ specification was built from it. To achieve this, we introduced the TLA variables as proper objects with references to the UML element they were derived from. This allows us to annotate these UML elements if a theorem is violated. The TLA actions were also extended with references to all edges they traverse, to be able to visualize a TLA action when showing the TLC trace.

The mapping between UML elements and TLA variables is not one-to-one. For example, all parameter nodes of a sub-activity map to the ESM variable. A join node maps to one variable for each incoming edge it has. Initial and activity final nodes map to the _status variable. Hence we have two structures, mapping in the opposite direction of each other. This enables us to go both ways, if necessary.

4.2.2 Transforming Sub-Activities to TLA$^+$

Perhaps the biggest change from the original formulator is how sub-activities, activities with parameter nodes, are transformed into TLA$^+$. The original formulator can only transform system activities, not sub-activities. Hence one always has to create a surrounding environment activity to be able to test these. We can then detect if the environment activity sends a token into the sub-activity as allowed by its ESM, but the activity itself cannot cope with it. Or we can detect if the sub-activity is about to output tokens when not allowed by its ESM.

This may still allow for some inconsistencies to go undetected. It can often be hard to design an environment activity that can send a token on every pin in every state the ESM allows this, hence properly testing the activity. This is also quite a bit of unnecessary work for the developer.

In order to do the analysis described in Chapter 5, we decided to enabled the analyzer to create an independent TLA$^+$ module (/specification) based on a sub-activity and its ESM alone.

For this we need to express the relationship between steps in the UML activity and transitions of its ESM, in the TLA$^+$ specification. If a step that visits[2] a parameter node can take place when the ESM is in different states, there will be several TLA actions derived from it, one for each state. Yet we still want to be able to reference a step in TLA$^+$ without referencing a particular transition of the ESM (see Chapter 5).

The consequence of this is that there are now two kinds of TLA actions in TLA$^+$ specifications for sub-activities:

- Normal TLA actions that take into account the ESM state if they visit any parameter nodes. These represent things that can happen.
- TLA helper actions that correspond one-to-one with a step of the activity and make no mention of the ESM even though they visit parameter nodes. These are not actually executed, they are not part of the next-state relation of the specification. They are only used for writing theorems using the ENABLED keyword.

To show the difference between normal and helper actions, we will take another look at the *Button* building block that was presented in Sect. 3.2. Figure 4.2 shows two TLA actions that are present in the TLA$^+$ specification derived from the ButtonWrong activity of Fig. 3.1.

[2] A step visits a parameter node if it sends or receives a token through one. See Sect. 5.1 for further explanation.

start is a TLA helper action. Notice how it makes no reference to the ESM variable *ButtonWrong*.

start__initial_active is a normal TLA action that when executed, will change the state of ButtonWrong's ESM from "_initial" to "active". Figure 4.3 illustrates the parts of the ESM and the activity that this TLA action represents.

$start \triangleq$
$\land \_status = \text{“pre\_execution”}$
$\land r0 = 0$
$\land r0' = 1$ Getting ready to receive signal
$\land$ UNCHANGED $\langle \_status, r0\_counter, ButtonWrong \rangle$

$start\_\_initial\_active \triangleq$
$\land r0 = 0$
$\land ButtonWrong = \text{“\_initial”}$
$\land \_status = \text{“pre\_execution”}$
$\land \_status' = \text{“executing”}$ Setting activity to active
$\land r0' = 1$ Getting ready to receive signal
$\land ButtonWrong' = \text{“active”}$ Token arrived through start
$\land$ UNCHANGED $\langle r0\_counter \rangle$

Figure 4.2: Actions derived from the *start* step of the *ButtonWrong* TLA$^+$ specification

(a) ESM transition **(b)** Activity step

Figure 4.3: TLA action: start_initial_active

Similarly, Fig. 4.4 shows the TLA actions derived from the *stop* step, the step where a token enters through the *stop* parameter node and leaves through the *pushed* parameter node. In this simple example, the TLA helper action has no other content than the predicate that the activity must be executing. And the normal TLA action simply adds that a token is only allowed through the *stop* parameter node when the ESM is in state "active".

$stop \triangleq$
$\quad \land \_status = \text{“executing”}$
$\quad \land \text{UNCHANGED } \langle \_status, r0\_counter, r0, ButtonWrong \rangle$

$stop\_active \triangleq$
$\quad \land \_status = \text{“executing”}$
$\quad \land ButtonWrong = \text{“active”}$
$\quad \land \text{UNCHANGED } \langle \_status, r0\_counter, r0, ButtonWrong \rangle$

Figure 4.4: Actions derived from the *stop* step of the *ButtonWrong* TLA$^+$ specification

Figure 4.5 shows how none of the helper actions are part of the Next statement and hence are not executable. Instead it only contains the normal actions shown here and in Fig. 3.3.

$Next \triangleq$
$\quad \lor stop\_active$
$\quad \lor start\_\_initial\_active$
$\quad \lor e1$

Figure 4.5: The Next statement of the *ButtonWrong* TLA$^+$ specification

Chapter 5

ESM Consistency

When we reuse sub-activities, we assume their ESMs to be correct. This is what enables our approach to scale, as we abstract away the internal states and are left with just those that are visibly different to the outside. However, making sure the internals of a sub-activity are consistent with its ESM can be a challenging task, especially for larger ones. Hence we want the analyzer to aid the user in creating building blocks where the activity and its ESM are consistent.

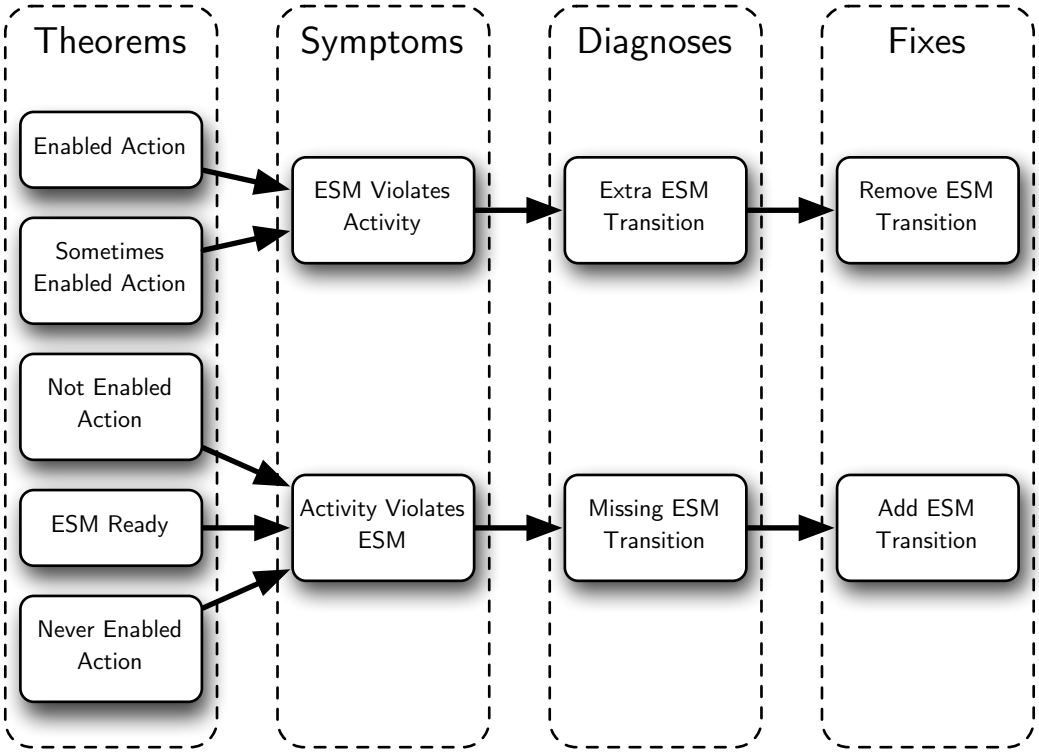


Figure 5.1: Topic: ESM Consistency

Note that the diagnoses and fixes we present in this chapter mainly support a bottom-up approach where the activity is created first and one wishes to check that the ESM matches. The theorems and symptoms work equally well in a top-down approach where the ESM is first specified and an implementing activity is to be created.

We will now take a look at the theorems, symptoms, diagnoses and fixes illustrated in Fig. 5.1. To aid the explanation of the concepts, we will go thoroughly through an example in the end.

5.1 Theorems

There are several things that can cause inconsistency when we design a sub-activity in Arctis. We will first take a look at how we create theorems to detect the different cases and report them as symptoms.

We follow the outline of Fig. 5.1 meaning we first discuss the two theorems that report the *ESM Violates Activity* symptom before we move on to the three that report the *Activity Violates ESM* symptom. At the end, we will also discuss a syntactic check that the analyzer performs.

Before we describe the first type of theorems that we create to check consistency between a sub-activity and its ESM, we must introduce a few new concepts as well as an example to help explain both the concepts and the following theorems.

Figure 5.2 shows a rather artificial example of an activity and its corresponding ESM. Its purpose is only to aid our following explanation. The activity is started through parameter node *S*. At this point it may receive a token through either parameter node *A* or *C*. If the timer has expired, an token entering through parameter node *A* may continue all the way through parameter node *B*, otherwise it will stop at join node *j1*. Note that a token arriving through parameter node *A* could also leave through parameter node *D*. The ESM does not specify this from state *s1*, hence there is an inconsistency between it and the activity. The remaining steps can be worked out by examining the activity diagram, but the interesting ones will be pointed out as we present the theorems.

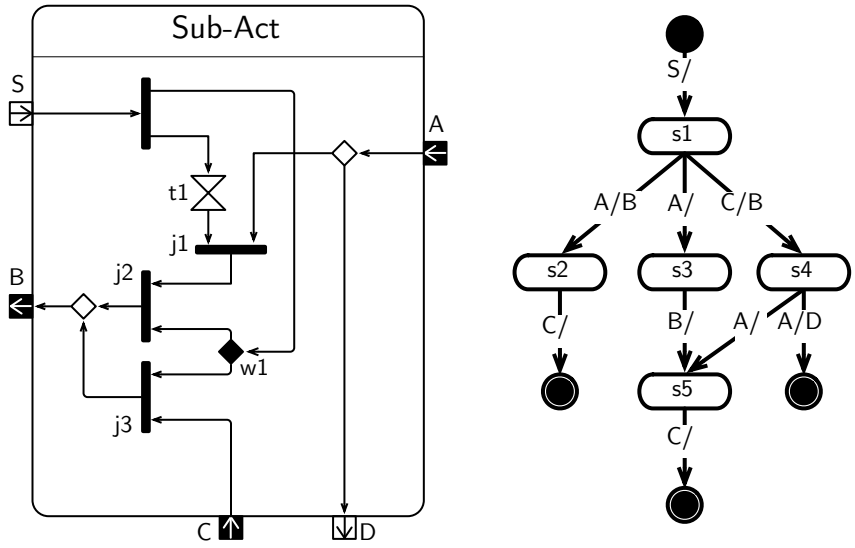
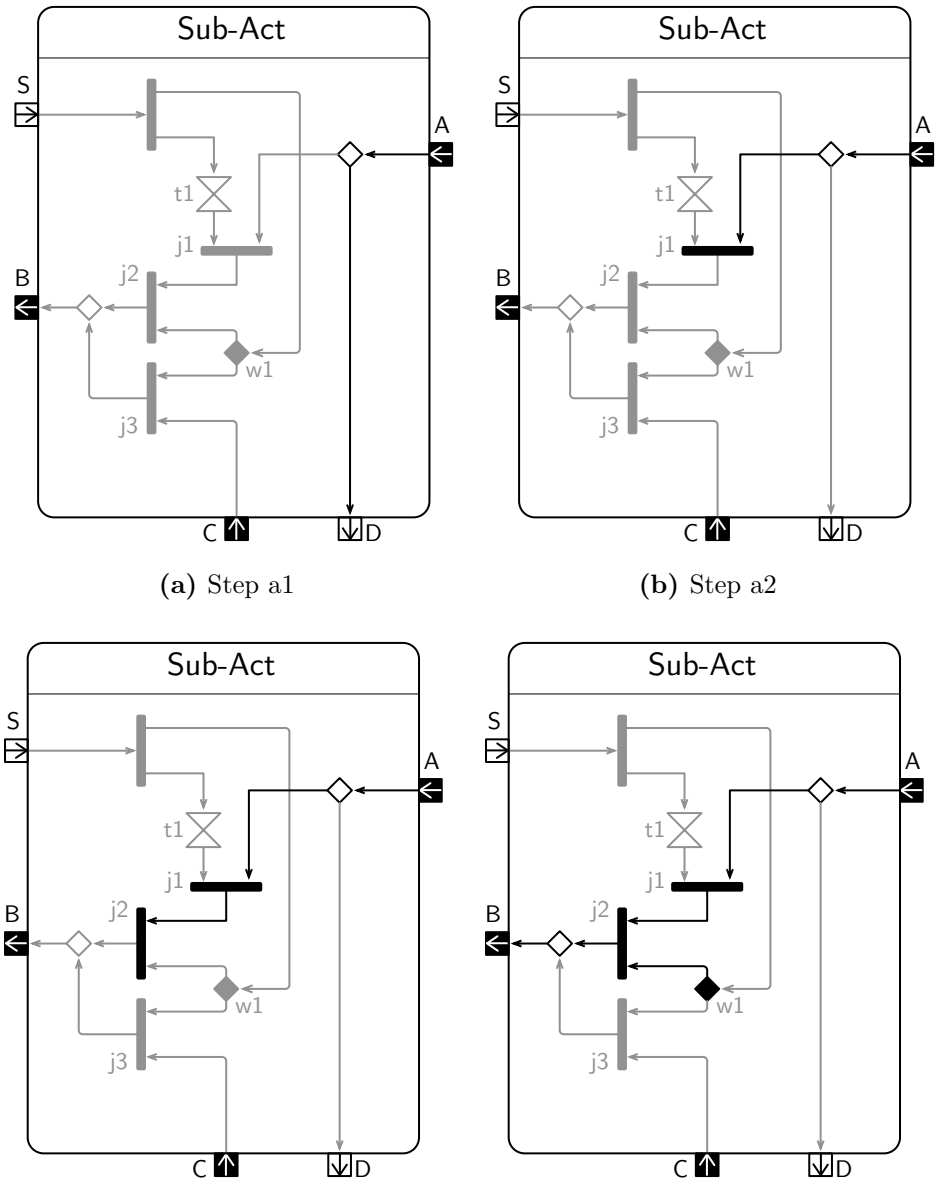


Figure 5.2: ESM consistency example

In Sect. 2.1, a *step* is explained to be the path a token can travel in a single transition of the activity. Figure 5.3 shows the four steps from a token arriving through parameter node *A*. Step *a1* is always enabled (when a token can arrive through parameter node *A*), as the decision node is non-deterministic. Step *a2* is only enabled if both token places in join node *j1* are empty. Step *a3* is enabled if there is a token waiting in the place from the edge arriving from timer *t1* instead, and none of the token places belonging to join node *j2* are filled. Step *a4* needs both *w1* to be filled and the timer *t1* to have expired, to be enabled.

We say that a step or the resulting TLA action(s) *visit(s)* a parameter node if



(a) Step a1 **(b)** Step a2

(c) Step a3 **(d)** Step a4

Figure 5.3: The steps from a token arriving through parameter node *A*

it implies a token traveling through the parameter node.[1] A step or resulting TLA action(s) visiting exactly the one or two parameter nodes specified in an ESM transition, is said to *implement* that ESM transition. From the example, we can see that step *a1* implements all transitions labeled "A/D", in this case *s4→A/D→_initial*. Step *a2* and *a3* implement all transitions labeled "A/" and step *a4* implements those labeled "A/B".

The relationship between theorems, steps, TLA actions, ESM transitions and ESM states is rather complex:

- The theorems we create relate to the source state of an ESM transition, not the transition itself. Each ESM transition only has one source state, but an ESM state can have many outgoing ESM transitions, like states *s1* and *s4* in the example.
- Several ESM transitions may be implemented by a single step. For example, step *a2* implements both *s1→A/→s3* and *s4→A/→s5*. In this case, there will be two normal TLA actions created. One that is enabled in *s1* and one that is enabled in *s4*.
- There may be several steps and hence TLA actions implementing an ESM transition. For example, both step *a2* and *a3* implement *s1→A/→s3* (and *s4→A/→s5*).

In the following sections, a step named *a_b* visits parameter nodes *A* and *B*. Hence it implements any transition labeled "A/B". We will use this as a short hand way of denoting steps that we have not properly described and named.

5.1.1 Enabled Action

The reason for creating *Enabled Action* [2] theorems is that we want to ensure that the activity is ready for any tokens let through by the ESM. More specifically, we create them in order to ensure that (one of) the right step(s) will be executed when a token enters through a certain parameter node. In order to create these theorems we reason as follows from every state of the ESM:

- S is the set of outgoing transitions from the current state.

[1] We use the word *implies* since a parameter node does not have an explicit variable in TLA$^+$, and hence it will not explicitly show up in the TLA action.

[2] The reason we name the theorem type *Enabled Action* (as opposed to *Enabled Step*), is that it refers to a step's TLA$^+$ counterpart, a TLA helper *action*. The same reasoning applies to the other theorem types.

- S_{in} is the subset of S where the trigger of each transition is an incoming parameter node.
- T_A is the intersection of S_{in} and the set of all transitions with incoming trigger A
- For each set T_X, at least one of the steps implementing these transitions must be enabled when the ESM is in the current state.

For our example, we can express the same for state *s1* as follows: When the ESM is in state *s1*, all helper actions implementing a transition leaving from it, with an incoming trigger parameter node, must be enabled. However, if there are several transitions that share the same trigger, only one of them need to be enabled. This results in the *Enabled Action* theorems

$$\begin{aligned} &theorem\_enabled\_a \triangleq \\ &\Box(ESM = \text{“s1”} \Rightarrow (\text{ENABLED } (a2) \vee \text{ENABLED } (a3) \vee \text{ENABLED } (a4))) \end{aligned} \tag{5.1}$$

$$theorem\_enabled\_c \triangleq \Box(ESM = \text{“s1”} \Rightarrow \text{ENABLED } (c\_b)) \tag{5.2}$$

Step c_b is the step that starts from parameter node C, continues through join node *j3* and stops at parameter node B.

If an enabled action theorem is violated, the interpretation is that the ESM allows a token through when none of the TLA actions supposed to deal with that token are enabled. Hence we report an *ESM Violates Activity* symptom. More on this in Sect. 5.2.

5.1.2 Sometimes Enabled Action

Some ESM transitions are spontaneous outputs. This means that they specify that the activity will output a token through a parameter node without this having been triggered by an incoming token in the same step. An example can be seen in Fig. 5.2 where the ESM transition s3→B/→s5 is a spontaneous output, since it is triggered by the internal timer.

We want to create a theorem that detects if the activity complies with the ESM in these cases. One may think the previously discussed *Enabled Action* theorems would be right for the job, but this is not the case. They are used to express that the activity *must* be ready to execute a certain step whenever the ESM is in a state that allows token in through a certain parameter node. But now we want to express that the activity *can* output a token when the ESM is in a state that has spontaneous output transitions originating from it, such as *s3* in the example.

For this we create theorems stating that the activity at least has the possibility of outputting a token as specified by the ESM. However, TLA is not a branching-time temporal logic (see Sect. 2.2), so we cannot state "the TLA action implementing the spontaneous output transition must be enabled in at least one alternative future". We can, however, state that it must "not never" be enabled. This is good enough for our purposes.

Hence, for the example in Fig. 5.2, we create the theorem

$$theorem\_sometimesenabled\_b \triangleq \neg\Box(ESM = \text{“s3”} \Rightarrow \neg\text{ENABLED } (b)) \quad (5.3)$$

Step b is the step that starts from the timer *t1* and continues all the way through *j1* and *j2* to parameter node *B*.

Using that $\Diamond \equiv \neg\Box\neg$ (see Sect. 2.2), we can rewrite the above theorem

$$\begin{aligned} theorem\_sometimesenabled\_b &\triangleq \neg\Box(ESM = \text{“s3”} \Rightarrow \neg\text{ENABLED } (b)) \\ &\triangleq \Diamond\neg(ESM = \text{“s3”} \Rightarrow \neg\text{ENABLED } (b)) \\ &\triangleq \Diamond\neg(\neg ESM = \text{“s3”} \vee \text{ENABLED } (b)) \\ &\triangleq \Diamond(ESM = \text{“s3”} \wedge \text{ENABLED } (b)) \end{aligned} \quad (5.4)$$

We now see that we are really considering an infinitely long timeline of infinitely many runs of the example activity. Hence the "not never" becomes "eventually". If we run the activity infinitely many times, it must eventually be able to output a token when in the state that has a spontaneous output transition.

To implement this theorem, we need to modify the TLA$^+$ specification in such a way that there is a TLA action from the terminated state to the initial state, a *reset* action. This is so that TLC can find a looping behavior that does not satisfy the liveness property expressed in the theorem. Also, we may need to add strong fairness (see Sect. 2.2) to the TLA actions implementing the spontaneous output. This is to make sure they are chosen in some of the executions, if they depend on a decision node.

Should a *Sometimes Enabled Action* theorem be violated, we interpret this as the ESM having promised something to the environment that the activity cannot keep. Hence we report an *ESM Violates Activity* symptom.

5.1.3 Not Enabled Action

In state *s4* of the example, the transitions labeled "A/" and "A/D" are allowed, but not any labeled "A/B". This brings us to the point of the *Not Enabled Action* theorems. We create these under the following conditions:

- The current step has parameter node X as trigger.
- Parameter node X is an incoming parameter node.
- A state is found that does not have any outgoing transitions implemented by the current step, but there are ESM transitions with trigger X.

In this case, the ESM is allowing tokens to be received on parameter node X, but not for the transition implemented by the current step to happen. Hence we must make sure that the current step is not enabled in this state.

For our example, we create the theorem

$$theorem\_notenabled\_a\_b \triangleq \Box(ESM = \text{“s4”} \Rightarrow \neg\text{ENABLED } (a\_b)) \tag{5.5}$$

We also create such theorems for step *a_d* from state *s1*, step *c_b* from state *s2* etc. We do not show these theorems here.

If a *Not Enabled Action* theorem is violated, it means that the activity is ready to take the incoming token and do something other than what is specified in the ESM with it. Hence we report this as an *Activity Violates ESM* symptom.

5.1.4 ESM Ready

If a step implements a transition where the trigger is an outgoing parameter node, another type of theorem is created, an *ESM Ready* theorem. These are created to ensure that whenever a step is enabled, the ESM is in a state that allows a token to be emitted on this parameter node.

The algorithm is quite straightforward: For every step that implements spontaneous output transitions, write a theorem that when this step is enabled, the ESM must be in the source state of one of those transitions.

For step *b* that implements the transition labeled *B/* in the example, we would write an *ESM Ready* theorem

$$theorem\_ESMready\_b \triangleq \Box(\text{ENABLED } (b) \Rightarrow (ESM = \text{“s3”})) \tag{5.6}$$

If there were more than one transition labeled *B/*, we would simply add the source state of each as a disjunct.

$$\begin{aligned} &theorem\_ESMready\_b \triangleq \\ &\Box(\text{ENABLED } (b) \Rightarrow (ESM = \text{“s3”} \vee ESM = \text{“sX”} \vee \ldots)) \end{aligned} \tag{5.7}$$

Should an *ESM Ready* theorem be violated, it means that the activity is about to spontaneously output a token when the ESM is in a state that does not allow this. In other words, the activity violates the ESM and we report that as a symptom.

5.1.5 Never Enabled Action

It is quite likely that syntactically there will exist a step implementing a non-existing ESM transition. A step, or rather the corresponding TLA action(s), that does not implement any ESM transitions is OK, as long as it is never executed. If the step has an incoming trigger parameter node, a *Not Enable Action* theorem will be written to detect if it can be executed. Examples are the steps *s_b_j2* and *s_b_j3* from state *_initial*. Step *s_b_j2* starts from parameter node *S*, continues through *w2* and *j2* before it arrives at *B*. Similarly, step *s_b_j3* takes the route via join node *j3*.

When dealing with a step that does not visit an incoming parameter node, no such theorem is written. This would have been the case for step *b*, the step where the timer expires and outputs a token through parameter node *B*, if there were no transitions labeled "B/" in the ESM. For these cases, we write a *Never Enabled Action* theorem stating that they must never be enabled for execution.

An example, if there were no transitions labeled "B/" in the example ESM, would be:

$$theorem\_neverenabled\_b \triangleq \Box\neg\text{ENABLED } (b) \tag{5.8}$$

If there is a violation of a *Never Enabled Action* theorem, we reports this as an *Activity Violates ESM* symptom. The reason for this is that such a violation represents the activity doing something that is not specified by the ESM. It could be both doing unspecified things with a received token or spontaneously outputting tokens on parameter nodes that are not allowed to do so.

5.1.6 ESM Transitions Without Any Implementing Steps

From a technical point of view, there is another case that belong here with the others. There can be ESM transitions that do not have a single implementing step (a step that visits all parameter nodes that occur in the label of the transition).

All ESM transitions that only have a trigger parameter node will necessarily have at least one step implementing them since the alternative would be to have a parameter node without any connected edges. This would be picked up by the

regular syntactic inspectors. An ESM transition from an incoming trigger parameter node to an outgoing effect parameter node, like *s1→A/B→s2* in Fig. 5.2, can, however, end up without any implementing steps. Imagine an ESM transition *s4→C/D→_initial* in the example from Fig. 5.2. There are simply no edges connection parameter node *C* to *D*.

The analyzer checks for this by mapping all ESM transitions to a set of implementing steps. If there are any ESM transitions with empty sets, the user is notified. This is actually a syntactic check as we do not write any theorems for TLC to check. Hence we should ideally make a syntactic inspector just for this, but so far it is far more practical to use the existing code than to write a very large[3] syntactic inspector just to check this one thing.

5.2 Symptoms

There is a subtle difference between an *ESM Violates Activity* and an *Activity Violates ESM* symptom. They both mean that there is an inconsistency between an activity and its ESM, but they differ in a manner of causality.

An *ESM Violates Activity* symptom is reported when the ESM is making promises to the environment that the activity cannot keep. The flaw may very well lie in the activity, and not the ESM, but it is detected because the ESM claims that the activity will do something which, in fact, it cannot. The advice given by such a symptom is to "Alter (expand) the activity so that it conforms to the ESM or see *Extra ESM Transition* diagnosis."

An *Activity Violates ESM* symptom, on the other hand, is reported when the activity is doing more than the ESM has specified. Here the flaw may also lie in the activity or the ESM, but it is detected because the activity can do something other than what the environment can expect from looking at the ESM. The advice given by this symptom is to "Alter (reduce) the activity so that it conforms to the ESM or see *Missing ESM Transition* diagnosis."

5.3 Diagnoses and Fixes

The diagnoses discussed in this section are not very exciting, as they can always be set if the prerequisite symptom is detected. They do however differ from the symptoms in that while the symptoms do not specify whether an inconsistency is caused by the activity or its ESM, the diagnoses do make such an assumption.

[3]The code necessary to find the steps and TLA actions is around 1000 lines

The following diagnoses assume the ESM to be at fault, and are hence only helpful in a bottom-up approach.

In addition, there are two general diagnoses that every symptom tries to confirm: *Out-of-Order Delivery* and *Delayed Delivery*. These are presented in Chapter 6.

5.3.1 Extra ESM Transition

Every time we have the symptom *ESM Violates Activity*, we can suggest that the cause is that there is one or more extra ESM transitions that should be removed. The user may also solve the issue by changing the internals of the activity, to prevent it from being violated by the ESM. However, we have no specific suggestions as to what the user should do, so this is just advice given in the symptom.

The diagnosis comes with a corresponding fix. The algorithm for removing the extra transition(s) is as follows:

- Find the state of the ESM of the activity as the violation happened. This is found from the last state in the trace. Call this "x".
- Find the parameter node that the token enters the activity through from the TLA Variable of the violated theorem. Call this "y"
- Remove all transitions from the ESM where the source of the transition is "x" and the trigger is "y".

5.3.2 Missing ESM Transition

Whenever we have the symptom *Activity Violates ESM*, we can suggest the diagnosis that there is a missing ESM transition. This could also be solved by changing the internals of the activity, but yet again this is advice for the symptom to give.

Adding an ESM transition cannot be completely automated. We know from the trace what the transition should have as source state, and we can also determine the triggering parameter node from the theorem. If the theorem is a *Not Enabled Action* or *Never Enabled Action* theorem, we can also determine the effect parameter node. But we cannot determine what the target state of the transition should be. It may even be a completely new state not already in the ESM. Hence the user must be given a dialog to choose or input the target state.

5.4 An Example, the Timeliness Observer

We will present an example showing how an activity and its ESM correspond to the TLA$^+$ specification that is generated. We will have a look at the *Timeliness Observer*, Fig. 5.4, discussed in [Slå07] and originally from [HK07]. Since the full example contains hundreds of lines of TLA$^+$ we only show the initial state and four steps/TLA actions.

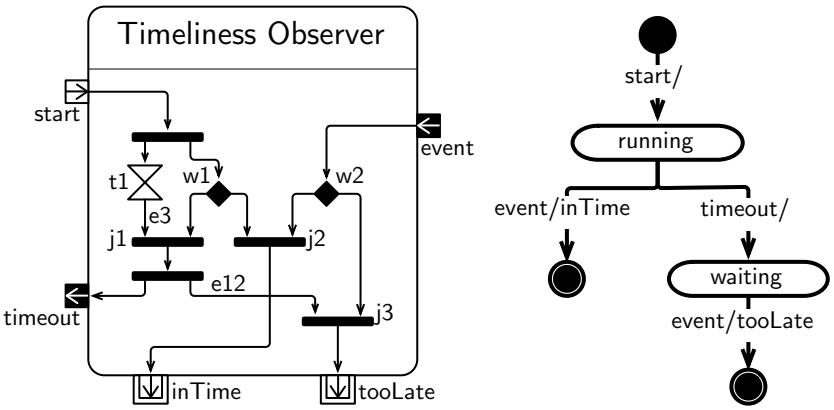


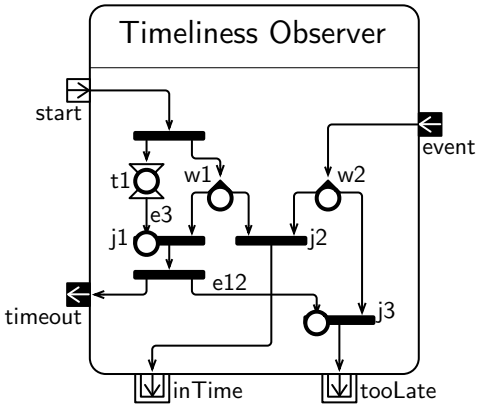
Figure 5.4: The Timeliness Observer. Figure adapted from [Kra07]

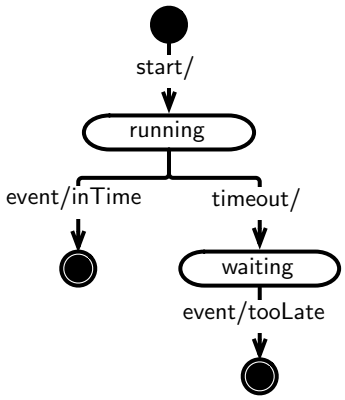
The Timeliness Observer is a building block (specifically, a sub-activity) used to detect whether an event happens within a certain period of time or not. Once started, it enters state *running* where it waits to receive a token on the *event* parameter node to trigger transition *running*→*event/inTime*→*_initial* or to emit a token on the *timeout* parameter node. If it emits a token on *timeout*, it will enter state *waiting*. When the *event* parameter node receives a token in state *waiting*, it will trigger the transition *waiting*→*event/tooLate*→*_initial* and hence output a token on the *tooLate* parameter node instead.

Figure 5.5 shows an annotated version of the Timeliness Observer where the variables have been marked by circles. It also shows the first part of the TLA$^+$ specification which contains all the variables and their initial values.

Step: start

Figure 5.6 shows the step where a token arrives on the *start* parameter node and fills both *t1* and *w1*. This matches the ESM transition labeled *start/* and hence the corresponding TLA action changes the ESM state from *_initial* to *running*.



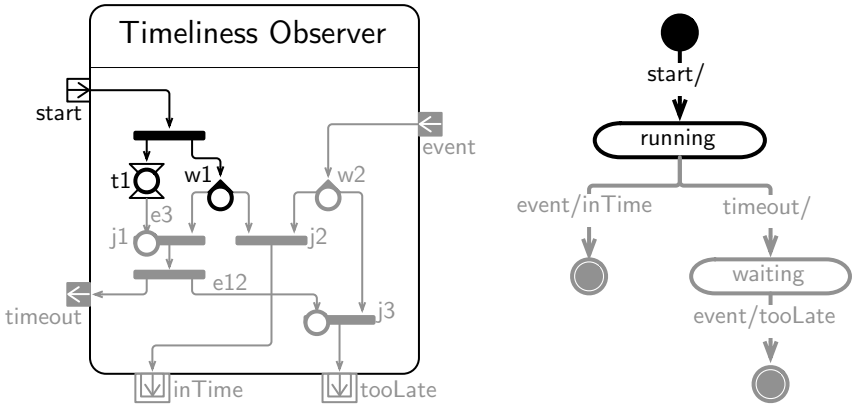



```
1  ──────────── MODULE TimelinessObserver ────────────
2  EXTENDS Naturals
3  VARIABLE w1  The token place of a waiting decision
4  VARIABLE w2  The token place of a waiting decision
5  VARIABLE j3e12  The token place for the join j3 and the edge e12
6  VARIABLE t1  The state of a timer
7  VARIABLE t1_counter
8      This variable counts the number of times a timer expires
9  VARIABLE j1e3  The token place for the join j1 and the edge e3
10 VARIABLE TimelinessObserver  Represents the state of the ESM of an Activity
11 VARIABLE _status
12     Represents the state of the entire activity (active/inactive)
13 vars ≜ ⟨w1, w2, j3e12, t1, t1_counter, j1e3, TimelinessObserver, _status⟩
15 Init ≜
16   ∧ w1 = 0
17   ∧ w2 = 0
18   ∧ j3e12 = 0
19   ∧ t1 = 0
20   ∧ t1_counter = 0
21   ∧ j1e3 = 0
22   ∧ TimelinessObserver = "_initial"
23   ∧ _status = "pre_execution"
```

Figure 5.5: The variables and initial state of the Timeliness Observer

Since this is a starting transition, the value of the *_status* variable is changed from *pre_execution* to *executing*.

We do not show the corresponding TLA helper action as it would be the same as the TLA action shown in the figure, except without references to *TimelinessOb-*



$start\_\_initial\_running \triangleq$
$\land\ t1 = 0$ Checking that incoming token place is free
$\land\ w1 = 0$ Checking that incoming token place is free
$\land$ Checking that at least one other token place is empty
$\quad \lor\ j1e3 = 0$
$\land$ Checking that at least one other token place is empty
$\quad \lor\ w2 = 0$
$\land\ TimelinessObserver =$ "_initial"
$\land\ \_status =$ "pre_execution"
$\land\ w1' = 1$ Adding a token in a waiting decision
$\land\ \_status' =$ "executing" Setting activity to active
$\land\ t1' = 1$ Setting timer
$\land\ TimelinessObserver' =$ "running" Token arrived through start
$\land$ UNCHANGED $\langle j1e3,\ j3e12,\ w2,\ t1\_counter\rangle$

Figure 5.6: Step *start*

server and *_status*. We will not show the helper actions in the following steps either.

There is no one to one mapping between TLA actions and *One-Boundedness* theorems. Still we say that a TLA action *contributes* to such a theorem.

TLA action *start__initial_running* contributes to the following theorem.

$$\begin{aligned} &theorem\_\_status\_TimelinessObserver\_w1 \triangleq \\ &\Box((\_status = \text{"pre\_execution"} \land TimelinessObserver = \text{"\_initial"}) \\ &\Rightarrow (w1 = 0)) \end{aligned} \tag{5.9}$$

It states that whenever Timeliness Observer is in the state *_initial*, the waiting decision *w1* must be empty.

It also contributes to

$$\begin{aligned} &theorem\_\_status\_TimelinessObserver\_t1 \triangleq \\ &\Box((\_status = \text{“pre\_execution”} \wedge TimelinessObserver = \text{“\_initial”}) \\ &\Rightarrow (t1 = 0)) \end{aligned} \tag{5.10}$$

stating that whenever Timeliness Observer is in the state *_initial*, the timer *t1* must be empty.

The analyzer also creates the theorem

$$\begin{aligned} &theorem\_enabled\_start\_\_initial \triangleq \\ &\Box((\_status = \text{“pre\_execution”} \wedge TimelinessObserver = \text{“\_initial”}) \\ &\Rightarrow \text{ENABLED } (start)) \end{aligned} \tag{5.11}$$

stating that when the activity has not yet started, the step (or really, TLA helper action) *start* must be enabled since it implements the starting transition of the ESM.

Step: event

Figure 5.7 shows the step where the Timeliness Observer is in the running state, receives a token on parameter node *event* and neither *j2* nor *j3* are ready to fire. There are actually two TLA actions generated from this step. One which is enabled when TimelinessObserver is in state “running” and one for when it is in state “waiting”. Only the first is shown in Fig. 5.7. In addition, there is the TLA helper action *event* which is also not shown.

The analyzer produces the theorem

$$\begin{aligned} &theorem\_notenabled\_event\_running \triangleq \\ &\Box(TimelinessObserver = \text{“running”} \Rightarrow \neg\text{ENABLED } (event)) \end{aligned} \tag{5.12}$$

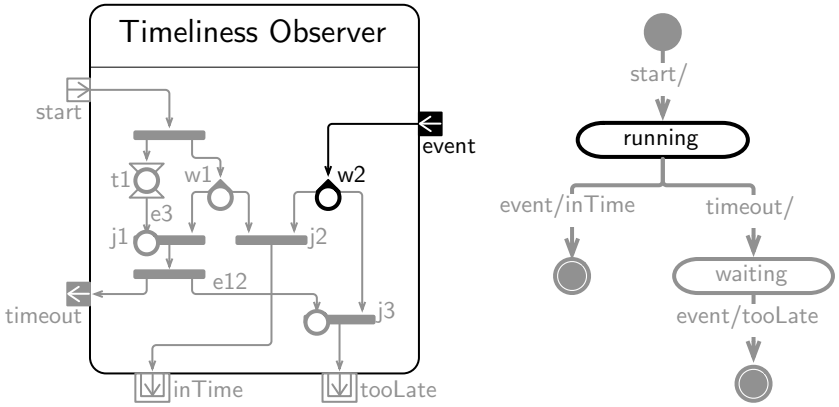
as there is a transition labeled *event/inTime* leaving from state *running*. This transition shares the trigger with the current TLA action. Likewise,

$$\begin{aligned} &theorem\_notenabled\_event\_waiting \triangleq \\ &\Box(TimelinessObserver = \text{“waiting”} \Rightarrow \neg\text{ENABLED } (event)) \end{aligned} \tag{5.13}$$

is created because of the situation in state *waiting*.

This step also contributes to

$$\begin{aligned} &theorem\_\_status\_TimelinessObserver\_w2\_2 \triangleq \\ &\Box((\_status = \text{“executing”} \wedge TimelinessObserver = \text{“running”}) \\ &\Rightarrow (w2 = 0)) \end{aligned} \tag{5.14}$$



$event\_running \triangleq$
$\land \_status =$ "executing"
$\land w2 = 0$ Checking that incoming token place is free
$\land$ Checking that at least one other token place is empty
$\lor w1 = 0$
$\land$ Checking that at least one other token place is empty
$\lor j3e12 = 0$
$\land TimelinessObserver =$ "running"
$\land w2' = 1$ Adding a token in a waiting decision
$\land$ UNCHANGED $\langle w1, \_status, j1e3, j3e12, t1, TimelinessObserver, t1\_counter\rangle$

Figure 5.7: Step *event*

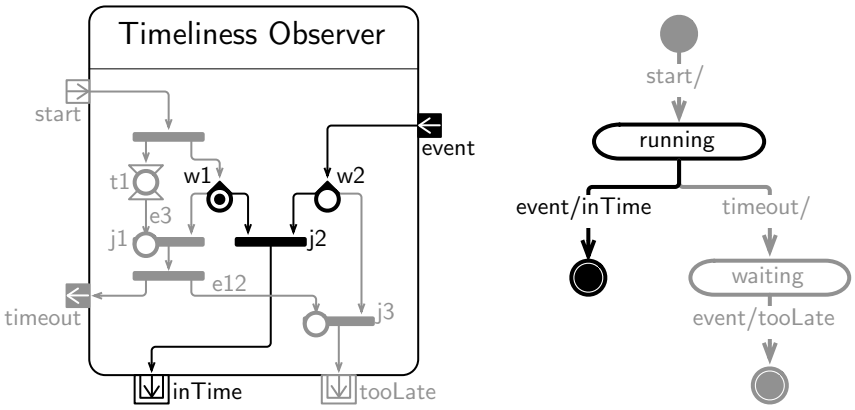
stating that whenever Timeliness Observer is in the state *running*, the waiting decision *w2* must be empty. Likewise it contributes to the following theorem.

$$\begin{aligned} &theorem\_\_status\_TimelinessObserver\_w2 \triangleq \\ &\Box((\_status = \text{"executing"} \land TimelinessObserver = \text{"waiting"}) \\ &\Rightarrow (w2 = 0)) \end{aligned} \tag{5.15}$$

Step: event_j2

Figure 5.8 shows the step where a token is received on the *event* parameter node and a token is in *w1*. This causes *j2* to fire and a token to be output on the parameter node *inTime*. This matches the ESM transition *event/inTime* and hence a TLA action is created that changes the ESM from *running* to *_initial*. Since this is a terminating transition, the value of the *_status* variable is set to *post_execution* as well.

This step also contributes to Theorem 5.14 and Theorem 5.15, stating that when-



$event\_j2\_running\_\_initial \triangleq$
$\wedge\ \_status =$ "executing"
$\wedge\ w1 = 1$ Checking that all other token places are filled
$\wedge\ w2 = 0$ Checking that incoming token place is free
$\wedge\ TimelinessObserver =$ "running"
$\wedge\ w1' = 0$ Firing join
$\wedge\ \_status' =$ "post_execution" Setting activity to inactive
$\wedge\ TimelinessObserver' =$ "_initial"
Token arrived through event and exited through *inTime*
$\wedge$ UNCHANGED $\langle j1e3,\ j3e12,\ t1,\ w2,\ t1\_counter\rangle$

Figure 5.8: Step *event_j2*

ever Timeliness Observer is in the state *running*, the waiting decision *w2* must be empty.

The existence of the *event_j2* step, causes the analyzer to create the theorem

$$\begin{aligned} &theorem\_enabled\_event\_inTime\_running \triangleq \\ &\Box(TimelinessObserver = \text{"running"} \Rightarrow \text{ENABLED } (event\_j2)) \end{aligned} \tag{5.16}$$

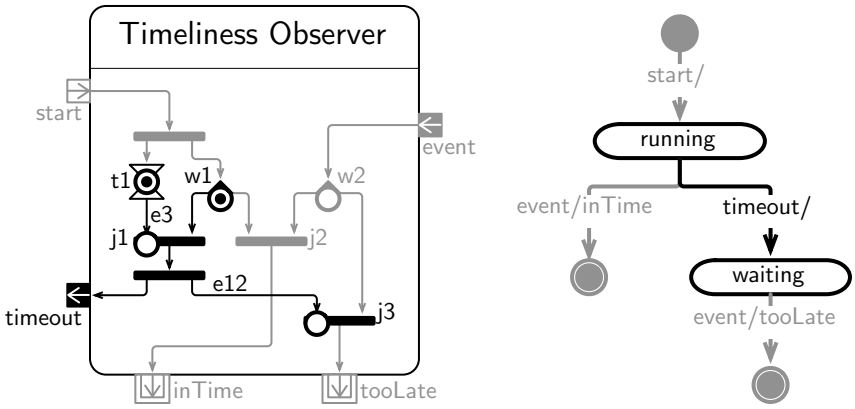
This states the property that whenever the ESM is in state *running*, the TLA helper action *event_j2* must be enabled.

The final theorem created as a result of the step shown in Fig. 5.8 is

$$\begin{aligned} &theorem\_notenabled\_event\_inTime\_waiting \triangleq \\ &\Box(TimelinessObserver = \text{"waiting"} \Rightarrow \neg\text{ENABLED } (event\_j2)) \end{aligned} \tag{5.17}$$

It states that the TLA helper action *event_j2* must not be enabled when the ESM is in state *waiting*. This is because there is another transition with the same trigger as *event_j2* leaving from that state in the ESM. Section 5.1.3 explains this.

Step: e3_j1



$e3\_j1\_running\_waiting \triangleq$
$\land \_status =$ "executing"
$\land t1 = 1$ Timer is set and hence ready to fire
$\land j1e3 = 0$ Checking that incoming token place is free
$\land w1 = 1$ Checking that all other token places are filled
$\land j3e12 = 0$ Checking that incoming token place is free
$\land$ Checking that at least one other token place is empty
$\lor w2 = 0$
$\land TimelinessObserver =$ "running"
$\land w1' = 0$ Firing join
$\land j3e12' = 1$ Adding a token in a normal join token place
$\land t1' = 0$ Firing timer
$\land TimelinessObserver' =$ "waiting" Token arrived through timeout
$\land t1\_counter' =$ IF $t1\_counter < 2$ THEN $t1\_counter + 1$ ELSE 2
Incrementing counter
$\land$ UNCHANGED $\langle \_status, j1e3, w2 \rangle$

Figure 5.9: Step *e3_j1*

Figure 5.9 shows the step where the timer *t1* is ready to output a token and there is already a token in *w1*. This will then cause a token to be output on *timeout* and another token to be placed in *j3e12*.

This step contributes to

$$\begin{aligned} &theorem\_\_status\_t1\_j1e3 \triangleq \\ &\Box((\_status = \text{"executing"} \land t1 = 1) \Rightarrow (j1e3 = 0)) \end{aligned} \tag{5.18}$$

which states that *j1e3* must be empty if *t1* is already filled.

This step also contributes to

$$\begin{array}{l} theorem\_\_status\_t1\_w1\_j3e12 \triangleq \\ \Box((\_status = \text{“executing”} \wedge t1 = 1 \wedge w1 = 1) \Rightarrow (j3e12 = 0)) \end{array} \tag{5.19}$$

which says that *j3e12* must be empty if both *t1* and *w1* are already filled.

The step also makes the analyzer create

$$\begin{array}{l} theorem\_ESMready\_timeout\_running \triangleq \\ \Box(\text{ENABLED } (e3\_j1) \Rightarrow TimelinessObserver = \text{“running”}) \end{array} \tag{5.20}$$

which states that whenever the TLA helper action *e3_j1* is enabled, the ESM must be in state *running*.

5.4.1 Example with a Flaw

Now that we have presented how the activity, its ESM, the TLA$^+$ code and the theorems are related, we will look at how a flaw may be detected by the analyzer.

We modify the *Timeliness Observer* by connecting the edge that should lead to the *timeout* parameter node to a flow final node instead. The modified version is shown in Fig. 5.10 which also shows the trace resulting from running the analyzer.

Since no token ever left through the *timeout* parameter node, the ESM of the activity is still in state “running” when it reaches state 3. This leads to the following violations (refer to Sect. 5.1 for the theorem types):

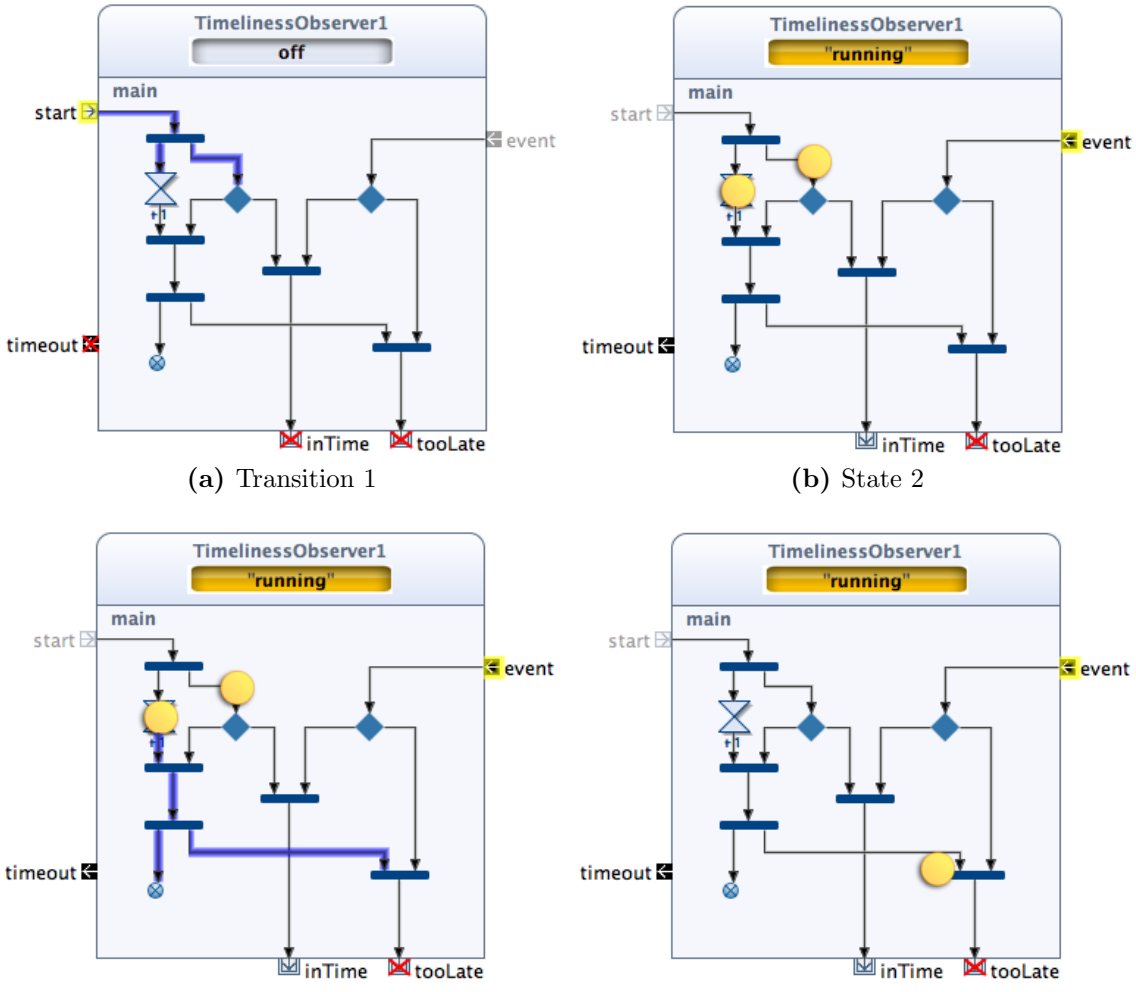
Enabled Action: The step *event_inTime* is not enabled even though the ESM is in state “running”.

Not Enabled Action: The step *event_tooLate* is enabled when the ESM is in state “running”. This state allows transitions trigger by a token through parameter node *event*, but does not allow the transition labeled *event/-tooLate*.

ESM transition without any implementing steps: There are no steps that implement the transition labeled *timeout/*. As said before, this is not a theorem, but a syntactic check that the analyzer makes.

The analyzer does not report all these violations at once. In fact, it will only report one, even if more violations occur in the same state[4]. See Sect. 10.2.1 for a discussion on how we may improve on this in the future.

[4]To produce this list, we manually removed each violated theorem so that the next one would be reported.



(a) Transition 1 **(b)** State 2

(c) Transition 2 **(d)** State 3

Figure 5.10: Trace for the erroneous Timeliness Observer

Chapter 6

Mutual Exclusion and Distributed Behavior

In this chapter, we first present the theorem, symptom, diagnosis and fixes that have to do with the mutual exclusion of two or more UML action elements. We then present two general diagnoses, that are especially relevant to mutual exclusion situations, and a fix for one of them. All framework elements that are handled in this chapter are depicted in Fig. 6.1. We illustrate the how the last two diagnoses are general by drawing arrows with dots between them, representing how all symptoms can call these diagnoses.

6.1 Mutual Exclusion

The user may assert that two UML action elements are to be mutually exclusive. This means that at most one of them may be executed (in a single run of the activity). This is done by applying a stereotype, ≪mutual exclusion≫, that references the mutually exclusive action elements.[1]

We will now describe the theorem that leads to a *Mutual Exclusion Violation* symptom, before we outline a future diagnosis and some future fixes for it.

[1]To avoid having multiple, redundant stereotypes, the ≪mutual exclusion≫ stereotype is simply applied to the activity as a whole.

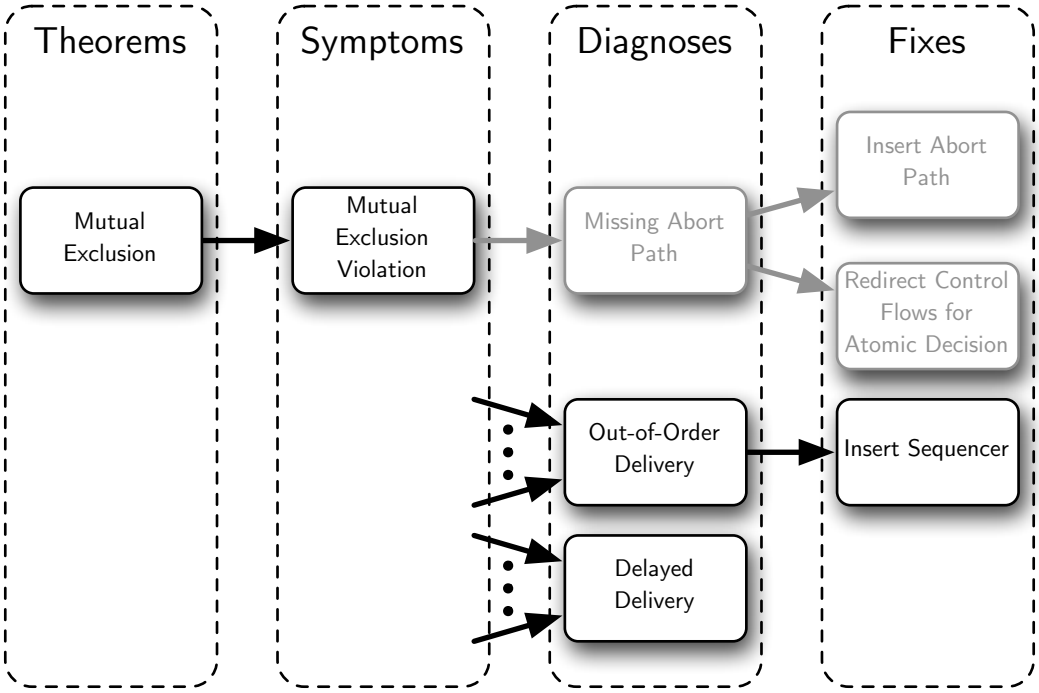


Figure 6.1: Topic: Mutual Exclusion and Distributed Behavior

6.1.1 Theorem and Symptom

When the analyzer transforms an activity into TLA$^+$, it generates counter variables for each UML action element. These are variables that are incremented every time a UML action element like a receive signal action, timer, call operation action or call behavior action[2] are executed. The range of the counter variables is 0 to 2. The values 0 and 1 mean *never* and *once* respectively, whereas 2 means *more than once*.

When the user asserts that two action elements are mutually exclusive, we write a theorem

$$\Box\neg(action1\_counter > 0 \wedge action2\_counter > 0) \tag{6.1}$$

We can also assert the mutual exclusion of three or more actions. We think the mutual exclusion of two actions will be the most common and useful variant, but there is nothing to stop us from making a completely general variant. In this case, we cannot make a single theorem that detects all violations. For example, if three action elements are asserted to be mutually exclusive, we create the theorems

$$\Box(action1\_counter > 0 \Rightarrow (action2\_counter = 0 \wedge action3\_counter = 0)) \tag{6.2}$$

[2]We plan to add a counter for every pin of a call behavior action, to be able to express assertions in more detail.

$$\Box(action2\_counter > 0 \Rightarrow (action1\_counter = 0 \wedge action3\_counter = 0)) \quad (6.3)$$

$$\Box(action3\_counter > 0 \Rightarrow (action1\_counter = 0 \wedge action2\_counter = 0)) \quad (6.4)$$

If a mutual exclusion violation is detected it means that the user failed to make sure that at most one of the action elements received a token. There are two ways of making sure of this.

- A choice is made at some point in the activity, and a token is only sent towards one of the mutually exclusive action elements.[3]
- The activity sends tokens towards both action elements, but as soon as one of them executes, the path of the token heading for the other action element is blocked. This is really a case of *mixed initiative* [Flo03] where we need to avoid that both initiatives are taken. Mixed initiative is also known as *conflicting initiative* [BH93] or *non-local choice* [BAL97] and an example is discussed in [KSH07].

If any of the above theorems are violated, we report a *Mutual Exclusion Violation* symptom. Such a symptom gives advice that either a decision must be made upstream of the elements, or there must be "abort paths" (explained shortly) between them.

6.1.2 Diagnosis

We have an idea for a diagnosis for the mixed initiative scenario. In this case, there must be a path leading from somewhere in the step(s) that traverse(s) action 1 to somewhere upstream of action 2. This is the only way to stop a token that is heading towards action 2. An example is shown in Fig. 6.2. We can find whether such an abort path exists by syntactic graph traversal. We will present how this can be done in Sect. 7.2.

Of course, for the mutual exclusion to work both ways, there should be a similar path from action 2 to action 1. Hence, in this example, the diagnosis would be confirmed, as the execution of action 2 does not prevent the execution of action 1 afterwards.

In this example, there is a shallow building block, *Switch*. The ESM of a switch is shown in Fig. 6.3. It is used to make a decision depending on whether some token has arrived on the *flip* pin or not.

[3]By sending a token *towards* an element, we mean that it will eventually reach it, not necessarily in the same step.

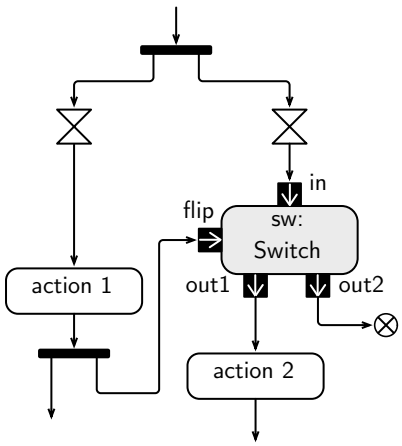


Figure 6.2: Example of feedback from action 1 to action 2

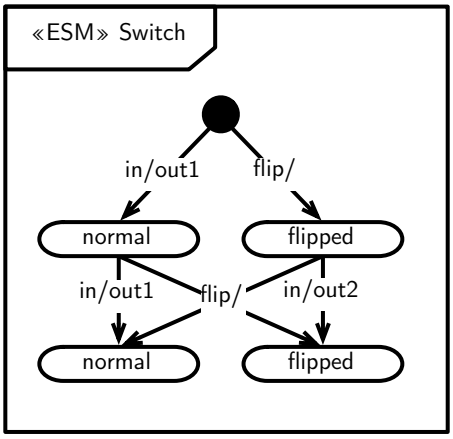


Figure 6.3: ESM of Switch

6.1.3 Fixes

There are two fixes we can think of, depending on whether the mutually exclusive actions are in the same partition or not.

If both action elements are located in the same partition, we can easily insert a switch before each action, as well as edges between them, to create the kind of structure shown in Fig. 6.4.

If the two actions are located in different partitions, there is no way to abort one just as the other has taken place. We therefore need to lead the control flow from both partitions to a common partition where a decision can be made in a single atomic step. Then the winning control flow can return to its originating partition to execute the action there. An example of this is shown in Fig. 6.5. The ESM

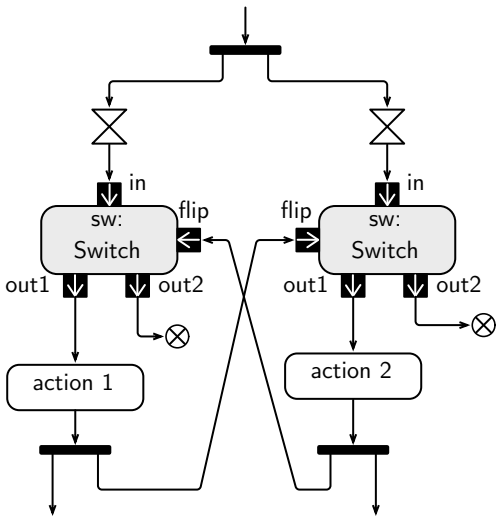


Figure 6.4: Example of mutual feedback between action 1 and 2

of a *First* shallow building block, is shown in Fig. 6.6.

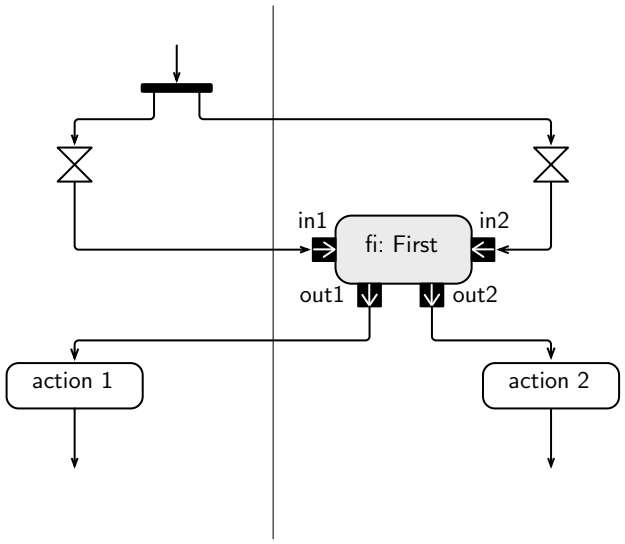


Figure 6.5: Example of *First* block to ensure mutual exclusion

However, as the example in the following section will show, what exactly should be considered *upstream* of an action element is not so easy to determine. If the mutually exclusive action elements are not normal action elements, but pins of a call behavior action, one needs to analyze the ESM to see if an "abort path" is provided by the specification. The automatic insertion of a *First* block, or even a full blown *MIPS/MISS* (see Sect. 6.2.2 and [KSH07]), is currently out of reach.

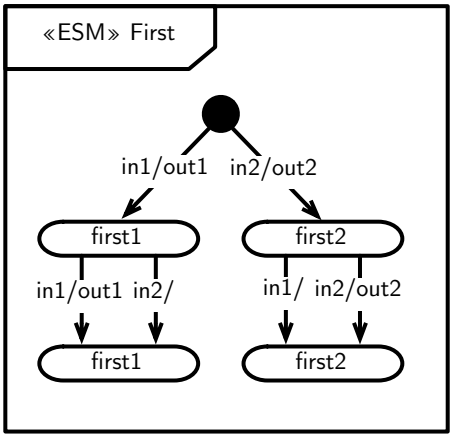


Figure 6.6: ESM of First

Both the diagnosis and the fixes therefore need some further work.

6.2 General Diagnoses

This section introduces two general diagnoses, *Out-of-Order Delivery* and *Delayed Delivery*. We say they are general because they are applicable from every symptom we have found so far, although they may be more relevant for some than others. We will also introduce a fix that can be applied to one of the diagnoses.

To demonstrate the diagnoses, we introduce an example building block, *Button Game*, as shown in Fig. 6.7. It represents a part of a game where a local and a remote player compete to be the first one to press their button, once started.[4] It is somewhat artificial in that it does not ensure that the buttons are activated at the same time, but it serves its purpose of demonstrating the diagnoses, without being too large and complicated. The *Button* building block, is introduced in Sect. 3.2.

6.2.1 Diagnosis: Out-of-Order Delivery

When we analyze the first version of *Button Game*, we find a *Respect ESM* theorem to be violated, causing an *ESM Violation* symptom to be reported (see Sect. 9.1.1). A token can arrive on pin *stop* of call behavior action *remote*, before

[4]The example can also be interpreted as a simplified version of the *Hotel Wakeup* activity from Fig. 2.4.

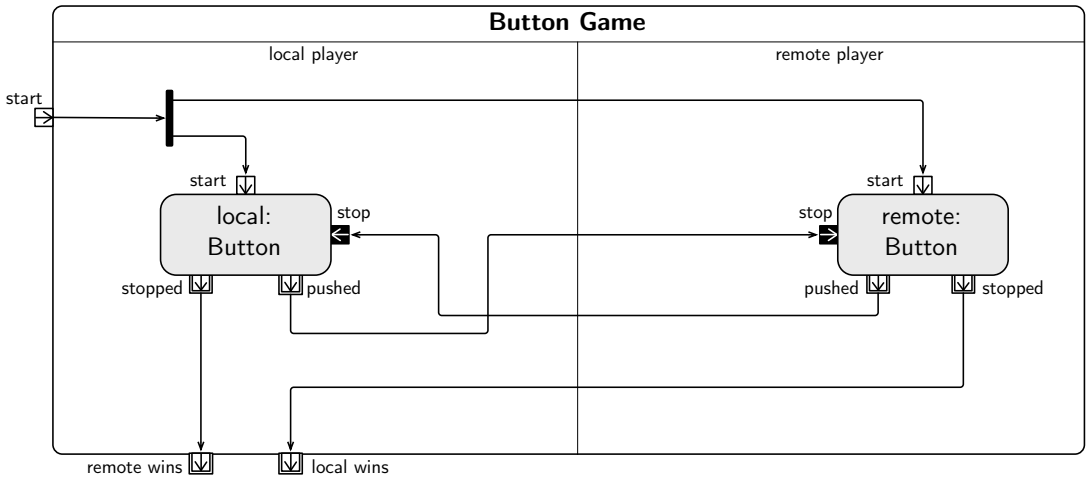


Figure 6.7: Button Game 1

it has even been started. In other words, the token sent to start the *remote* button is still in the queue between the *local player* and the *remote player* partition. Figure 6.8 illustrates the violating state of the activity.

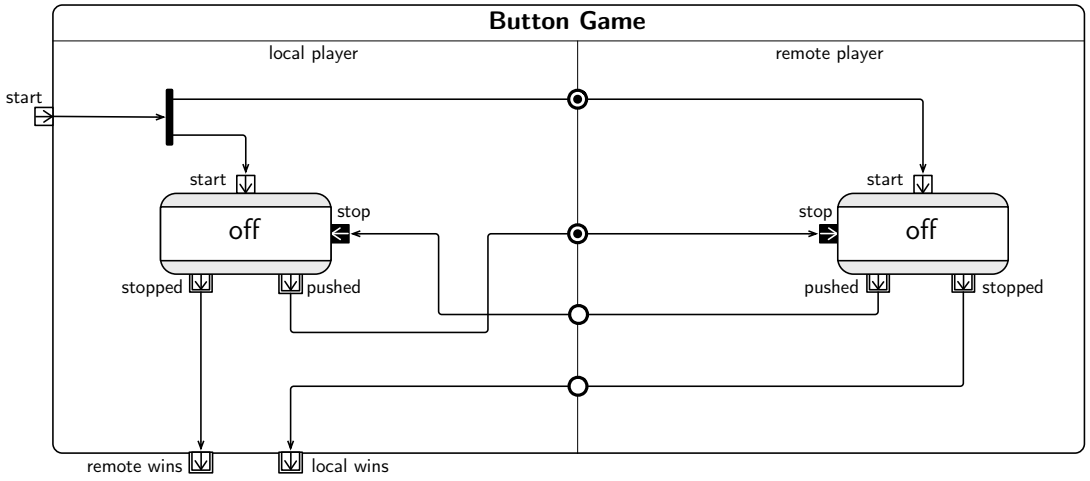


Figure 6.8: Button Game 1, violating state

There are no specific diagnoses for an *ESM Violation* symptom, as it is so general. One of the diagnoses we still check if can be confirmed is an *Out-of-Order Delivery* diagnosis. This diagnosis is confirmed if the following holds true:

- Somewhere in the trace leading up to the violation, two queues entering the

same partition carry tokens

- and the one that was filled last, is emptied first
- or the violating state is the first state where the second queue is filled.

We illustrate these two cases in Fig. 6.9. In the first case, one token has overtaken another. This is possible since our semantics does not guarantee channels between components to deliver signals in FIFO order. In the second case, the violation happens just as the second queue is filled. Hence the violation is caused by a state in which the second queue can now emit a token. This also means that the token would overtake the one in the first queue.

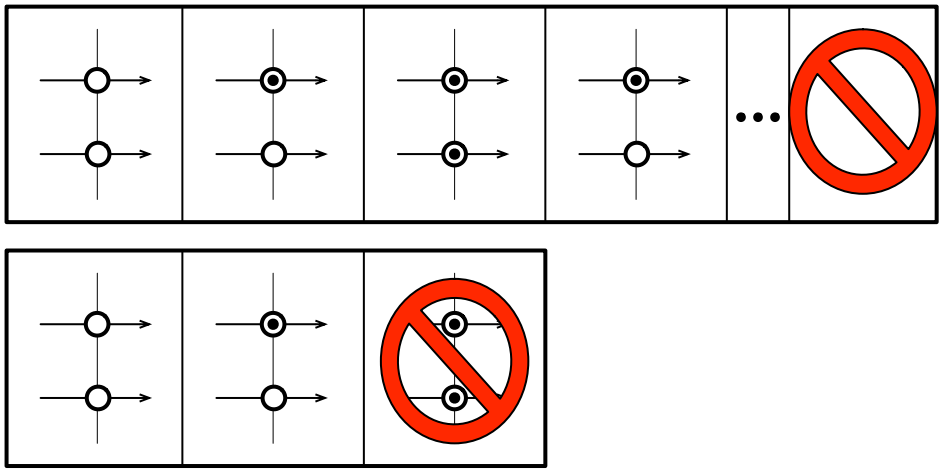


Figure 6.9: Traces that confirm an *Out-of-Order Delivery* diagnosis

We attempt to confirm this diagnosis because many problems could potentially be due to the fact that the developer is wrongfully assuming FIFO channels between components. If two tokens have been delivered "out-of-order", we explicitly notify the developer of this so he or she may pay special attention to whether this was the cause of the symptom.

Fix: Insert Sequencer

Whenever we report an *Out-of-Order Delivery* diagnosis, we also give the user the option of automatically inserting a sequencer. We apply this fix to our example and get the activity shown in Fig. 6.10. The nodes that make up the sequencer are circled.

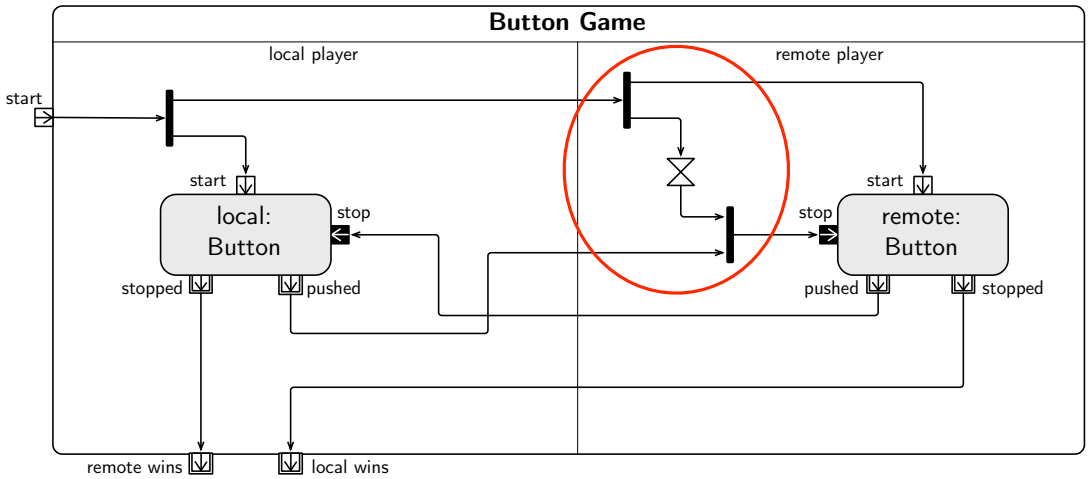


Figure 6.10: Button Game 2

For this fix to be automatic, we assume that the queue that is first filled, is the one that is intended to deliver its token to the receiving partition first. We call this the primary queue.

The fix inserts three nodes in the activity, and all nodes are inserted in the target partition of the queues. A fork node is inserted in the edge that represents the primary queue and a join node in the edge that represents the secondary queue. These are connected via a timer. The timer is there to ensure that both tokens are not consumed simultaneously, should the secondary queue be emptied first.

6.2.2 Diagnosis: Delayed Delivery

We analyze the second version of the *Button Game* and find that there is yet another problem: Both buttons can be pressed before the token sent to stop the other has arrived, as shown in Fig. 6.11. This also results in an *ESM Violation* symptom, as both buttons could receive a token on pin *stop* when they have already terminated. See Fig. 3.1(a) for the ESM of the *Button* building block.

We could also imagine that we had specified that not both buttons may be pushed by applying the mutual exclusion stereotype to the *pushed* pin of both buttons. In this case, we would rather get a *Mutual Exclusion Violation* symptom. But as we have yet to complete any diagnoses specific for this symptom, we will continue in the same manner from both symptoms.[5]

[5]In fact, in this example both theorems would be violated by the same state, so we do not know which theorem TLC would first report as violated.

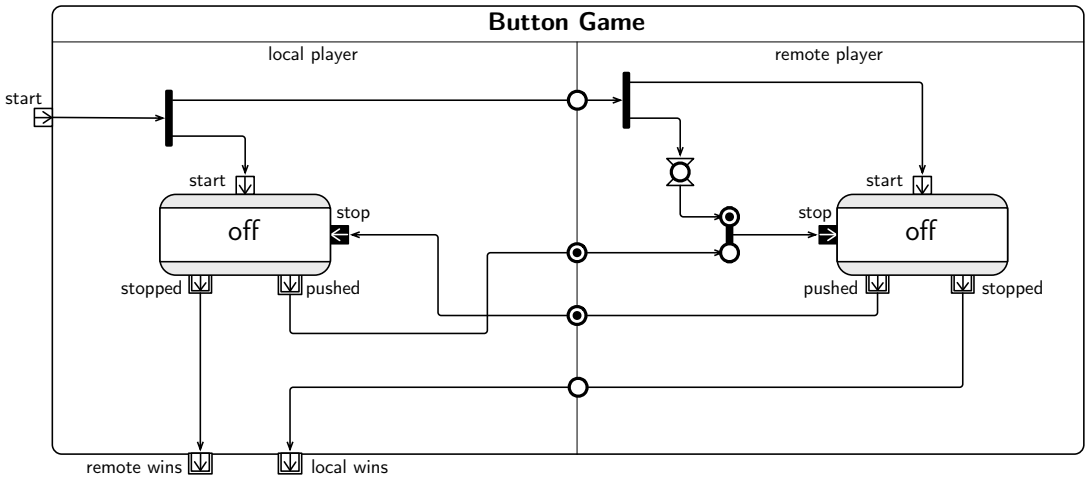


Figure 6.11: Button Game 2, violating state

The analyzer will still try to confirm an *Out-of-Order Delivery* diagnosis and in fact, that diagnosis will be confirmed. But having already applied the fix for this, we look to the next diagnosis reported, a *Delayed Delivery* diagnosis. This diagnosis is confirmed under the following condition:

- There is a state in the trace where
  - a queue from partition A to B has at least one token
  - and a queue from B to A has at least one token.

Said in another way, there are tokens crossing in each direction between two partitions. This kind of error trace is typical of a mixed initiative situation. While there are no corresponding automated fixes to this diagnosis, it does inform the user of this textually and suggests to have a look at the *MIPS* and *MISS* building blocks that are designed to solve mixed initiative situations.

Following its advice, we choose to insert a *MISS* (*Mixed Initiative Secondary Starter*) block between the two partitions. This gives the activity shown in Fig. 6.12. The secondary side now has to take into account that it can be overruled even after taking initiative. In our example, being overruled is no different to loosing normally, except that when overruled, we do not have to stop the *local* button.

The ESM of the *MISS* building block is shown in Fig. 2.13. It gives priority to the non-starting side, as explained in [KSH07].

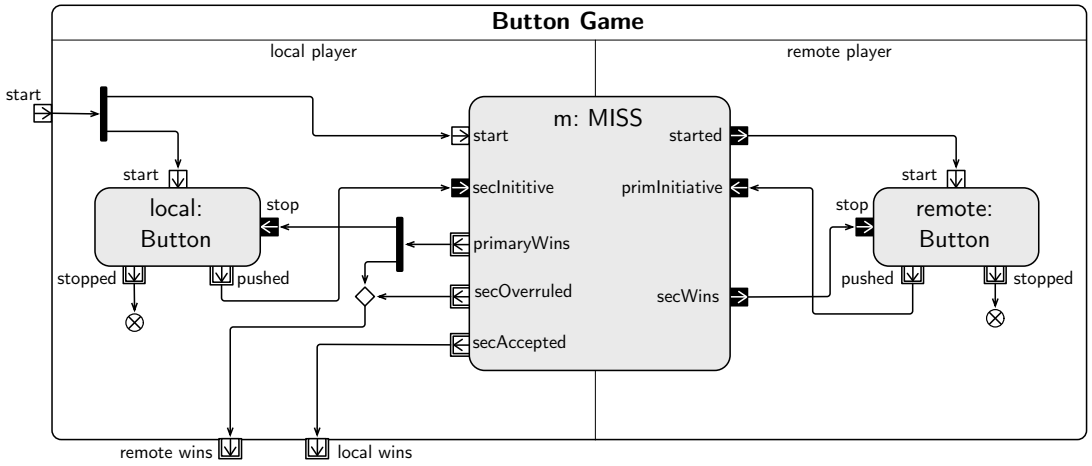


Figure 6.12: Button Game 3

Chapter 7

Bounded Queues

The reactive systems we specify are typically distributed. The services provided within such systems are spread across several components that communicate asynchronously through buffered channels. Since real channels have finite buffers, we want to detect if they could potentially be overflown by tokens.

A channel, or queue, is represented by an edge crossing the border between two partitions of an activity. The model checker can only work in a finite state space, so we provide a parameter that the user can set, for how many tokens a queue can contain before we assume it to be unbounded. By default, this parameter is set to five tokens. We will use the default value in the following discussion.

We will illustrate how we may find an unrestrained producer by going through the scenario shown in Fig. 7.1.

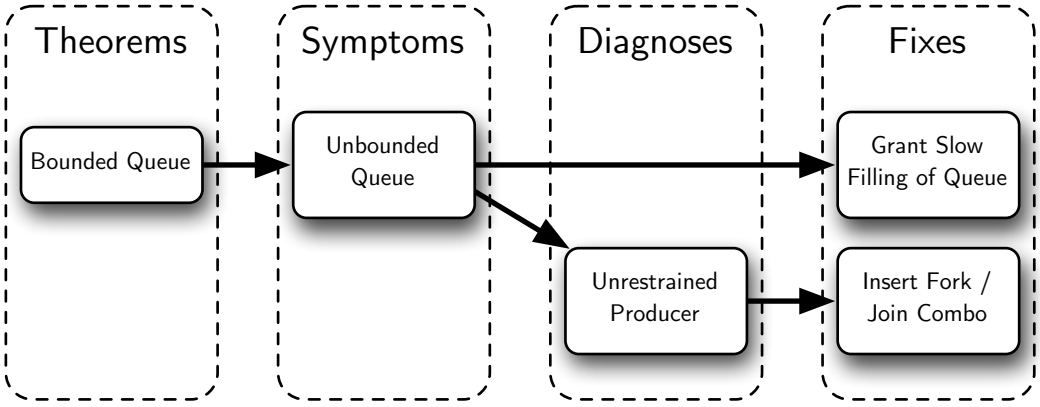


Figure 7.1: Scenario: Unbounded Queue

To aid our explanations, we revisit the example introduced in Chapter 1, the Location Tracker. This building block tracks the position of a mobile client of

some sort, and notifies both the serves side and the client side when the client is close to a target location.

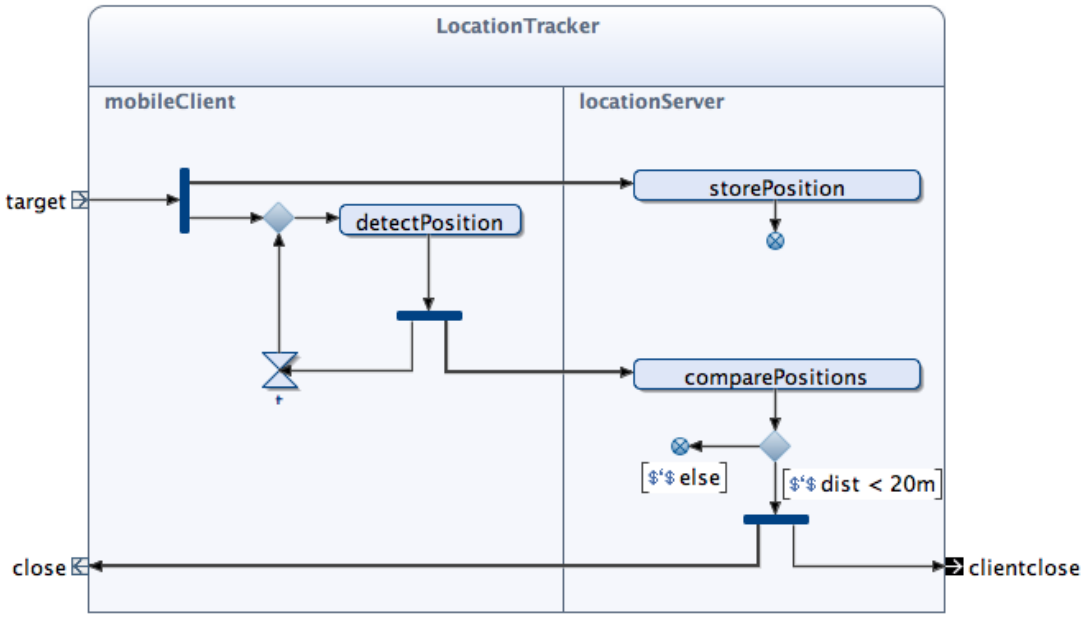


Figure 7.2: The Location Tracker

7.1 Theorem and Symptom: (Un)bounded Queue

To detect an unbounded queue, we write a theorem stating that the number of tokens in the queue should be at most five (or the value the user has set). We call this a *Bounded Queue* theorem.

$$theorem\_queue\_name\_K\_bounded \triangleq \Box(queue\_name \leq 5) \tag{7.1}$$

For the example, one theorem would be written for each of the three edges that cross the partition border.

This theorem only has one symptom and hence there is no additional processing required to determine that the symptom is an *Unbounded Queue*. This symptom explains that the queue is likely to be unbounded, but that the user may change the number of tokens allowed in the queue, to see if this makes a difference.

7.2 Diagnosis: Unrestrained Producer

A queue (or other element) can only be overflown by tokens if they are being duplicated somewhere. The only activity node with the ability to duplicate a token, is the fork node. Hence we start by looking for fork nodes that could have caused caused other elements to be overflown. We only need to consider fork nodes that occur in the trace as any other fork nodes will never have received a token and hence could not have duplicated it. For the fork node to perform the duplication more than once, at least one of the duplicates must return to the fork node. This can only happen if there is a cycle in the activity that contains the fork node. We call such a cycle a *producer*. Yet, if this cycle also contains the overflown element, a token will leave the element just as often as a token will arrive. In other words, the producer will be restrained by this element and the element will not be overflown. Hence the cause can only be a cycle that contains a fork node, but not the overflown element. We call such a cycle an *unrestrained producer*.

Figure 7.3 shows a producer cycle containing the fork node *f1*. This producer is restrained with respect to queues *q1* and *q2*, but unrestrained with respect to *q3*. Remember that we only try to confirm a diagnosis after we have a symptom. Hence, if queue *q3* is overflown, the figure would show an unrestrained producer, otherwise not.

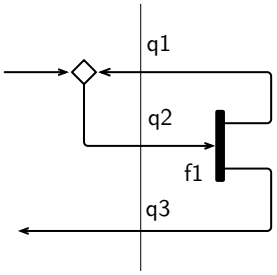


Figure 7.3: A Producer Cycle

We will now present three ways of detecting an unrestrained producer, as well as a pragmatic compromise between two of them.

7.2.1 Alternative 1: Syntactic Cycle Detection

We can detect an unrestrained producer by the algorithm shown in Fig. 7.4. We find all fork nodes that have been visited in the trace. Then we find any paths from these fork nodes back to themselves, but remove any cycles that also contain

the overflown element. We are then left with a list of unrestrained producers.

The *FindCycle()* method is a recursive implementation of depth-first search. To get a demonstrator ready in time, the current implementation of the *FindCycle()* method, is quite simple. That implementation does not take into consideration that to continue from a join node, you must be able to arrive via all incoming edges. Also, for sub-activities the implemented method follows all outgoing edges (from all output pins), but ideally it should only follow those that are reachable via direct or indirect transitions from the input pin.

```
   Data: Trace from TLC
   Data: UML Activity
   Data: Violated element (node or queue)
   // List of fork nodes
 1 List⟨ForkNode⟩ ForkNodesInTrace;
 2 foreach TLA Action in trace do
 3 |   foreach Visited edge do
 4 |   |   if Target of edge is a fork node then
 5 |   |   |   Add to ForkNodesInTrace;
 6 |   |   end
 7 |   end
 8 end
   // A map from each fork node to a list of cycles containing
      that fork node
 9 Map⟨ForkNode, List⟨List⟨Edge⟩⟩⟩ CycleMap;
10 foreach Fork node in ForkNodesInTrace do
   |   // Recursive method that finds all paths from the fork node
   |      back to itself and add them to CycleMap
11 |   FindCycle(ForkNode target, List⟨Edge⟩ pathSoFar);
12 end
13 foreach Entry in CycleMap do
14 |   for All cycles in map entry do
15 |   |   Remove all cycles containing the violated element from CycleMap;
16 |   end
17 |   if Fork node has no remaining cycles then
18 |   |   remove entry from CycleMap;
19 |   end
20 end
21 return CycleMap
```

Figure 7.4: Detecting an unrestrained producer, syntactic algorithm

For our example, the algorithm would start with the two first fork nodes, as these are the only ones that will have received a token when the violation occurs (see Fig. 1.8). For the first fork node (the one closest to the starting parameter node), it will follow all outgoing edges, but never be able to arrive back to the starting fork node. When the algorithm examines the second fork node, it will find a path through the timer, the merge node and the call operation action *detectPosition*, that leads back to this very fork node. It then checks if the second queue from the top is contained in this path. Since it is not, it reports a possible unrestrained producer.

Unfortunately, it is not possible to set an accurate diagnosis for an unrestrained producer purely syntactically. There could be sub-activities that we have to traverse in order to get back to the fork node. Imagine, for example, that the cycle containing the fork node goes through the Timeliness Observer (see Fig. 5.4) via the transition "event/inTime". There is no way to syntactically check whether or not this transition will be enabled. That depends on the state of the timeliness observer something which cannot be determined by syntactic analysis. Hence our syntactic algorithm takes a "worst case" approach in that it assumes that tokens will be able to travel along all the edges. This means that the diagnosis will sometimes be confirmed for a producer that is actually restrained.

Even though syntactic analysis can give too many unrestrained producer diagnoses, we claim that it is still useful. In the case of there being only one unrestrained producer in addition to another producer, restrained due to a join node or some time dependent choice, we will give a warning on both of them. Through common sense and with the help of the visual trace, the user should be able to decide which one is the real problem.

7.2.2 Alternative 2: TLA Refinement

To set a truly accurate diagnosis, we need to analyze the behavior. We need a theorem stating that "If we reach a state where we can fill the queue, we can only reach such a state again via a state where we just emptied the queue". Unfortunately this is not that straightforward as most other theorems. This theorem is a small program in itself as pointed out in [Lam94, Sect. 6.3]. Hence we would have to use a TLA refinement proof [Lam96] stating that the specification derived from the UML activity implies an abstract specification where you never reach two filling states in a row without going through an emptying state. We did not have time to explore further if creating the refinement mapping (see Sect. 2.2.1) necessary for this proof can be done automatically, but this is something we aim to investigate later.

7.2.3 Alternative 3: Analyzing the Trace

There is another way to analyze the behavior of the specification. We still have the trace from TLC caused by the violation of the *Bounded Queue* theorem. This trace contains a subset of the activity behavior, the one leading up to the violating state. We can use the trace to look for cycles as well. This avoids the problems of a syntactic analysis in that we are only given the steps that actually took place. An abstraction of the algorithm is as follows:

- Add every fork node you encounter when going through the trace to a list.
- If you encounter the violated element, empty the list.
- If you encounter a fork node that is already in the list, you have found an unrestrained producer.

```
Data: Trace from TLC
Data: Violated element (node or queue)
// List of fork nodes
List⟨ForkNode⟩ ForkNodesInTrace;
// List of fork nodes that are part of unrestrained producers
List⟨ForkNode⟩ UnrestrainedProducerForkNodesInTrace;
foreach TLA Action in trace do
    foreach Visited edge do
        if Target of edge is a fork node then
            if ForkNodesInTrace.contains(Target) then
                Add target to UnrestrainedProducerForkNodesInTrace;
            else
                Add target to ForkNodesInTrace;
            end
        else if Target of edge is the violated element then
            Empty ForkNodesInTrace;
        end
    end
end
return UnrestrainedProducerForkNodesInTrace
```

Figure 7.5: Detecting an unrestrained producer, trace analysis algorithm

Figure 7.5 shows the new algorithm written out in detail. It does not give the complete unrestrained producers, since it only knows what fork nodes are part

of one, not what edges form the cycles. However, the current fix (see Sect. 7.3) does not need to know the full cycle to work, just the fact that the violation is caused by an unrestrained producer.

Applying this algorithm to our example, we add the first fork node (right after the starting parameter node), then the second one (right after *detectPosition*) and then encounter the second fork node again. We can hence be certain to have found an unrestrained producer.

The problem with this algorithm is that it is not guaranteed to find an unrestrained producer that violates the one-boundedness of a node.[1] The one-boundedness property is violated before the token actually fills an already filled node, see Sect. 8.1. We could imagine an example such as the *Location Tracker* where all the queues between *mobileClient* and *locationServer* are replaced by timers. The property would be violated in state 2 (see Fig. 1.8), where the fork node has still only received a token once. The reason for this is that *One-Boundedness* theorems are of the type

$$\Box(n1 = 1 \Rightarrow n2 = 0) \tag{7.2}$$

where *n1* and *n2* are inner place nodes. In contrast, *Bounded Queue* theorems are of the type

$$\Box(queue\_size \leq 5) \tag{7.3}$$

meaning that a *One-Boundedness* theorem is violated one state earlier than a *Bounded Queue* theorem.

7.2.4 A Pragmatic Compromise

The best practical solution currently, is to choose either alternative 1 or 3 depending on whether the symptom is an unbounded queue or something else. The syntactic algorithm finds all unrestrained producers, but may give too many confirmed diagnoses. The algorithm analyzing the trace is accurate, but only works if the symptom is an unbounded queue.

7.3 Fix: Insert Fork / Join Combo

We have implemented an automated fix that can be applied if the diagnosis of an unrestrained producer is confirmed. The fix takes as input the triggering node

[1] Figure 3.7 shows that the *Unrestrained Producer* diagnosis can be called from a *One-Boundedness Violation* symptom.

of the action that puts the token in the violated element (the queue) and the violated element itself.

The fix does the following, as illustrated in Fig. 7.6:

- Inserts a fork node right after the violated element.
- Inserts a join node right after the triggering node of the action that puts a token in the violated element.
- Inserts an edge from the fork node to the join node.

The fix seems to work nicely on the example, but there is a catch: By applying the fix, we have created a new producer that contains the previously violated element. If there are more stateful elements downstream of it, they will now be overflown. Hence a better idea would be to apply the fix to the last stateful element downstream of the producer. But this is hard to implement as the last violation will not show up before all the elements between the producer and the last element has had the fix applied. We could then suggest to remove the formerly applied fixes and supply the user with an automatic way of doing that. This is not a very elegant solution, though.

Another solution might be to let TLC run with the "-continue" keyword (see Sect. 2.2.2) to discover all violations at once. This could work as the specifications are made in such a way that actions overflowing nodes are not enabled[2] and hence the behavior past the point of the first violation should be correct. However, this is not currently the case for queues. We could add the precondition that a queue must contain less than five tokens to put a token into it. Then the specification would give meaningful behavior even after the state where the first violation is detected. We will revisit this topic in more detail in Sect. 10.2.1.

In our example, the queue from *locationServer* to *mobileClient* can be overflown once the fix has been applied. So we see that although the fix solved the first problem, there is a better way: We could connect the edge that is currently connected to a flow final node, to the join node created by the fix. This would mean that there is no feedback if the client is close to the target, but this is OK. In fact, *Location Tracker 2* will not be consistent with its ESM (Fig. 1.4) unless we do this: The current solution would allow the activity to send more than one token through parameter node *clientclose*, before it is terminated.

This illustrates that even though we can supply automated fixes in some cases, the user still has to think about whether applying the fix is consistent with the

[2]For example, the TLA action in Fig. 5.6 requires both *t1* and *w1* to be empty.

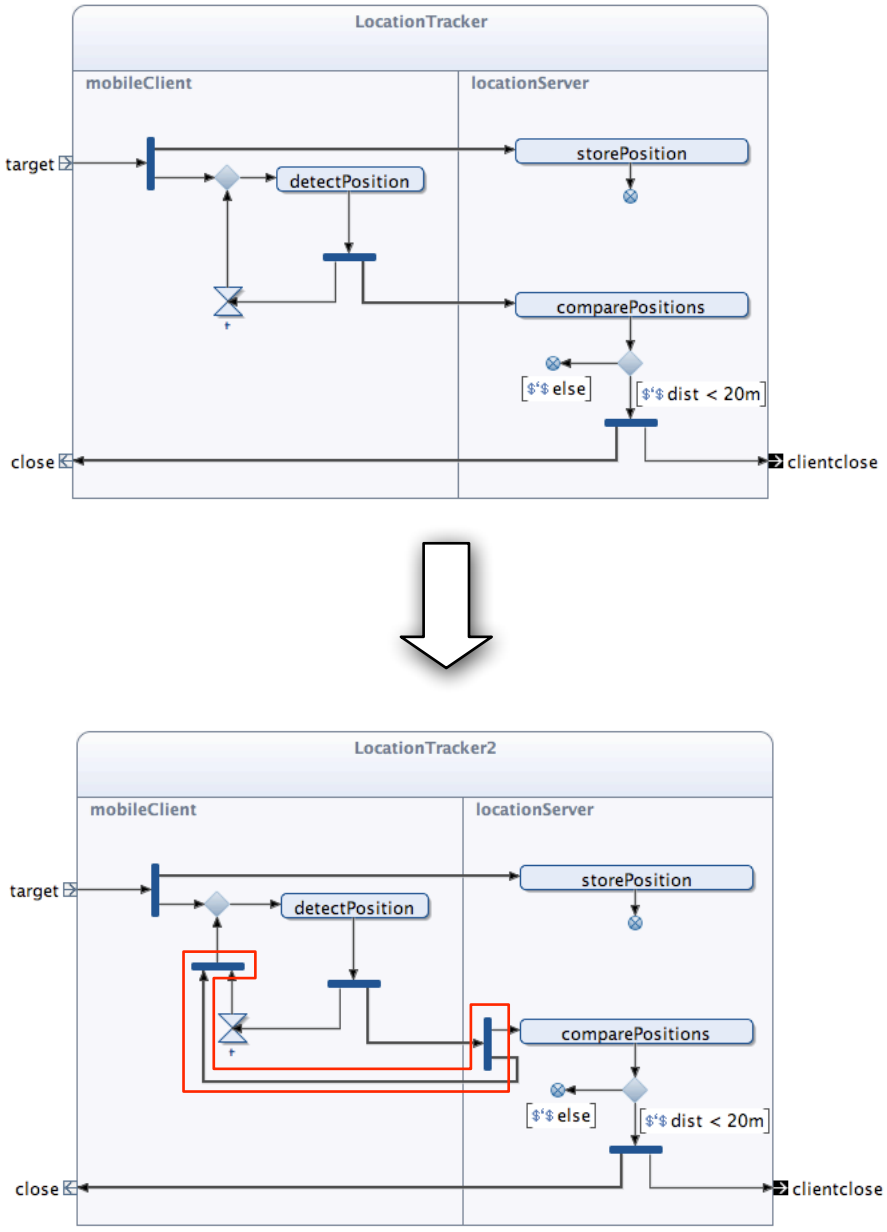


Figure 7.6: Fix: Insert Fork / Join Combo

intentions for the specification. For this specific example, the fix may act more like an inspiration for how to fix the problem.

7.4 Alternative Fix: Give Grant

There is another way to solve the problem of an unbounded queue. The user could give a grant in the form of a UML stereotype, stating that the queue is emptied faster than the producer fills it [KHB06, Sect. 6]. This is not really a fix in that it does not change the model, rather the way the model is interpreted. In TLA$^+$ the proposed grant would take the form of an extra precondition on the action(s) filling the queue "$\wedge$ queue $<$ 5"

Giving such a grant has the unfortunate effect of tying the specification closer to the implementation. The implementation is now responsible for emptying the queue faster than it is filled. This somewhat defeats the purpose of a thorough analysis at the specification level to avoid problems later. Still, there will be times when a queue obviously is not going to be overflown, like when a signal is receive once an hour and the processing of it takes mere milliseconds. Then it is useful for the developer to have the option of giving a grant instead of having to change the specification to accommodate the tool, which should never be the case.

Chapter 8

One-Boundedness

According to the semantics that we assume for activities, all inner places need to be one-bounded (see [KH07b]), meaning that they can hold at most one token. An inner place is any token place such as a timer or a join node except for the queues between partitions.

We will now have a look at the scenario outlined in Fig. 8.1 where a *One-Boundedness* theorem is violated leading us to report a *One-Boundedness Violation* symptom. We will then see how we can set a *Misplaced Merge Node* diagnosis, and in the end we will take a look at the available fixes.

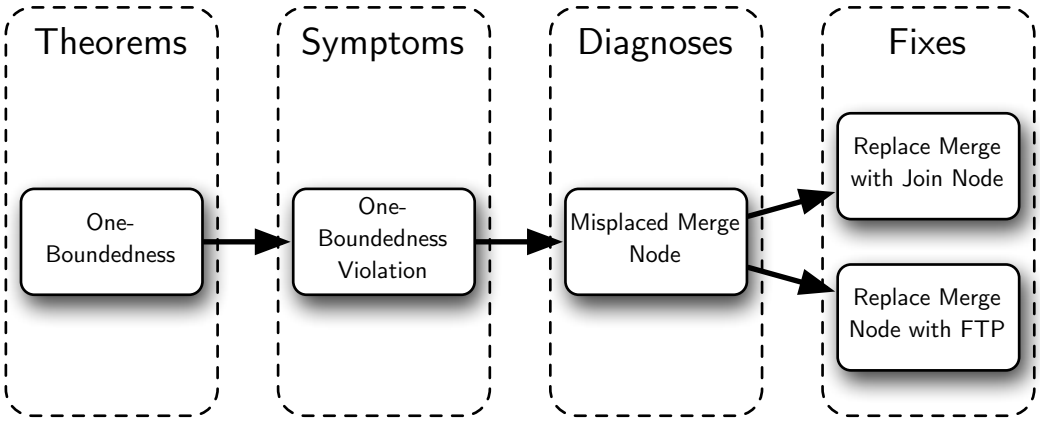


Figure 8.1: Scenario: One-Boundedness

To aid the readability of this section, we provide an example, the *Operate Door* service, as shown in Fig. 8.2. The example is inspired by the *Access Control System* from [KH06] and models the operation of a door.

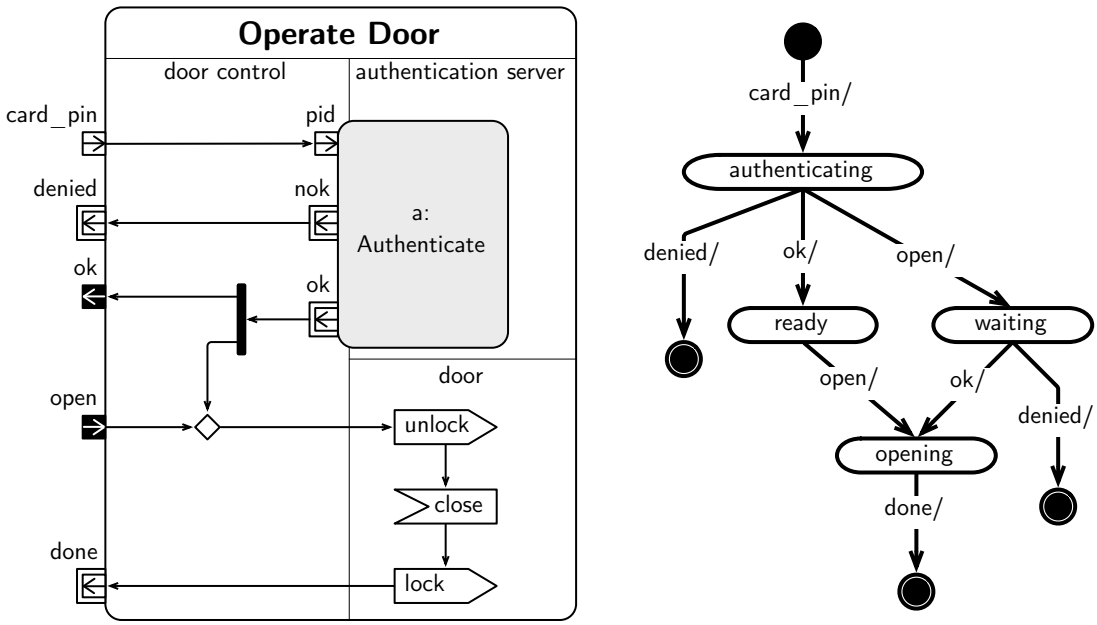


Figure 8.2: Operate Door, First Attempt

The operator must first authenticate, but may send the next command (via pin *open*) even before the authentication is completed. We only model that this command can be to open the door, but imagine that there are more, like locking the door for the night, not allowing any normal users to pass.

8.1 Theorem and Symptom

To assure one-boundedness of all inner places, we need to make sure that any token that can be emitted cannot reach a filled inner place in a single step. We do this by creating a *One-Boundedness* theorem for all starting nodes and every inner place a token sent from them can reach in a single step.

In our example, there is only one theorem like this.The theorem is illustrated in Fig. 8.3 and states that whenever the *Operate Door* service is active/executing, and there is a token in the queue, there must not already be a token in the *close* receive signal action:

$$\begin{aligned} &theorem\_\_status\_DoorControl\_Door\_e6\_close \triangleq \\ &\Box((\_status = \text{“executing”} \wedge DoorControl\_Door\_e6 > 0) \Rightarrow (close = 0)) \end{aligned} \tag{8.1}$$

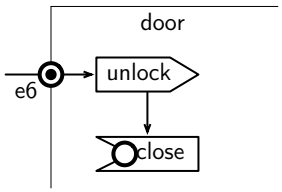


Figure 8.3: Illustration of theorem 8.1

Have another look at the Timeliness Observer example in Sect. 5.4 for several other examples of theorems that are created to ensure one-boundedness.

When a *One-Boundedness* theorem is violated, we report a *One-Boundedness Violation* symptom. The symptom explains that two tokens are about occupy the same token place and reports any diagnoses that are confirmed.

8.2 Diagnoses

A *One-Boundedness Violation* symptom currently only has two specific diagnoses that it attempts to confirm: *Unrestrained Producer* (discussed in Sect. 7.2) and *Misplaced Merge Node*.

A merge node, unlike a join node, for example, passes on every token that it receives. Hence it is a candidate for causing a *One-Boundedness Violation* symptom.

To confirm a *Misplaced Merge Node* diagnosis, the analyzer checks for merge nodes in the trace. It does this by simply checking if the target node of any of the traversed edges in the trace is a merge node. If the same merge node occurs two or more times, meaning that at least two token passed through it, the diagnosis is confirmed.

The *Misplaced Merge Node* diagnosis is only reported by a *One-Boundedness Violation* symptom if no other diagnoses were confirmed. This is because for any example where an *Unrestrained Producer* diagnosis would be confirmed, there would most likely be a merge node in the trace as well. Hence the *One-Boundedness Violation* symptom gives lowest priority to the *Misplaced Merge Node* diagnosis.

When we analyze the behavior of the *Operate Door* building block, theorem 8.1 is violated and a *One-Boundedness Violation* symptom is reported. The analyzer first tries to confirm the *Unrestrained Producer* diagnosis, but does not find anything suspicious. The analyzer then attempts to confirm the *Misplace Merge Node* diagnosis. It finds two occurrences of the merge node in the trace and reports this as a possible cause of the symptom.

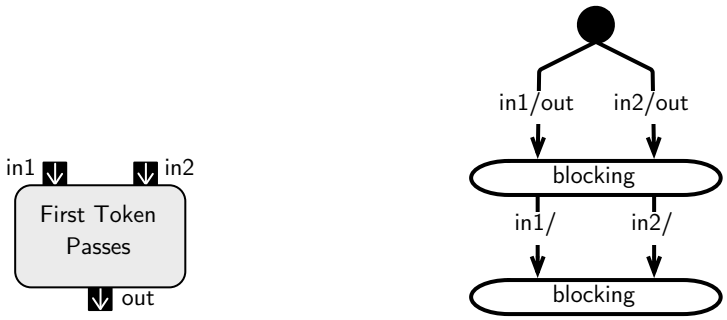
8.3 Fixes

There are currently two fixes available for a misplaced merge node:

1. Replace merge node with join node
2. Replace merge node with a special building block that only lets one token pass.

The first is self-explanatory while the latter needs an introduction.

An *FTP*, *First Token Passes*, is a shallow building block that allows the first token through and then stops all others. The building block with its ESM is shown in Fig. 8.4.[1]



(a) Building Block **(b)** ESM

Figure 8.4: First Token Passes, shallow building block

Both fixes replace the merge node by something that has different behavior, but looks syntactically similar. While the first is obvious, the last one could also serve as a way of informing the user about a building block that he or she was not previously aware of.

In our example, we can easily see that replacing the merge node with a join node is going to change the behavior into the intended one. The application of the fix is illustrated in Fig. 8.5. Note that applying the last fix would also have gotten rid of the symptom. This underlines the point that we still need the user to make choices according to what the intentions really are. We just try to make the process as comfortable as possible.

[1] We show a version with two inputs, but it could have more.

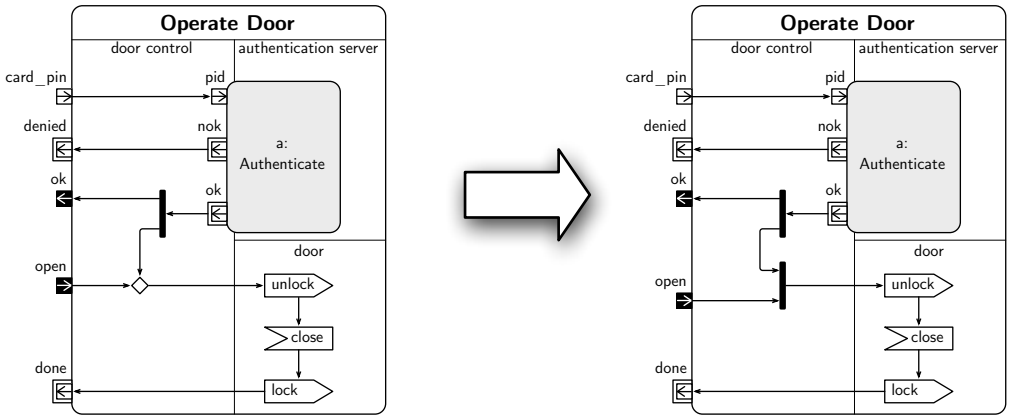


Figure 8.5: Application of *Replace Merge Node with Join Node* on *Operate Door*

Chapter 9

Other Theorems and Symptoms

There are still some theorems and symptoms from Fig. 3.7 that have not yet been discussed. Figure 9.1 depicts the remaining ones, that we will go through in this chapter.

9.1 Respect ESM

An ESM violation is found if a token can enter a call behavior action through a pin when the ESM of the referred sub-activity is not in a state where is accepts this. An example is if the surrounding activity of the *Operate Door* building block (Fig. 8.2) is wired in such a way that it tries to send a token through the *open* pin before it has sent one through the *card_pin* pin.

Figure 9.2 shows an example where we instantiate a *Timeliness Observer* in a system activity. The instance, *to*, is started and after a delay, a token is sent through the *event* pin. Any outputs are simply discarded or terminate the system. The ESM of *Timeliness Observer* is also shown for convenience.

In the example shown in Fig. 9.2, the theorem derived from the step between the timer t and the *event* pin would look like

$$\Box(t = 1 \Rightarrow (to = \text{“running”} \vee to = \text{“waiting”})) \tag{9.1}$$

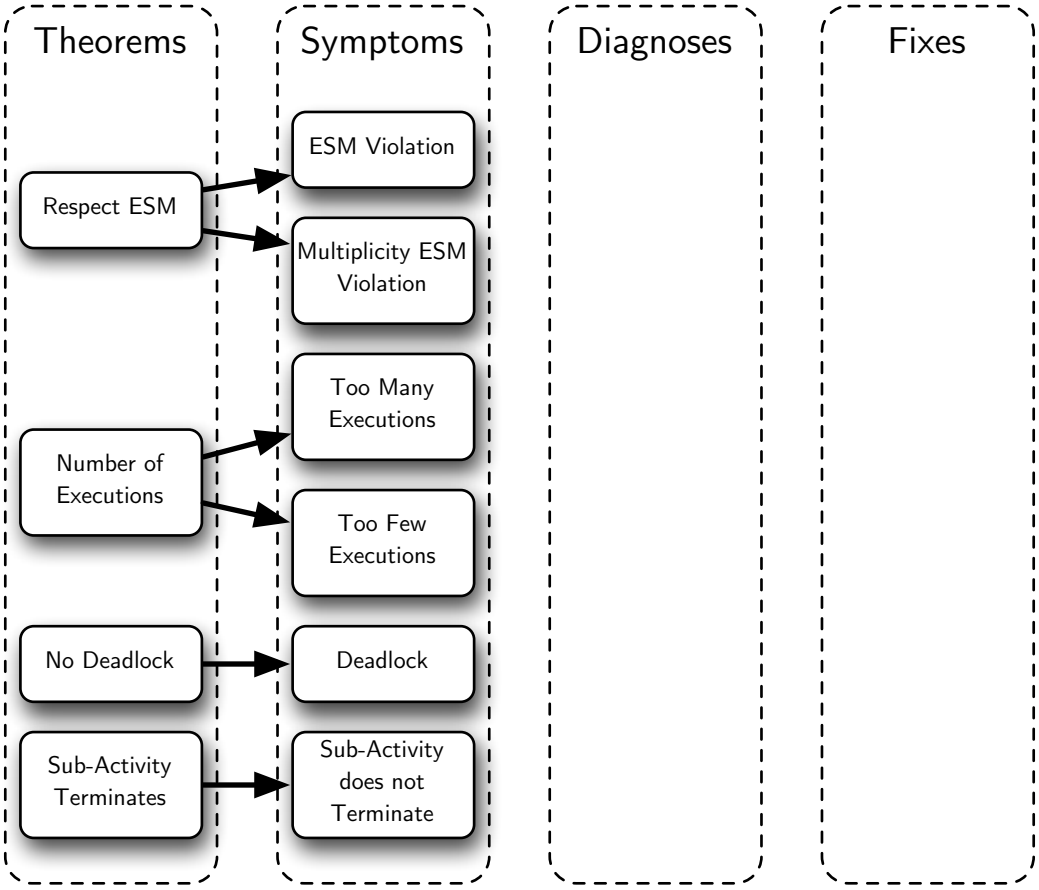


Figure 9.1: Remaining theorems and symptoms

9.1.1 ESM Violation

We report an *ESM Violation* symptom if a *Respect ESM* theorem is violated. Note that this is different from Chapter 5 where we discuss how we can detect if the inside of an activity is consistent with its ESM. Here we check if the environment where such an activity is instantiated, respects that ESM.

We have yet to device any specific diagnoses for this symptom, as it is such a general one. More on this in the following section.

9.1.2 Multiplicity ESM Violation

A *Multiplicity ESM Violation* symptom is detected by a *Respect ESM* theorem (see theorem 9.1) just like an *ESM Violation* symptom. What sets a *Multiplicity ESM Violation* symptom apart from a normal *ESM Violation* symptom is that a *Multiplicity ESM Violation* symptom is reported when a token arrives on a

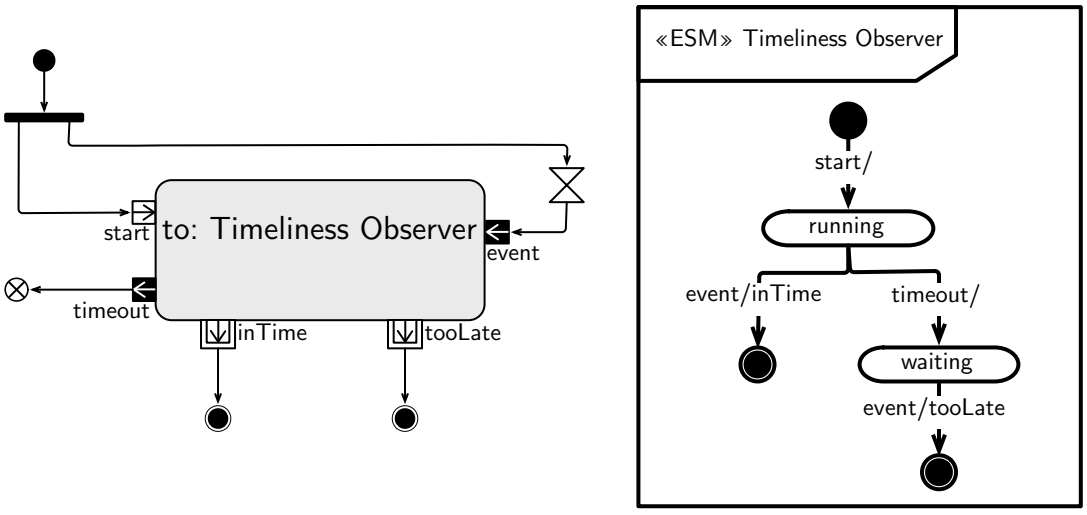


Figure 9.2: A system activity instantiating a Timeliness Observer.

pin of a call behavior action after a transition triggered by this pin has already happened. In this sense, it is similar to a *One-Boundedness Violation* symptom in that several tokens arrive at the same place too quickly.

The reason for separating this symptom from a normal *ESM Violation* symptom is that it warrants that we try to confirm some specific diagnoses. In fact, when a *Multiplicity ESM Violation* symptom is found, we attempt to confirm the same specific diagnoses as for a *One-Boundedness Violation* symptom: *Unrestrained Producer* and *Misplaced Merge Node* (in addition to the general diagnoses).

9.2 Number of Executions

The user may assert that a UML action element should be executed a certain number of times. As with all assertions, this is done by applying a UML stereotype, that contains this meta-information, to the affected element .

As mentioned earlier, the range of the counter variables is 0 to 2, 2 meaning *more than once*. We chose to limit it in this way as it still allows the expression of the most useful execution ranges, yet keeps the state space small.

The user can choose between the following alternatives. The corresponding TLA theorem is shown for each entry.

0..* No restriction on the number of executions. This is the default setting.

0..1 The action may be executed at most once.

$$\Box action\_counter \in 0..1 \tag{9.2}$$

1 The action must be executed exactly once.

$$\Diamond\Box action\_counter = 1 \tag{9.3}$$

1..* The action must be executed at least once.

$$\Diamond\Box action\_counter > 0 \tag{9.4}$$

There are two types of symptoms that can be reported when a *Number of Executions* theorem is violated: *Too Few Executions* and *Too Many Executions*. While there are currently no specific diagnoses available if there are too few executions, we attempt to confirm both the *Unrestrained Producer* and *Misplaced Merge Node* diagnoses if there are too many executions. Additionally, we always try to confirm the general diagnoses.

We can determine which symptom to report by seeing which sub-type of theorem is violated. A theorem like theorem 9.2 being violated, leads to a *Too Many Executions* symptom to be reported.

For the other theorem types, it is not so easy to give a symptom. TLC does not identify what theorem is violated if the theorem expresses a liveness property (see Sect. 2.2.2). Hence we do not get a reference back to the violated theorem, which would contain a reference to the UML action element in question as well as the counter variable name.

We have to work around this problem by finding the violated theorem(s) ourselves. The theorem factory of the analyzer contains a list of theorems that express liveness properties. We check each of these against the trace to find which ones are violated.

Hence, for theorems of the same type as theorem 9.3, we examine the trace to see if they are violated and if there are too many or too few executions. Similarly, if the examination shows that a theorem like theorem 9.4 is violated, it will also show that there were not enough executions.

9.3 Deadlock

A deadlock occurs when there are no tokens that can travel further, yet the activity has not terminated. By terminated we mean that a token exited through

a terminating parameter node or reached an activity final node. To detect a deadlock, we write a theorem

$$\Box(\text{ENABLED } (Next) \vee (\_status = \text{“post\_execution”})) \tag{9.5}$$

which states that either one of the actions in the *Next* statement is enabled or the activity has terminated.

If we detect a deadlock, the user will, as always, be able to view the trace of states and steps leading up to it. Naturally, something along the trace needs to be altered to avoid the deadlock. There are currently no specific diagnoses available, only the general ones.

9.4 Sub-Activity Terminates

Whenever a sub-activity is instantiated through a call behavior action, we want to make sure that it terminates before the surrounding activity does. This is so that we do not have any leftover tokens inside of them that, in the real system, could still be there when the surrounding activity is restarted.

We check for this by creating a theorem

$$\Box(\_status = \text{“post\_execution”} \Rightarrow cba\_name = \text{“\_initial”}) \tag{9.6}$$

for every call behavior action in the activity. It states that the call behavior action must not be active when the activity has terminated.

There are currently no specific diagnoses that the analyzer tries to confirm when a *Sub-Activity does not Terminate* symptom is found. The user can use the trace to see how the activity terminated while the call behavior action was still active and hopefully see what has to be changed.

Chapter 10

Future Work

In this chapter, we outline some ideas for future work based on this thesis. We start by presenting some ideas for further analysis following in the same direction as this work. We then discuss how we may adopt a more holistic approach to the analysis by considering multiple traces, as well as multiple fixes. At the end we outline a modularized architecture for the implementation of the analyzer.

10.1 Further Analysis

We have developed an analysis framework and started filling it with contents. There are surely new properties that can be checked for and more ways to analyze violations. Naturally, we also aim to complete the elements that have been introduced, but are not yet fully developed (see Sect. 6.1.2 and Sect. 6.1.3). We now present some ideas for further contributions.

10.1.1 Partition Termination

Currently, an activity is terminated globally when a token reaches an activity final node or a terminating parameter node. This is a rather large simplification since different activity partitions model parts of distributed components.

We do not currently know what underlying semantics will be the most useful. We need to see what problems we encounter when creating more and bigger specifications. This decision also relies heavily on what we can expect from the underlying execution platform.

One extension to the present framework would be to implement a theorem that

makes sure that all queues inside of an activity are empty when it terminates. Currently, the execution platform is responsible for solving any problems due to tokens that are in transit when an activity terminates.

To investigate further how activity termination can cause problems, we propose an extension where termination is modeled on a partition level. In such a scenario, all knowledge is local to a partition unless it is explicitly shared by sending tokens to other partitions. Hence a partition may not be aware of how another one has terminated their shared instance of an activity. As long as the partition(s) that is (are) still executing do not attempt to send any tokens to the terminated partition, this could be acceptable.

In order to write theorems to check that terminated partitions do not receive tokens, we would first have to alter the way a TLA$^+$ specification is created. We need a separate variable for each partition to contain the execution status of it.

We then imagine that the following would be useful to check for:

- A token is in an incoming queue of a terminated partition. It could be that such tokens would just be discarded by the real system, but this may not be the intention of the developer, who hence should be made aware of it. A worse scenario is that tokens could appear in the next instance of the service and cause unspecified behavior.
- A call behavior action referencing a sub-activity is restarted before it has been terminated in all participating partitions. In the real system, tokens from the new instance could perhaps reach the not yet terminated partition of the old instance and cause unspecified behavior.

One might wish to solve such issues at the implementation level, perhaps by having a session ID on every signal to filter out those that belong to older or newer sessions. Again, this would be a restriction on what kind of executions platforms are supported. We would like our specifications to be as platform independent as possible.

10.1.2 Automatic Refinement Proof

As mentioned in Sect. 7.2.2, the most elegant way to prove more complex properties is to create a TLA refinement proof. But for TLC to check this, one must make a refinement mapping between the variables of an abstract specification that models the property and the detailed TLA$^+$ specification that the analyzer creates. We would like to investigate if this can be done automatically. If so, we believe it would open up many possibilities of checking advanced properties.

Creating automatic refinement mappings would allow us to write "diagnostic theorems", theorems that if violated (or not violated) constitute a diagnosis directly. The theorem described in Sect. 7.2.2 is itself an example of this. If violated, it would tell us of an unrestrained producer, not just an unbounded queue.

One of the main features of our approach is that we can reduce the state space by just considering the ESM of an activity, hence allowing model checking to scale. It is therefore important that every ESM is indeed a correct abstraction of the activity it belongs to. If we can create an automatic refinement mapping, we would like to apply this to checking consistency between activity and ESM. Given the mapping, we could achieve consistency checking by writing a theorem stating that the activity implements (is a refinement of) the ESM.

10.2 Holistic Analysis

The way we currently use TLC, it searches the state space until it finds a violation of a theorem and terminates. We then get a single trace showing how the property is violated. We will now discuss how we can consider more than one trace (or violation) at a time to improve our analysis and hence the support provided to the user.

10.2.1 Check for All Errors at Once

As mentioned in Sect. 2.2.2, TLC has the option of continuing its search even if it finds a violated theorem. It then reports all violations in the end, after searching the entire state space. The problem with this is that the behavior past the first violation may not be useful to us. There are two reasons for this:

1. The following behavior violates our semantics for activities. For example, if a *Bounded Queue* theorem is violated, the following behavior is likely to be that of an ever increasing number of tokens in the queue. In this case, TLC will keep going until it runs out of memory.

2. The following behavior may be irrelevant as fixing the first flaw may change the behavior.

There is a way of avoiding the first problem: being careful of how we write our TLA$^+$ specifications. There are (at least) two styles of expressing properties and TLA actions:

1. The TLA actions simply increase the number of tokens in a token place. A TLA action filling a timer would then contain
   $\wedge\ timer' = timer + 1$

   To check for one-boundedness, we would write a theorem

   $$\Box(timer \leq 1) \tag{10.1}$$

2. On the other hand, we can state that the filling TLA action is only enabled if the timer is empty. A TLA action filling the timer would then contain:
   $\wedge\ timer = 0$
   $\wedge\ timer' = 1$

   To detect a violation, we then have to state that if it is possible for a token to arrive at the timer, it must be empty. If we assume the timer to be filled from an incoming parameter node, we might have a theorem like

   $$\Box((ESM = \text{“active”}) \Rightarrow (timer = 0)) \tag{10.2}$$

For the last alternative, if the timer is already full, another TLA action will be chosen to be executed. If no TLA actions are enabled, a deadlock will occur (and be detected).

By altering our specifications so that all theorems are violated as the property is about to be violated (one state before), and that all actions are specified in such a way that only the legal behavior can happen, we could make sure that TLC only explores semantically correct behavior, even after a violation.

Currently, both inner places and ESM states are already treated this way. Queues between partitions are currently not, but the changes needed would not be great. We would simply need to add a precondition “$\wedge\ queue\_name < 5$” to every action that puts a token into a queue. The *Bounded Queue* theorems would also have to change to say that “whenever a node can emit a token into a queue, that queue has less than five tokens in it”. For a timer that emits a token into a queue, the theorem would look like:

$$\Box((timer = 1) \Rightarrow (queue\_name < 5)) \tag{10.3}$$

Once our TLA$^+$ specifications are made in this way, we could then find all current violations at once (in one run of TLC). While we would still have the problem that fixing one flaw may change the following behavior, we would have the potential for creating better symptoms, diagnoses and fixes by comparing the different violations.

Could we find a way for the analyzer to decide which violations are related? Would it be possible to give them a meaningful ranking, for example by comparing the length of the trace?

In Sect. 7.3, we point out how we risk applying the fix for an unrestrained producer over and over again. If we can group violations by cause and rank them, the analyzer could recommend to apply the fix to the very last (furthest downstream) violation caused by an unrestrained producer. This might be better advice for what to do, but further study is required to determine if there is a pattern we can detect for when to do this.

Finding all property violations at once has another benefit: We may get several traces that violate the same theorem. We will discuss the implications of this in Sect. 10.2.3.

10.2.2 Rank Fixes According to Effect

If the framework for behavioral analysis is expanded sufficiently and we also start checking for all violations at once, there may come a time when developers are somewhat overwhelmed by the number of options given to them. To address the specific issue of a large number of automatic fixes, we think it would be interesting to see if it is possible to rank them according to how likely they are to fix the underlying flaw.

When the analyzer confirms one or more diagnoses, it could make a copy of the model for every fix that is available. It could then apply the fixes to the copies and run itself on the copies. We would then have to find a way to compare and rank the fixes depending on the resulting traces. Maybe the length of the shortest trace could be used as a useful indicator? Perhaps the total number of violations, after the fix has been applied, will tell us how well it worked? Further work is needed to provide answers.

The downside of this approach is likely to be performance. The ranking is more valuable if there is a large number of fixes. But this is when the time to perform the ranking would be the longest. Since applying the fixes and analyzing the altered models are independent tasks, the performance drop could be countered by doing them in parallel.

10.2.3 Find Most Relevant Parts of a Trace

In Sect. 2.4, we discuss some approaches that compare a number of error traces to find the most relevant parts of them. By relevant parts, we here mean the parts of the trace that contain the states and transitions that made this path through the state space lead to a violation.

It would be very interesting to see such an approach adapted to the analyzer.

If we are successful, we could highlight the key transitions when displaying the trace. For long traces, we could use this new information to skip "uninteresting" parts of the trace, so that the developer does not have to click through a large number of inevitable states and transitions, before coming to the relevant part.

In [BNR03], they describe a method for finding the most relevant part of an error trace for a program written in C. They exclude the actual transition that causes a violation from the following runs of the model checker. In this way they can collect several traces leading up to the same violation. This is similar to what we propose in Sect. 10.2.1. In [GV03], they do something similar for Java programs.

Both approaches describe ways of finding the most relevant part of the error traces. Even though they work on the source code of programs, we believe that their approaches could be adapted to ours.

We have already described a way of getting multiple error traces. We could group these according to what theorem they violate. If we could also get TLC to return traces that reach the terminated state without causing any violations, we could analyze the differences between the "successful" traces and the error traces that violate a specific theorem. We briefly present the three types of analysis that are given in [GV03].

Transition Analysis. Traces are here referred to as *positives* (successful traces) or *negatives* (error traces). Traces are divide into pairs $\langle s, a \rangle$ where s is a control state and a is an action. They then define a number of sets for pairs that are represented in all negatives ($all(neg)$) or all positives ($all(pos)$), as well as only in negatives ($only(neg)$) or positives ($only(pos)$). They then define a set $cause(neg)$ as $all(neg) \cap only(neg)$ and a set $cause(pos)$ as $all(pos) \cap only(pos)$. These are interesting as they contain what all negatives (or positives) have in common yet does not happen in the other type of trace. In some cases, these sets may be empty, though.

Invariant Analysis. In this analysis, they compute data invariants over negatives and positives. They choose instrumentation points in the code where they check the variable values. They use an existing approach for dynamically discovering invariants. Finally, they compare the invariants from the negatives to those from the positives.

Transformation of Positives into Negatives. They find the smallest number of changes that must be done to a positive to get a negative. This is called the *minimal transformation*. They can do this from every positive to every negative and then sort the result according to transformation size. The smallest transformations are likely to point out the relevant parts that can cause errors. They can also run the transition analysis on the pairs that

constitute the transformed sections of the traces. This can give non-empty causal sets where the normal transition analysis failed to do so.

10.3 Modularized Architecture

In the current architecture (see Fig. 4.1), the *Formulator* creates the *TLA Module* which again creates the *Theorem Factory*. This creates a tight coupling between these three parts. You cannot have one without the others.

Ideally, the *Formulator* should simply produce a list of steps showing what nodes and edges are traversed by a token in an atomic transition of the activity. This list would then be independent of any specific target language and simply be a representation of the semantics of our kind of UML activities. This could be used by advanced syntactic inspectors as well as a starting point to transform the activity into input for any model checker.

The list of steps from the *Formulator* would then be input to the *TLA Module* that would build itself. Finally, the *Theorem Factory* would take the *TLA Module* as input to create theorems. The *Controller* could then request that the *Theorem Factory* return the TLA$^+$ specification including theorems.

This kind of architecture would reduce the dependencies that exist in the current one. Hence it would be easier to add new things or alter existing code without having to spend so much energy on testing and debugging.

Chapter 11

Conclusion

We set out to provide a developer, using Arctis, with the benefits of formal methods for verifying behavioral properties, but without requiring any knowledge in the domain of formal methods.

We started by giving a summary of the relevant theory and the work that we build on. We presented a framework, containing theorems, symptoms, diagnoses and fixes, for the analysis of a property violation, through a combination of syntactic and behavioral analysis. Later, we described each element of the framework in detail, providing examples of their application for many of them.

In response to our goal, we presented the *Arctis Analyzer*, an extension to the Arctis tool suite. The analyzer expresses our semantics for a UML activity (and its ESM) in the form of a TLA$^+$ specification. It also generates a number of properties that must hold. It runs the model checker TLC to detect any property violations. Any problems found will be accompanied by an error trace. The analyzer can visualize this error trace in terms of the graphical model the user is working on.

The analyzer uses the analysis framework to interpret the violation of a property. In the event of a violated property being found, a symptom will be reported and the analyzer may attempt to confirm a number of diagnoses. If a diagnosis is confirmed, an automatic fix may be provided to correct the activity.

To summarize, the analyzer gives Arctis the capability of supporting the user with formal methods, linear-time temporal logic in our case, without the user ever having to see a single formula. This is achieved by the following features:

- The transformation from UML to TLA$^+$ requires no user input.
- Theorems for properties are automatically generated (or generated from high-level input on the UML model, by the user).

- Error traces from the model checker are visualized in terms of the UML model.
- A framework has been developed for interpreting the property violations and uncover the underlying flaws.
- For some flaws, the framework supplies automatic fixes.

To the best of our knowledge, there are no other tools for specifying reactive systems that have all of these capabilities (see Sect. 2.4). In particular, we have found none that can provide automatic fixes.

We have implemented much of the analysis framework in the analyzer tool. The analyzer has also been upgraded, from the previous formulator, to express the semantics of a sub-activity, not only complete system activities. We presented the architecture of the implementation, explaining the task of every entity.

We have included screenshots showing how the implemented analyzer is used by a developer. Together with the theoretical sections of the thesis, we believe they give a convincing argument for how the initial goal has been met: Providing behavioral analysis through formal methods, to a developer that has no knowledge of the applied formalism.

It is our hope that having a tool that provides behavioral analysis already at an abstract level, will encourage developers to spend more time in the early stages of development, weeding out problems when they are still quick and inexpensive to fix.

List of Figures

Bibliography

[Aal98] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. URL `citeseer.ist.psu.edu/vanderaalst98application.html`.

[Aal99] W. M. P. van der Aalst. Woflan: a Petri-net-based workflow analyzer. *Syst. Anal. Model. Simul.*, 35(3):345–357, 1999. ISSN 0232-9298.

[BAL97] Hanene Ben-Abdallah and Stefan Leue. 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.

[BBK+04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Proceedings of the International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2004.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. 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*, pages 1–20, October 2004.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL*. The BCS Practitioner Series. Prentice Hall International, 1993. ISBN 0-13-034448-6.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003. ISSN 0362-1340.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM Toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, London, UK, 2001. ISBN 3-540-42345-1.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, London, UK, 1982. ISBN 3-540-11212-X.

[DS03] Yang Dong and Zhang Shensheng. 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*, pages 47 – 54. Huntsville, AL, USA, 2003.

[Ecl08a] Eclipse Modeling – MDT – Home, April 2008. URL `http://www.eclipse.org/modeling/mdt/?project=uml2`.

[Ecl08b] Eclipse.org home, April 2008. URL `http://www.eclipse.org`.

[Esh06] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1): 1–38, 2006. ISSN 1049-331X.

[FF06] Christian Flender and Thomas Freytag. Visualizing the Soundness of Workflow Nets. In *Proceedings 13th Workshop Algorithms and Tools for Petri Nets, AWPN*, pages 47–52. Hamburg, Germany, 2006.

[Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, Norwegian University of Science and Technology, 2003.

[GCKK06] Heather Goldsby, Betty H. C. Cheng, Sascha Konrad, and Stephane Kamdoum. A visualization framework for the modeling and formal analysis of high assurance systems. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 707–721. Springer, 2006. ISBN 3-540-45772-0.

[GM05] Nicolas Guelfi and Amel Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 283–290. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2465-6.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135. Portland, Oregon, May 9–10, 2003.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HK07] Peter Herrmann and Frank Alexander Kraemer. Design of Trusted Systems with Reusable Collaboration Models. In *Joint iTrust and PST Conferences on Privacy, Trust Management and Security*. IFIP, 2007.

[Hol03] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[Hoo02] Jozef Hooman. Towards Formal Support for UML-based Developement of Embedded Systems. In *Proceedings PROGRESS 2002 Workshop, STW*, 2002.

[JDS08] The Data Structures Library in Java, March 2008. URL `http://www.jdsl.org/`.

[JGr08] Welcome to JGraphT - a free Java Graph Library, March 2008. URL `http://jgrapht.sourceforge.net/`.

[JUN08] JUNG - Java Universal Network/Graph Framework, March 2008. URL `http://jung.sourceforge.net/`.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07a] Frank Alexander Kraemer and Peter Herrmann. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, October 2007.

[KH07b] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In *Prooceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Kra07] Frank Alexander Kraemer. Building Blocks, Patterns and Design Rules for Collaborations and Activities. Avantel Technical Report 2/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, March 2007. Draft.

[Kra08] Frank Alexander Kraemer. UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, June 2008.

[KSH07] Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Engineering Support for UML Activities by Automated Model-Checking — An Example. In *Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE)*, November 2007.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. ISSN 0164-0925.

[Lam96] Leslie Lamport. Refinement in state-based formalisms. Technical report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, December 1996. URL `http://research.microsoft.com/users/lamport/pubs/refinement.pdf`.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison Wesley, 2002.

[LP99] J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. *14th IEEE International Conference on Automated Software Engineering*, pages 255–258, October 1999.

[Mik99] Tommi Mikkonen. The two Dimensions of an Architecture. In *WICSA1, First Working IFIP Conference on Software Architecture*, 1999.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. ISBN 0-387-97664-7.

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.2, November 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[Rus00] *Disappearing formal methods*, 2000. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=895446`.

[SGK+05] Stefan Sarstedt, Dominik Gessenharter, Jens Kohlmeyer, Alexander Raschke, and Matthias Schneiderhan. ActiveChartsIDE - An Integrated Software Development Environment comprising a Component for Simulating UML 2 Activity Charts. In *The 2005 European Simulation and Modelling Conference (ESM'05)*, pages 66 – 73, October 2005.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[Stö05] Harald Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. In *Electronic Notes in Theoretical Computer Science*, volume 127, pages 35 – 52, 2005. ISSN 1571-0661.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3. IEEE Computer Society, Washington, DC, USA, 2000. ISBN 0-7695-0710-7.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — A Tool for the π-Calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[Wop08] Welcome - The WoPeD Homepage, May 2008. URL `http://woped.ba-karlsruhe.de/woped`.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999. ISBN 3-540-66559-5.