Christian-Albrechts-Universität zu Kiel

Diploma Thesis

A Constructive Model/View Approach for the Refinement of UML 2.0 Sequence Diagrams

cand. inform. Marco Zingelmann

November 26, 2007

Department of Computer Science
Real-Time and Embedded Systems Group

Advised by:
Prof. Dr. Reinhard von Hanxleden
Dipl. Phys. Carsten Ziegenbein[*]

[*]Philips Medical Systems, Hamburg, Germany

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

An industrially proven technique to manage the development of complex systems is to begin with an abstract high-level description of the requirements that are stepwise refined with increasing level of detail. UML 2.0 sequence diagrams provide a common, well-suited graphical formalism to specify requirements. Developers are thereupon able to model the whole behavior of the system under development based on the refined, detailed requirements. UML sequence diagrams, however, present only loosely coupled partial views of the system's behavior and, thus, the risk of inconsistencies is inherent.
To overcome this problem, this thesis introduces a constructive model/view concept for UML 2.0 sequence diagrams, which understands diagrams as views on an underlying formalized model. The introduced concept supports different levels of abstraction and adopts changes in one level of abstraction into all dependent levels. Constructive modifications of a model force the user to resolve ambiguities while they occur. This results in an all-time consistent model. The consistency definitions used in this thesis aim at the refinement relation and the completeness of each requirement level. Furthermore, these definitions allow highly efficient algorithms by addressing sequence diagram consistency at a syntactical level.

Keywords UML, sequence diagrams, model view, constructive development, consistency, refinement

Preface

My thanks go to André Ohlhoff for the constructive discussions concerning various aspects of my thesis and the deeply reading of this work. In addition, I would like to thank Jan Täubrich for his advice, reviews, the given lift, and the fruitful discussions also during the return journeys. Last but not least, my thanks go to Björn Lüdemann for the reviews and the challenging new ideas.

Contents

List of Figures

Glossary of Symbols

Notation	**Description**	**Page**
SD	a sequence diagram	9
I	a set of instances	9
E	a set of events	9
O	a set of sequence diagram objects	9
ι	$E \to I \times O$ maps events to lifelines and objects	9
$<$	$E \times E$ is a partial event order	9
η	$O_O \to O_F$ maps operands to fragments	9
t	$O_F \to \{\text{alt}, \text{break}, ...\}$ determines fragment type	9
τ	$O_O \to \Sigma$ assigns a constraint to an operand	9
O_M	set of messages	9
O_F	set of fragments	9
O_O	set of operands	9
O_R	set of references	9
O_S	set of states	9
O_A	set of actions	9
O_{CO}	set of continuations	9
E_{MS}/E_{MR}	a set of message send / receive events	9
E_{OB}/M_{OE}	a set of operand begin / end events	9
E_R	a set of reference events	9
E_S	a set of state events	9
E_{AB}/E_{AE}	a set of action begin / end events	9
E_{CO}	a set of continuation events	9
$E\|_i$	restricts E to events that belong to instance i	9
$<\|_i$	forms a total order for all events of instance i	9
$!m$	refers to the sending event of m	12
$?m$	refers to the receiving event of m	12
h	an instance hierarchy	15
μ	maps refining instances to their master instance	15
π	maps child instances to their parent instance	15
PN	a Petri net	19
P	a non-empty set of places	19
T	a non-empty set transitions	19
F	$(P \times T) \cup (T \times P)$ a flow relation	19

Notation	**Description**	**Page**
m	$P \to \{0,1\}$ a Petri net marking	19
$\bullet p$ / $\bullet t$	pre-set of place p / transition t	19
$p\bullet$ / $t\bullet$	post-set of place p / transition t	19
SR	a state relation	31
G	a partial function that relates messages with their refinements	32
M	a data model	37
σ	$O_S \to I$ maps states to instances	37
$Seqs$	a set of sequences	37
Seq	a sequence in the data model	37
f	maps levels of abstraction to a set of instances and an event order	37
SDV	a sequence diagram view	38
l	a level of abstraction	38
VI	a set of instances	38
ξ	$VI \to E \times E$ maps each instance to a start- and end event	38
MV	a model/view system	39
V	a set of sequence diagram views	39

1. Introduction

The major challenge of today's software development is to handle the ever-growing complexity, *e.g.*, the product size or the level of distribution. Failures occur during requirement definition, software design, implementation, and testing. It is well-known that the earlier an error is discovered, the less it costs [16]. Software development processes try to address the complexity issue with defined sequences of work steps, specified roles, and outputs. The *Rational Unified Process* (*RUP*) [13] is such a software development process with the following major disciplines: *Business Modeling*, *Requirements*, *Analysis & Design*, *Implementation*, *Test* and *Deployment*. The process distinguishes four different phases, namely *Inception*, *Elaboration*, *Construction*, and *Transition*, with several iterations for each phase. The output of each phase is a *milestone*. The RUP is based on *use cases*, which are described in prose or a more formal notation to master all defined disciplines. The *Unified Modeling Language* (*UML*) [26] is the common formalism for the unified process and a wide spread language in the industrial environment. The UML defines a set of formalisms to describe parts of the system under development from different perspectives and at different levels of abstraction; most of them are graphically represented. The UML as well as the RUP are in continuous development to meet today's requirements in software engineering. The current major version 2 of the UML describes six structural and seven behavioral diagrams. Two important behavioral diagrams are state machine diagrams based on Harel's Statecharts [8] and sequence diagrams. State machines describe the behavior of the system or a sub-system, whereas sequence diagrams specify the behavior between different systems or sub-systems [9]. Class diagrams are important structural diagrams, which specify relations between different classes. *Classes* can be specified by an annotation called *stereotype* to an explicit element, and a concrete incarnation of a class is called an *instance*. During all disciplines these diagrams can be used to specify the system under development in an abstract manner and provide a base for dependent disciplines.

UML diagrams, however, are only loosely coupled partial views on the system. Different diagrams may show the same structure or behavior from different perspectives, which introduces redundancy. The number of diagrams will grow throughout the development process and changes to existing diagrams will be made in consecutive iterations of the development process. Elements changed in one diagram have to be updated in other diagrams, which depend on this element, to prevent inconsistencies. Thus, consistency of the total system is hard to maintain.

This thesis introduces a consistent and constructive model/view concept based on an iterative software development process using refinement of diagrams. It distinguishes between diagrams as views and an underneath model. As a result, changes in one

view can be observed in all views on the modified part of the model. Whenever ambiguities arise, this concept provides interactive support to resolve them.
In the remainder of this chapter, Section 1.1 describes the environment in which this thesis was created. Section 1.2 briefly introduces UML sequence diagrams. Furthermore, Section 1.3 presents the objectives of this thesis and finally, Section 1.4 outlines the further chapters.

1.1. Environment

This diploma thesis was developed in cooperation with a software development team at *Philips Medical Systems* (*PMS*) [23] in Hamburg, Germany. This team develops real-time embedded software for high-voltage X-ray generators based on a *ROOM* framework [25]. ROOM stands for real-time object-oriented modeling. The units of composition in the ROOM framework are called *capsules*, which are encapsulated entities communicating over defined ports with other capsules. Capsules are independent and concurrent, and their internal structure is hidden to the environment. This description matches to the UML term *component*, which is, such as a capsule, a stereotyped class. Throughout this thesis the terms capsule, component, and instance are used synonymously. Capsules can be hierarchically composed with subcapsules. The behavior of a capsule is defined by a state machine, which is used for transformation into implementation code in a classical programming language such as C++ or Real-time-Java.
UML sequence diagrams describe external behavior between different instances at the interface level. Because of the restriction of capsules to communicate only over ports with each other, sequence diagrams are well-suited for specifying the behavior between capsules. The software development team at PMS uses a customized version of the Rational Unified Process and makes heavy use of sequence diagrams at all development disciplines. Thus, requirements for the system under development are based on use cases and formalized by sequence diagrams. Starting at an abstract high level, consequent refinement is used to structure the system; components are split-up and for that reason more internal behavior is added. This process is separated in different levels and is not limited to the requirement discipline. Figure 1.1 depicts an exemplary view on this process.

The figure is separated into two parts: the upper half describes the requirement process, where basic scenarios are refined into detailed scenarios. The last requirement level n forms the first level $n+1$ of the implementation. During the implementation process requirements are refined to physical realities *e.g.*, messages are renamed to fulfill given protocols or bus systems. Message orderings in the requirements discipline, which may result in race conditions while arbitrary execution have to be resolved during the implementation phase. The bottom of Figure 1.1 depicts that state machines are used to model the whole behavior of the components at the end of the refinement process. Finally, these state machines are transformed into source code.

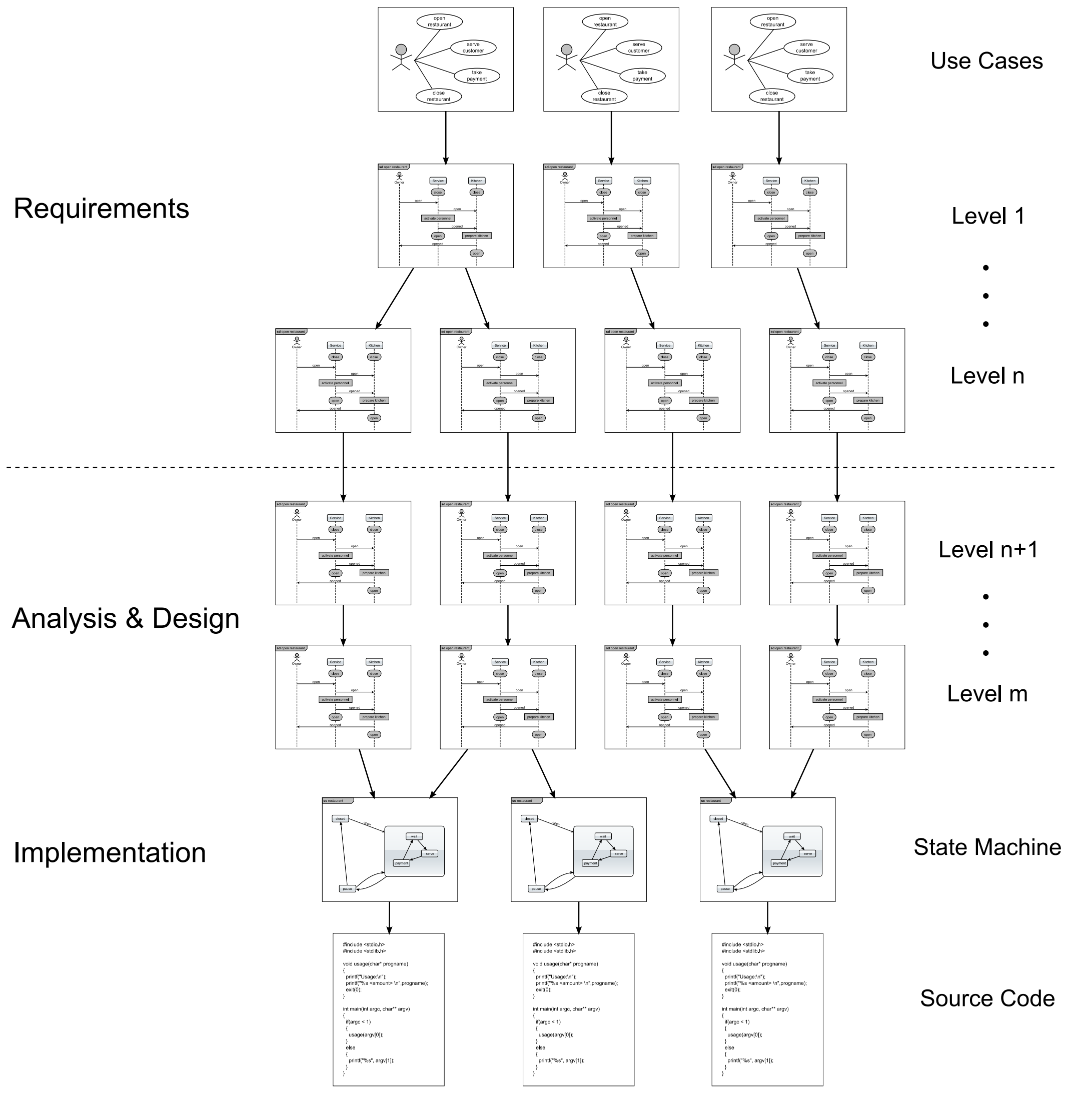


Figure 1.1.: Development process.

1.2. Sequence Diagrams

Sequence diagrams are part of the UML and belong to the class of behavioral diagrams. In the upper left corner of a UML diagram there is a pentagon containing the type of the diagram followed by the label of that diagram. In the case of sequence diagrams the type is *sd*. Sequence diagrams represent the participating components in the horizontal dimension and the temporal progress of the interaction in vertical dimension. Every contained component has a dashed lifeline, which describes the temporal progress for that component. Exchanged messages between two components are represented through arrows between the corresponding lifelines, whereas the arrow is annotated with the label of the message. Figure 1.2 depicts a simple sequence diagram with two components and two messages.

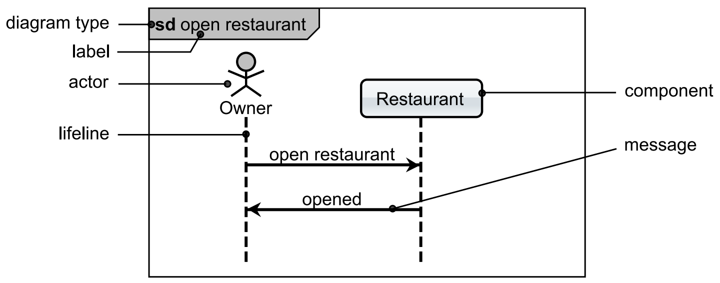


Figure 1.2.: Simple sequence diagram.

The component Owner is an actor, which is drawn as stick-figure. At PMS actors are those components of interactions that initiate the communication by sending the first message to another component.
All examples used throughout this thesis are based on a restaurant-example formerly introduced by Lischke [17] and Lüdemann [18] and extended by Ohlhoff [21]. Its plain components allow small examples on the one side and on the other side all refinement and consistency terms can be well explained.
Because of their simplicity, sequence diagrams can take several tasks in the development process:

Modeling As mentioned in Section 1.1, sequence diagrams can be used to specify the inter-object behavior of different components. They can be used at the requirement level as well as at the implementation level to describe valid scenarios that a component has to fulfill. Moreover, UML 2.0 allows describing invalid scenarios, too.

Communication Sequence diagrams are a useful communication medium when talking to different stakeholders. It is easier to discuss desired and unintended behavior on the basis of a simple graphical representation. In terms of the described development process, different levels of abstraction suit different stakeholders.

Documentation Traceability and documentation is an important part of many development processes and often required due to domain specific regulations. Sequence diagrams can be annotated with comments to clarify customer demands at requirement level and to explain design decisions at implementation level.

Verification Sequence Diagrams describe traces of the system under development. This information can be used to verify the execution of the system. A test environment may take the role of the actors and send the first message to start the specified interaction. The sent and received messages can be compared with the ones defined by the sequence diagram without human interaction. In a regression test many of such sequence diagrams can be automatically tested to guarantee that new software versions still implement the specified scenarios.

In UML 1.x the expressiveness of sequence diagrams is limited to mainly one trace between the described components or a set of traces in case of concurrency, respectively. Many concepts of *Message Sequence Charts* (*MSC*) [14] and *Live Sequence Charts* (*LSC*) [2] have been adopted in UML 2.0 and definitely increased the expressiveness of sequence diagrams. One important concept is combined fragments, which group a set of messages together with an operator such as loop, par, or strict. UML 2.0 defines a set of 12 operators and every fragment is separated in several operands by a dashed line. Each fragment has at least one operand containing the corresponding sequence diagram elements. Furthermore, combined fragments can be arbitrarily nested. The impact of one fragment is limited to the lifelines and objects it covers. Each operand can be annotated with a constraint that specifies whether the operand is executable.
Figure 1.3 depicts a sequence diagram with two combined fragments, where the outermost alt has two operands and describes an alternative trace. After the Owner sent his request to open the Restaurant, the answer depends on the status of the Restaurant. The innermost fragment opt has one operand and describes an optional message, which will be send only on Mondays, because on Monday the new offers have to be unhinged.

Each message shown in a sequence diagram consists of a message send event on the senders lifeline, a message receive event on the receivers lifeline, and the message itself. Thus, a trace of a sequence diagram is an ordered list of event occurrences. The semantics of sequence diagrams is given as a pair of sets of traces. On the one hand there are valid traces given through sequence diagrams with no fragments or operators such as alt, par, or opt. On the other hand there are invalid traces, given by operators such as neg, which describes traces that shall not occur. The union of these two sets does not have to cover all possible traces.
Figure 1.4 shows an overview of the most important sequence diagram elements. *Coregions* represent that the enclosed events may occur in arbitrary order. A *General Ordering* depicts that independent events can be ordered. *Gates* are message connection points to send messages to the outside of an interaction or combined fragment. A *Reference* is a mechanism to reuse a former defined interaction in the

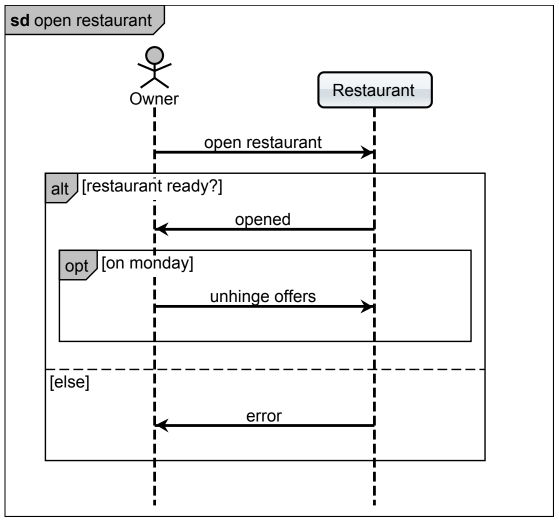


Figure 1.3.: Sequence diagram with two combined fragments.

current diagram, which supports the *Write Things Once* or *WTO* principle. *State Invariants* annotate lifelines with runtime conditions, such as the value of a variable or internal state. A *Local Action* specifies the execution of an action within the lifeline. *Continuations* are used in combination with two different alt fragments, they transport the decision, which operand was chosen in the first alt fragment, to the second alt fragment.

1.3. Objectives

The primary objective of this thesis is to introduce a constructive model/view concept for the refinement of UML 2.0 sequence diagrams with support of the sketched software development process (cf. Section 1.1). Working with the defined model should be possible in a constructive way to resolve arising ambiguities through an interactive dialog. The benefit of an interactive process is that operations that would lead to an inconsistent model can be denied. In consequence, model operations have to form a transaction, which could be only completed if all ambiguities are solved. Having a consistent model at any time avoids many errors before they occur. Some further objectives arise from this approach:

- Define all necessary model operations and introduce a refinement concept for local states, by using Ohlhoff's *Consistent Refinement of Sequence Diagrams in the UML 2.0* [21].

- Extend Lischke's *consistency of sequence diagrams of the same abstraction level* for UML 1.4 sequence diagrams [17] to incorporate new elements from UML 2.0.

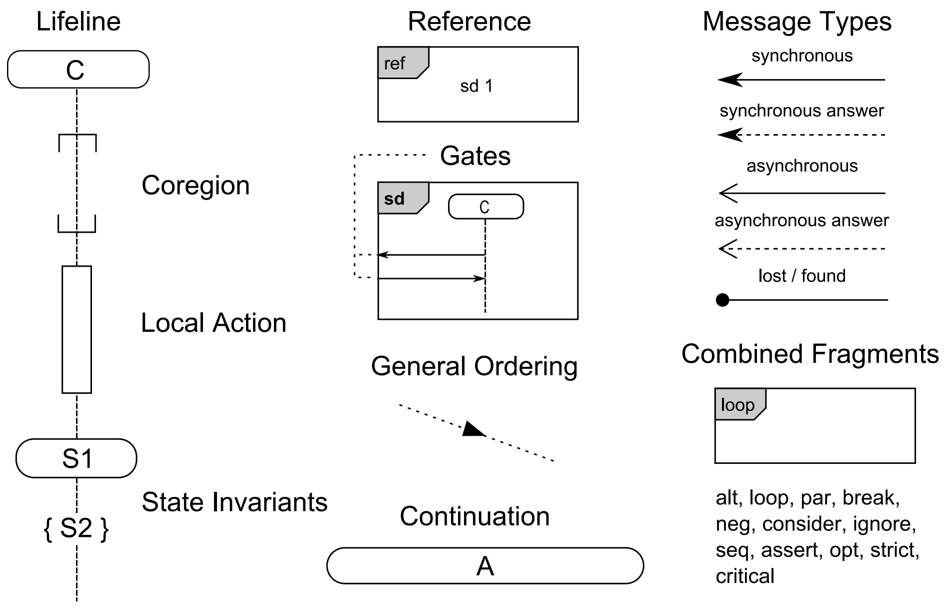


Figure 1.4.: Sequence diagram elements.

- Determine valid integrations and classify them according to consistent traces from top level to bottom, if ambiguities arise throughout the refinement process.
- Introduce a simple message refinement concept to support the differentiation between requirement level and implementation level in the same underlying model.

1.4. Overview

The remainder of the diploma thesis is organized as follows: Chapter 2 introduces important consistency notions for sequence diagrams and the refinement process. These terms are used in subsequent chapters to define a consistent model behind the sequence diagrams. Chapter 3 describes the model/view concept, the necessary data model, and all model operations. Furthermore, Chapter 3 introduces how ambiguities, which may arise through the refinement steps, can be solved interactively. Chapter 4 presents a Java prototype for the main ideas of the model/view concept. Chapter 5 provides an overview of related work and points out relations to this thesis. Finally, Chapter 6 assesses the results of this thesis and presents further work.

2. Consistency / Refinement

As mentioned in the introduction, the main objective is to maintain at all times a consistent model. Hence, a notion of consistency has to be defined and satisfied by any model operation to gain this objective. Therefore, this chapter describes the refinement principle of intra- and inter-level refinement that was introduced by Ohlhoff [21] and the horizontal consistency concept of a level state machine introduced by Lischke [17]. These ideas build the basement of this thesis. The level state machine concept is extended to support new elements of UML 2.0 and the refinement principles used here.
Checking consistency of sequence diagrams on the trace level is easy. It is simple to check if a trace is included in another trace with defined projections. Due to the large amount of traces of even small sequence diagrams, it is, however, inefficient to address consistency checks at the trace level. Furthermore, there exists no rigorous semantics for all sequence diagram elements in UML 2.0, such as negative traces [24]. All consistency notions defined in the remainder of this chapter are highly efficient by addressing sequence diagram consistency on a syntactical level. For that reason, these concepts avoid nonspecific semantics and can be easily extended to all future interpretations of the UML elements. As a result of addressing consistency at the syntactical level, these concepts allow constructive modeling, in contrast to concepts based on post-operational model checking.

2.1. Inter- / Intra-level Refinement

This section recapitulates the inter- and intra-level refinement concept that was introduced by Ohlhoff [21] and transforms it to the ideas presented in this diploma thesis. The introduction of the refinement concept is, however, informal. Chapter 3 formalizes the ideas in connection with the model operations.

Section 1.2 introduced sequence diagrams in an informal way. The following definition provides a formal base that further definitions and algorithms will refer to. The further concepts, definitions, and examples in this section are taken from Ohlhoff's thesis [21] and adjusted to the new formalization and to the needs of the following chapters.

Definition 2.1 (Sequence Diagram):
A sequence diagram (SD) is a tuple

$$SD = (I, E, O, \iota, <, \eta, t, \tau)$$

where

I	is a set of instances, disjoint union of I_a (actors) and I_c (components),
E	is a set of events,
O	is a set of sequence diagram objects,
ι	$: E \rightarrow I \times O$ maps events to lifelines and objects,
$<$	$\subset E \times E$ is a partial event order,
η	$: O_O \rightarrow O_F$ maps operands to fragments,
t	$: O_F \rightarrow \{\text{alt}, \text{break}, ...\}$ determines fragment type, and
τ	$: O_O \rightarrow \Sigma$ assigns a constraint in an alphabet to an operand.

The set O of sequence diagram objects is a disjoint union of the following sets:

O_M	set of messages,
O_F	set of fragments,
O_O	set of operands,
O_R	set of references,
O_S	set of states,
O_A	set of actions,
O_{CO}	set of continuations.

The set E of events is a disjoint union of the following sets:

E_{MS}/E_{MR}	set of message send / receive events,
E_{OB}/M_{OE}	set of operand begin / end events,
E_R	set of reference events,
E_S	set of state events,
E_{AB}/E_{AE}	set of action begin / end events,
E_{CO}	set of continuation events.

The following shorthands will be used later:
$E|_i := \{e \in E \mid \exists\ o \in O : \iota(e) = (i, o)\}$ restricts E to events that belong to instance i and
$<|_i := \{(e, e') \in < \mid \exists\ o, o' \in O : \iota(e) = (i, o) \wedge \iota(e') = (i, o')\}$ forms a total order for all events of instance i.

This definition introduces many symbols to distinguish all sequence diagram elements. The key concept, however, is not difficult: Every element, *e.g.*, a message, in a sequence diagram is an object and each object is associated with a set of events, which represents the object's occurrence on a lifeline. Figure 2.1 shows a sequence diagram annotated with the events and objects from the definition to demonstrate how the visual presentation and the formal definition match. The only notable difference is the representation of operand delimiters. In the diagram operands are separated by a dashed line and in the formal representation there is one operand end event and one operand begin event. The advantage of this approach, in contrast to something such as a next operand event, is that every operand can be handled in the same manner. The functions η, t, τ are necessary to formalize the conditions for start and end states (cf. Section 2.2), *e.g.*, τ allows to formalize operand assertions with a special meaning, such as an else operand for alt fragments. Another important fact is that every lifeline has a total order of its events, which reflects the visual representation and not the logical flow. Due to the total order of each lifeline, a minimum (min) and maximum (max) element exists for each $E|_i$ with $i \in I$.

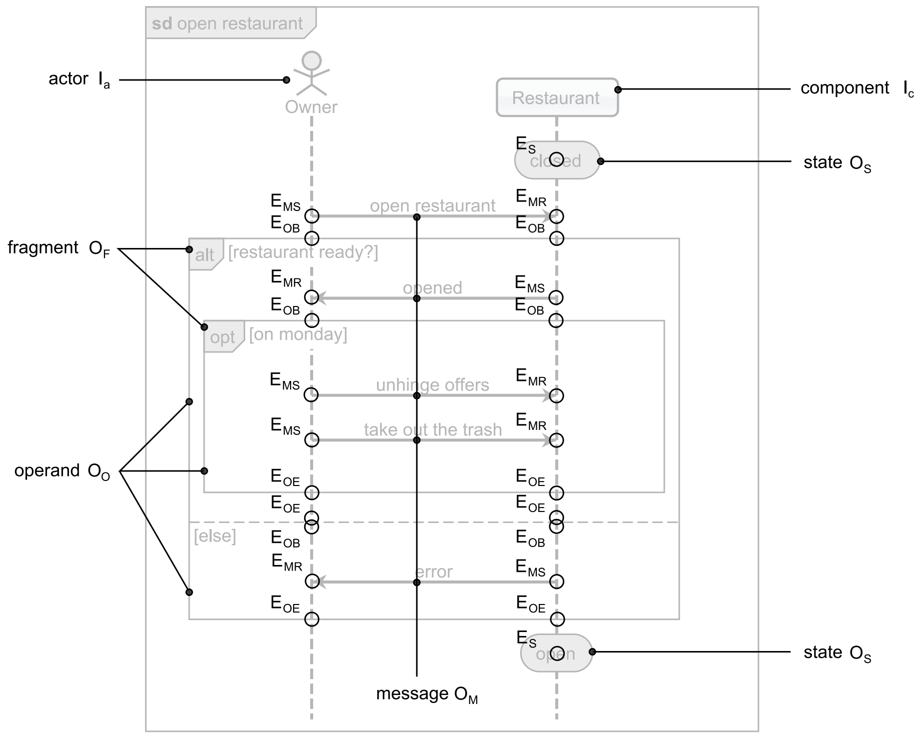
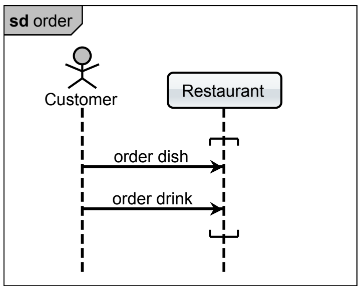


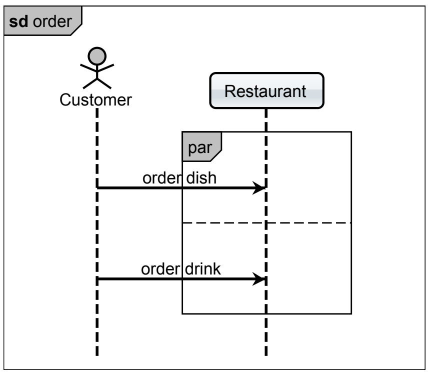
Figure 2.1.: Illustration of the sequence diagram definition.

Figure 1.4 depicts coregions as another sequence diagram element, which cannot directly be recognized in Definition 2.1. Coregions allow expressing that no ordering constraints hold for the covered events. A coregion, however, does not express that the events have to be executed simultaneously. A coregion in UML 2.0 is a notational shorthand for a par fragment that covers one lifeline. Figure 2.2(a) presents a simple sequence diagram with a coregion where a Customer orders dish and drink in the Restaurant. To express that both events may be received in arbitrary order, a coregion encapsulates the receive events. Figure 2.2(b) depicts the same sequence diagram with a par fragment. In the remainder of this thesis, coregions are used in the par fragment representation and, thus, handled as other fragments.

Instances in a sequence diagram interact with each other through messages. Each



(a) Exemplary usage of a coregion.



(b) Representation of the coregion as par fragment.

Figure 2.2.: Representation of coregions.

message consists of a message send event on the senders lifeline and a message receive event on the receivers lifeline. Besides asynchronous messages, there are several other message types, which were already mentioned in Section 1.2. These other message types, however, can be treated as asynchronous messages, because the message type is not important for the refinement concept, *e.g.*, synchronous messages can be replaced by asynchronous request and reply messages.

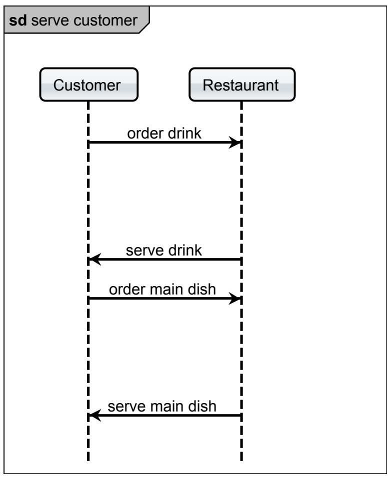
Notation 2.1 (Message):
Let $SD = (I, E, O, \iota, <, \eta, t, \tau)$ be a sequence diagram. A *message* $m \in O_M$ is a pair $(s, r) \in E_{MS} \times E_{MR}$. $!m$ and $?m$ refer to the sending event and receiving event of m, *i.e.*, $!m = s$ and $?m = r$

cf. [21, p. 6].

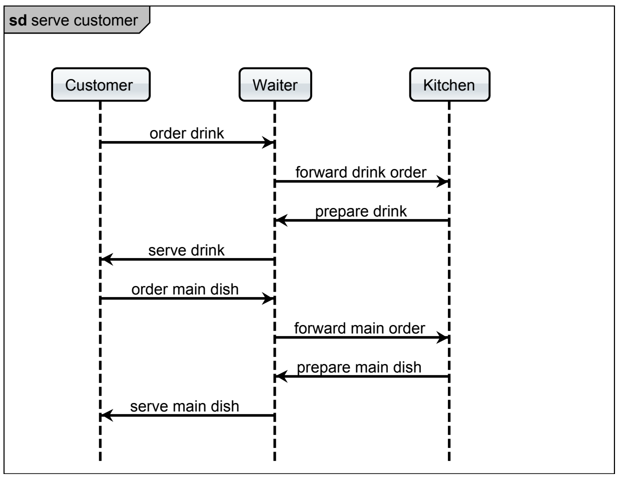
The presented refinement concept distinguishes between two kinds of refinement steps, namely inter-level refinement and intra-level refinement. To introduce these refinement steps, the following examples provide insight into the concepts, which are formally defined in Chapter 3. Let s and t be sequence diagrams; if s is an inter-level refinement of t, t is called s's *master diagram*. In a similar manner, if s is an intra-level refinement of t, t is called s's *parent diagram*.

Inter-level Refinement

Inter-level refinement introduces a more detailed view of a scenario to describe the interaction on a lower level of abstraction. More specific instances replace current abstract ones. Figure 2.3(a) depicts the high-level scenario of serving a customer, who firstly orders a drink and secondly orders the main dish. The Restaurant handles both requests by returning the ordered drink and main dish, respectively. This scenario is inter-level refined by the sequence diagram shown in Figure 2.3(b). The



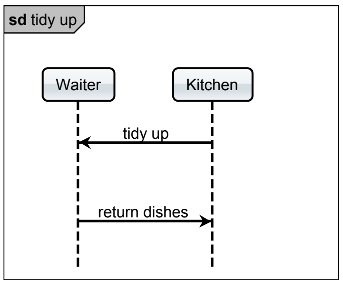
(a) An abstract master diagram.



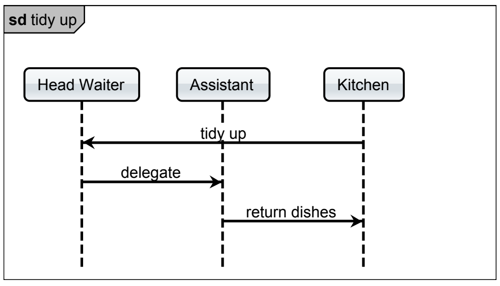
(b) Waiter and Kitchen replace the Restaurant.

Figure 2.3.: An inter-level *refinement* cf. [21, p. 15].

more specific instances Waiter and Kitchen substitute the Restaurant. The Waiter takes on the direct interaction with the Customer, while the Kitchen prepares the drink and the main dish. The new messages in Figure 2.3(b) appear only between the two substituting instances. All messages from the master diagram are also in the refinement. The handling of messages sent or received by the replaced instance, however, has not to be done from the same refined instance. Figure 2.4 shows another valid inter-level refinement. The Head Waiter and the Assistant in Figure 2.4(b) replace the Waiter from the master diagram in Figure 2.4(a). In the master diagram, the Waiter receives the message tidy up and sends the message return dishes. In the refinement, the Head Waiter receives the tidy up instruction and delegates it to the Assistant, who returns the dishes. Thus, the events from the Waiter are separated into the refined instances.



(a) The Kitchen instructs the Waiter to tidy up.



(b) Head Waiter and Assistant substitute the Kitchen.

Figure 2.4.: An inter-level *refinement* with distributed events cf. [21, p. 21].

Intra-level Refinement

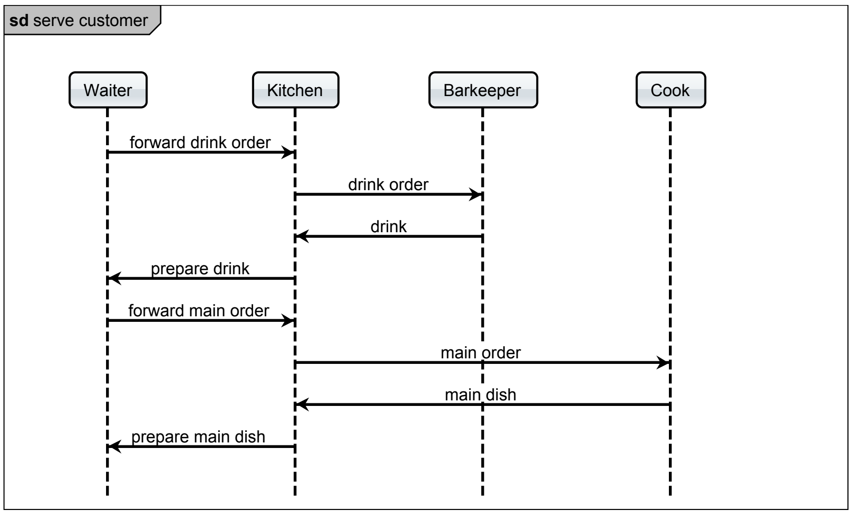


Figure 2.5.: Intra-level refinement of Figure 2.3(b) cf. [21, p. 16].

Intra-level refinement uncovers the internal behavior of existing instances without changing the level of abstraction. Revisiting the example from Figure 2.3(b), Figure 2.5 depicts an intra-level refined sequence diagram. Two new instances were added to the diagram, namely Barkeeper and Cook, which are subinstances of the Kitchen. The Barkeeper takes on the drink order and the Cook prepares the main dish. The interaction as seen by the Waiter is still the same. Furthermore, the Waiter cannot send messages to the Barkeeper or the Cook directly, since they are internals of the Kitchen. Thus, the Kitchen has to forward messages for their subinstances. Besides the message exchange, refinement can be used to reduce the diagram's set of instances to the ones, which are important for the meaning of this scenario. For example the Customer in Figure 2.5 is absent.

Instance Hierarchy

The different refinement steps structure the set of instances, because inter-level refinement replaces instances and changes the level of abstraction, while intra-level refinement adds subinstances to the set of instances. To restrict the refinement possibilities this structure is defined independently.

Definition 2.2 (Instance Hierarchy):
An *instance hierarchy* is a tuple

$$h = (I^0, I^1, \dots, I^n, \mu, \pi)$$

where

- $I^0, I^1, \ldots, I^n$ are pairwise disjoint sets of instances representing the different abstraction layers. I denotes the union of these sets.
- $\mu : I \to I \cup \{\bot\}$ maps *refining* instances to their *master* instance, *i.e.*,

$$\forall\ i \in I : \mu(i) \neq \bot \Rightarrow \exists\ k \in \{1, \ldots, n\} :\ i \in I^k \land \mu(i) \in I^{k-1}.$$

- $\pi : I \to I \cup \{\bot\}$ is an acyclic function that maps *child* instances to their *parent* instance, *i.e.*,

$$\forall\ i \in I : \pi(i) \neq \bot \Rightarrow \exists\ k \in \{0, \ldots, n\} :\ i \in I^k \land \mu(i) \in I^k,$$

and μ and π fulfill the following restrictions:

$$\forall\ i \in I \setminus I^0 : \mu(i) = \bot \Rightarrow \pi(i) \neq \bot, \text{ and}$$
$$\forall\ i \in I \setminus I^0 : \pi(i) \neq \bot \neq \mu(i) \Rightarrow \mu(\pi(i)) = \pi(\mu(i)) \neq \bot$$

cf. [21, p. 17].

Figure 2.6 depicts the instance hierarchy used throughout this section. The vertical arrows point to the master instance of an inter-level refined instance and the horizontal arrows point to the parent instance of an intra-level refined instance.

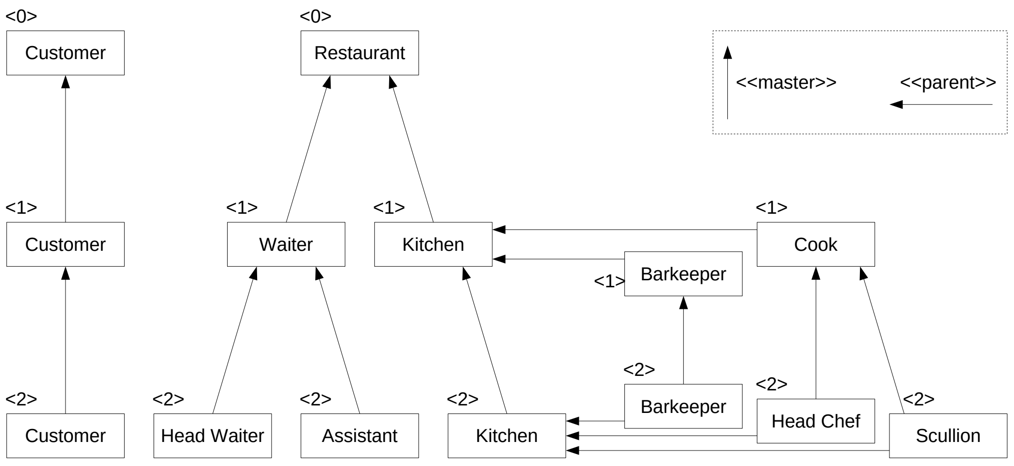


Figure 2.6.: An instance hierarchy with three different levels of abstraction cf. [21, p. 17].

The following definitions restrict the set of valid sequence diagrams according to the instance hierarchy. Due to the structure of capsules, messages are only valid between children of the same parent. Communication with subinstances of a parent from outside the parent can be done by communicating with the parent, which forwards the message to the corresponding child. Thus a message between the Barkeeper and the Cook in Figure 2.6 is valid, while messages between Waiter and Cook are invalid. The following definition describes in detail which message are illegal.

Definition 2.3 (Illegal Message):
Let sd be a sequence diagram and $p : I \times O \to I$ with $\forall\ i \in I, o \in O$: $p(i,o) = i$ a projection to the first tuple element. A message $m \in O_M$ is *illegal* if the sending and receiving instance of m have different parent instances, and neither is the sending instance the parent of the receiving instance nor is receiving instance the parent of the sending instance. The set $\mathrm{ILL}(sd)$ refers to the *illegal messages* of sd, *i.e.*,

$$\begin{aligned}\mathrm{ILL}(sd) = \{m \in O_M \mid\ & \pi(p(\iota(!m))) \neq \pi(p(\iota(?m))) \\ & \wedge\ \pi(p(\iota(!m))) \neq p(\iota(?m)) \\ & \wedge\ \pi(p(\iota(?m))) \neq p(\iota(!m))\}\end{aligned}$$

cf. [21, p. 18].

Knowing the set of illegal messages for a sequence diagram, the following definition presents when sequence diagrams are well-formed according to the instance hierarchy.

Definition 2.4 (Well-Formed Sequence Diagram):
Let sd be a sequence diagram and I_{sd} the set of instances of sd. The sequence diagram sd is *well-formed*, if it contains no illegal message and if all instances have the same level of abstraction, *i.e.*,

$$\mathrm{ILL}(sd) = \varnothing\ \wedge\ \exists\ k \in \{0,\dots,n\} : I_{sd} \subseteq I^k$$

cf. [21, p. 18].

2.2. Level State Machine

Besides a consistent refinement, the *completeness* is an equally important aspect that arises during the development process: Have scenarios been specified that have no meaningful continuation in the remaining specification? If this is true, the whole system is under-specified and this can lead to unintended or conflicting behavior. Lischke has defined a consistency notion for sequence diagrams [17], which addresses, among others, the completeness of one specification level. For this purpose Lischke annotated lifelines with state invariants to express the start and end state of a scenario for each participating instance. Figure 2.7 depicts a simple sequence diagram annotated according to the described consistency notion. The Service is initially in the state service closed and the Kitchen in the state kitchen closed, respectively. These states are the start states of this interaction, because the components remain in these states until the Owner initiates the successive communication. Thus, the states are a precondition for the interaction. After the communication is done, both components remain in a new state, namely service open for the Service and kitchen open for the Kitchen. The actor Owner, however, has no state invariants because an actor is an external component to the shown interaction, for which, *a priori*, no state information is known.

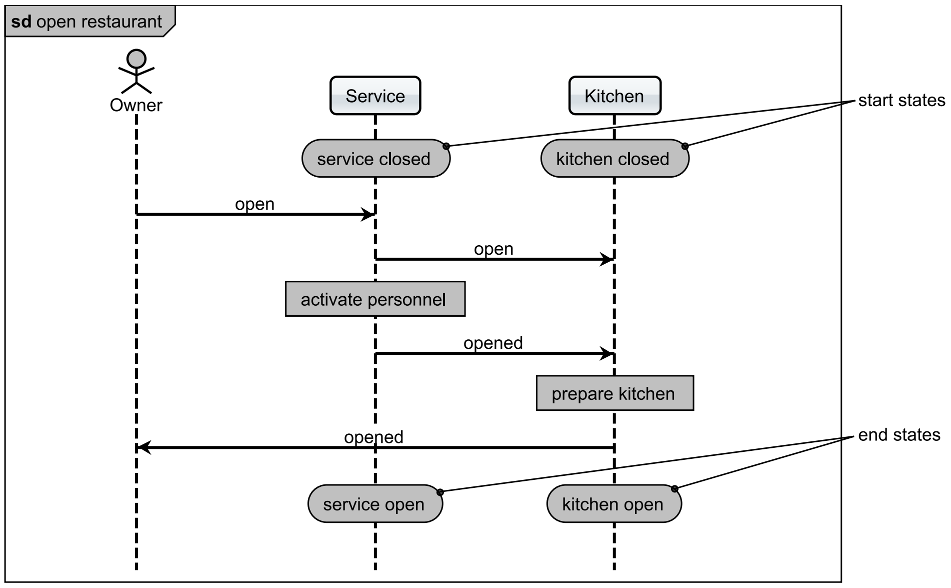


Figure 2.7.: Sequence diagram annotated with state invariants.

All start states of a sequence diagram are combined to a state vector. The same is done for all end states. As a result every sequence diagram can be understood as a transition from its start state vector to its end state vector. Lischke's idea is to take a state machine where each of these state vectors forms one state and every specified sequence diagram forms a transition between those two states. Afterwards, an initial state is marked, which represents the state vector the system initially has, and end states are characterized, which represent state vectors the system can terminate in. This leads to a state machine representation of all sequence diagrams in one level of abstraction, originally called *System State Machine*.
This thesis uses the term *Level State Machine*, because of the many different levels in the software development process, while *System State Machine* suggests that there exists only one state machine for all levels.
Most importantly the level state machine gives information about the completeness of the corresponding level. If there are states that are not reachable from the initial state, the represented scenarios will never occur. Furthermore, if there are states with no outgoing transitions that are not marked as end states, the corresponding scenarios will be reachable, but are no desired termination points for the whole system. These two cases can be automatically determined, but there are even more discrepancies a developer can recognize, *e.g.*, some scenarios should be applicable in a loop, but there is no cyclic dependency in the level state machine. Another discrepancy may be that the level state machine has an unexpected path between two state vectors, *e.g.*, a path to an error state without informing somebody about the failure.

2.2.1. Example

The following example shows a simple scenario and the corresponding level state machine to illustrate the concept. The scenario includes two components: service and kitchen. Each component has several states, namely open, closed, and closure for both components and additional wait for the service and pause for the kitchen. Furthermore, the level state machine needs the necessary sequence diagrams to describe the state transitions. The details of the actual interaction that takes place between the start and end states are not important for this example, thus, the following condensed representation is sufficient to describe one sequence diagram.

sequence diagram label ⟨*start states*⟩ → ⟨*end states*⟩

- start states: tuple of participating components and their start states (*e.g.*, service.open, kitchen.open)
- end states: tuple of participating components and their end states (*e.g.*, service.closed, kitchen.closed)

Using this representation, the following sequence diagrams represent the transitions of the level state machine.

SD label	⟨*start states*⟩ → ⟨*end states*⟩ • ⟨*comment*⟩
open restaurant	⟨*service.closed, kitchen.closed*⟩ → ⟨*service.open, kitchen.open*⟩ • the owner opens the restaurant in the morning
close restaurant	⟨*service.open, kitchen.open*⟩ → ⟨*service.closed, kitchen.closed*⟩ • the owner closes the restaurant in the evening
serve customer	⟨*service.open, kitchen.open*⟩ → ⟨*service.wait, kitchen.open*⟩ • service and kitchen serve the customer
take payment	⟨*service.wait, kitchen.open*⟩ → ⟨*service.open, kitchen.open*⟩ • the customer pays the bill and leaves the restaurant
take a rest	⟨*kitchen.open*⟩ → ⟨*kitchen.pause*⟩ • the kitchen's personnel takes a rest
closure of business	⟨*service.open, kitchen.open*⟩ → ⟨*service.closure, kitchen.closure*⟩ • the owner has to go out of business

Finally, the initial state vector and the end state vectors have to be determined by the user. In this example ⟨*service.closed, kitchen.closed*⟩ is the initial state vector, which means that initially the service and the kitchen are closed. In addition this state combination is one of the end state vectors. Furthermore, ⟨*service.closure, kitchen.closure*⟩ is a second end state vector that arises after the closure of business. Figure 2.8 shows the resulting level state machine with the six state vectors and all possible transitions between these state vectors. The initial state vector is decorated with a round, black filled pseudo state and the end states have an unlabeled transition to the final state.

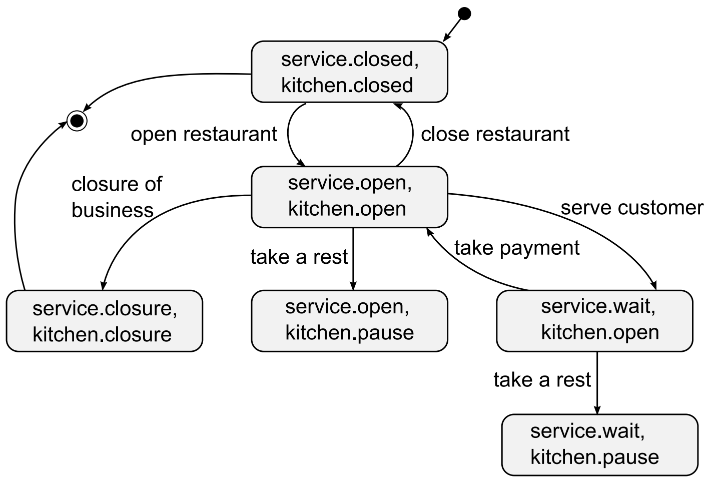

service.closed, kitchen.closed
open restaurant
close restaurant
closure of business
service.open, kitchen.open
serve customer
take a rest
take payment
service.closure, kitchen.closure
service.open, kitchen.pause
service.wait, kitchen.open
take a rest
service.wait, kitchen.pause

Figure 2.8.: Level state machine for the example.

The level state machine has two states without outgoing transitions, which implies that these states represent dead locks of the whole system. There are no scenarios that describe how to leave these states. This problem can be solved by adding a new sequence diagram, which specifies for the kitchen personnel how to end the break. There are, however, several other possibilities to make this example dead lock free, *e.g.*, by adding two different sequence diagrams, one for each problematical state. To sum up, the exemplary scenario used here is not consistent and can lead to unintended behavior if it will be used. The level state machine uncovers these problems and highlights possibilities how to fix the problems.

2.2.2. Petri Nets

Petri nets are a common formalism in computer science and mathematics used for modeling and analyzing distributed and concurrent systems. Since there are detailed publications on Petri nets [20], this section only gives a brief introduction to the formalization of Petri nets that is used throughout this thesis.

Definition 2.5 (Petri Net):
A Petri net (PN) is a tuple

$$PN = (P, T, F, m_0)$$

where

$$\begin{array}{rl} P & \text{is a non-empty set of } \textit{places}, \\ T & \text{is a non-empty set of } \textit{transitions}, \\ F & \subseteq (P \times T) \cup (T \times P) \text{ is a flow relation, and} \\ m_0 & : P \to \{0,1\} \text{ is a binary initial marking} \end{array}$$

The set of places P and the set of transitions T are disjoint. The *pre-set* for a transition t is $\bullet t := \{p \in P \mid (p,t) \in F\}$ and the *post-set* for t is $t\bullet := \{p \in P \mid (t,p) \in F\}$. Analogously, the pre-set for a place p is $\bullet p := \{t \in T \mid (t,p) \in F\}$ and the post-set for p is $p\bullet := \{t \in T \mid (p,t) \in F\}$. A *marking* of a Petri net is a function $m : P \to \{0,1\}$. A place p is called *marked* by m if $m(p) = 1$ and *unmarked* otherwise.

Thus, a Petri net is a directed, bipartite graph separated into transitions as active elements and places as passive elements. Each place is either marked or unmarked. The current marking m of a Petri net *PN* represents the state of *PN* and serves for the definition of an execution model for Petri nets.

Definition 2.6 (Execution of a Petri net):
Let $PN = (P,T,F,m_0)$ be a Petri net with the current marking m. A transition $t \in T$ is *enabled* in m, iff $\forall\ p \in \bullet t :\ m(p) = 1$. To take one execution step of *PN* an arbitrary transition t from $\{t \in T \mid t \text{ is enabled}\}$ is chosen. The process of taking a transition is called *firing*. If a transition is fired, the result will be a new Petri net marking, which is written as $m \xrightarrow{t} m'$ with

$$m'(p) := \begin{cases} 0 & \text{if } p \in \bullet t, \\ 1 & \text{if } p \in t\bullet, \\ m(p) & \text{otherwise.} \end{cases}$$

A Petri net *execution* is a sequence $\langle m_0, t_0, m_1, ... \rangle$ of successive markings and transitions with $m_i \xrightarrow{t_i} m_{i+1}$ for all i.

Besides having a defined mathematical model, Petri nets are well-suited to analyze properties of the modeled system. An important property is the reachability of a given marking, which means whether the state of the Petri net can be reached from the initial marking or not. For example, when modeling a traffic light system it is important that cars and pedestrians do not have both green for the same street at the same time. This state represents a Petri net marking that should not be reachable from the initial marking. Mayr [19] proved that the reachability problem is decidable and presented a first algorithm.

Definition 2.7 (Reachability):
Let $PN = (P,T,F,m_0)$ be a Petri net. A marking m is *reachable* from the initial marking m_0, iff there is a Petri net execution $\langle m_0, t_0, m_1, ..., m_n \rangle$ with $m = m_n$. The set of all possible reachable markings for a Petri net *PN* from its initial marking m_0 is denoted with $R(PN)$.
The reachability graph for a Petri net *PN* is a directed graph $G = (V,E)$ with $V = R(PN)$ and $E = \left\{ (m,t,m') \,\middle|\, m \xrightarrow{t} m' \right\}$.

2.2.3. Sequence Diagram Start / End States

Lischke [17] introduced the system state machine concept for UML 1.4 sequence diagrams. Thus, the concept does not support new elements from UML 2.0. Combined fragments can be used to have more then one start and end state, *e.g.*, the operator break can be used to prematurely terminate a given interaction or an alt fragment can be used to specify an alternative start or end. Figure 2.9 shows the open restaurant example with a break fragment, which leads to two possible end states for the Restaurant instance. On the one hand, if the open restaurant request was successful, the restaurant answers with opened and reaches the state open. On the other hand, when an error occurred while opening the restaurant, an error message is returned and the end state of this interaction remains closed.

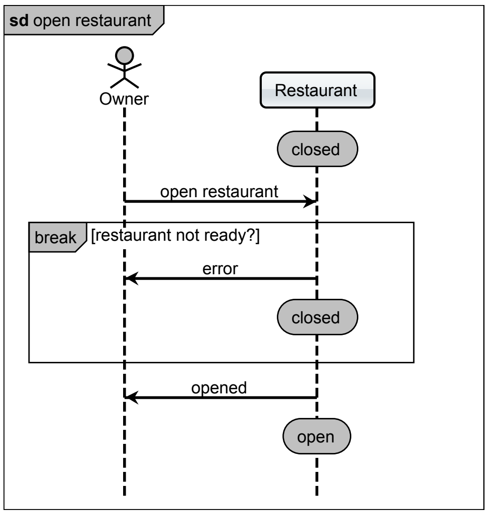


Figure 2.9.: Sequence diagram with two end states.

Consequently, one sequence diagram can represent more than one transition in the level state machine. Therefore it is necessary to determine all possible start and end state vectors of a sequence diagram. Not all states in a sequence diagram are possible start or end states, though. In the following, some possibilities how end states can arise are presented to motivate the subsequently defined consistency term. Every fragment whose execution might be optional or results in an alternative flow can be a source of additional end states. Figure 2.10 depicts an opt fragment at the bottom of the interaction, which can change the end states. Since the execution is optional, the end states have to be above the fragment and at the bottom of the fragment itself. Consequently, the resulting end state vectors are $\langle$*Service.open, Kitchen.open*$\rangle$ and $\langle$*Service.open, Kitchen.ready to cook*$\rangle$. Besides opt and break fragments, an alt fragment is another possibility to exit a sequence diagram with different end states, since the UML standard allows that no operand of an alt fragment is executed if none of the operand constraints evaluate to true. Figure 2.11 shows an interaction with an alt fragment at the bottom, which does not cover all lifelines and, thus, the result is a combination of end states. In the normal course of life for this restaurant, both alternative conditions are false and the Service, just as the Kitchen, terminates the interaction in state open. The alternative traces let the interaction terminate with

the end state vectors ⟨*Service.wait, Kitchen.open*⟩ and ⟨*Service.close, Kitchen.open*⟩, respectively. If one condition of an operand of the alt fragment is else, then there is always an applicable alternative and, thus, no state information has to be above the fragment.
A further fragment that leads to additional end state vectors is the loop fragment, which repeats the included sequence a number of times. Since this number can be zero, the components do not have to execute the included sequence. In conclusion, considering end states loop fragments can be handled as opt fragments. Figure 2.12 depicts the open restaurant example with a loop fragment, which can be executed zero times if there are no waiting reservations or a number of times until all reservations are processed.

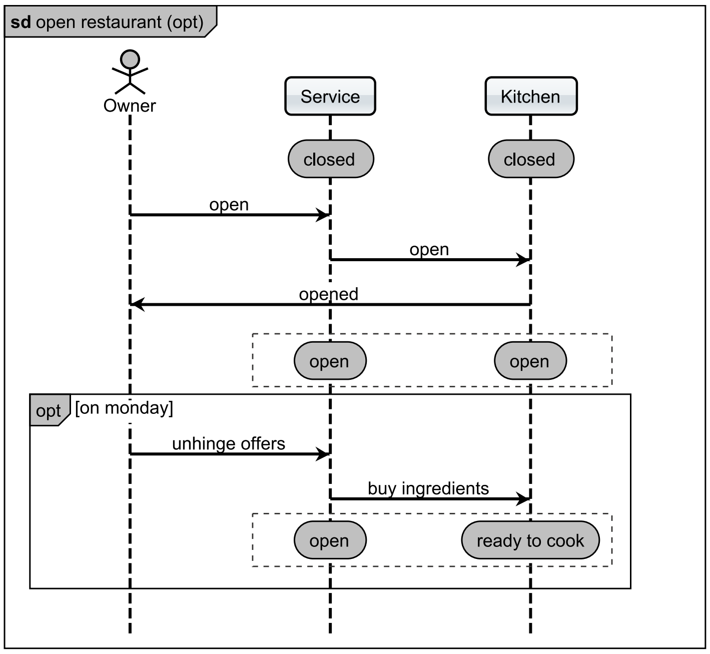


Figure 2.10.: Different end states through an opt fragment.

The following notation defines additional functions on event sets of sequence diagrams, which are necessary to define a consistency notion for start and end states.

Notation 2.2:
Let $SD = (I, E, O, \iota, <, \eta, t, \tau)$ be a sequence diagram.
Then operandEventSet(E, o) contains only events between the operand begin and the operand end event(s) of o, *i.e.*,

$$\begin{aligned}
\text{operandEventSet} &: E \times O_O \to E,\\
\text{operandEventSet}(E, o) &:= \{e \in E \mid \exists\ i \in I, o' \in O : \iota(e) = (i, o')\\
&\qquad \wedge (\exists\ e' \in E_{OB} : \iota(e') = (i, o) \wedge e' < e)\\
&\qquad \wedge (\exists\ e' \in E_{OE} : \iota(e') = (i, o) \wedge e < e')\}.
\end{aligned}$$

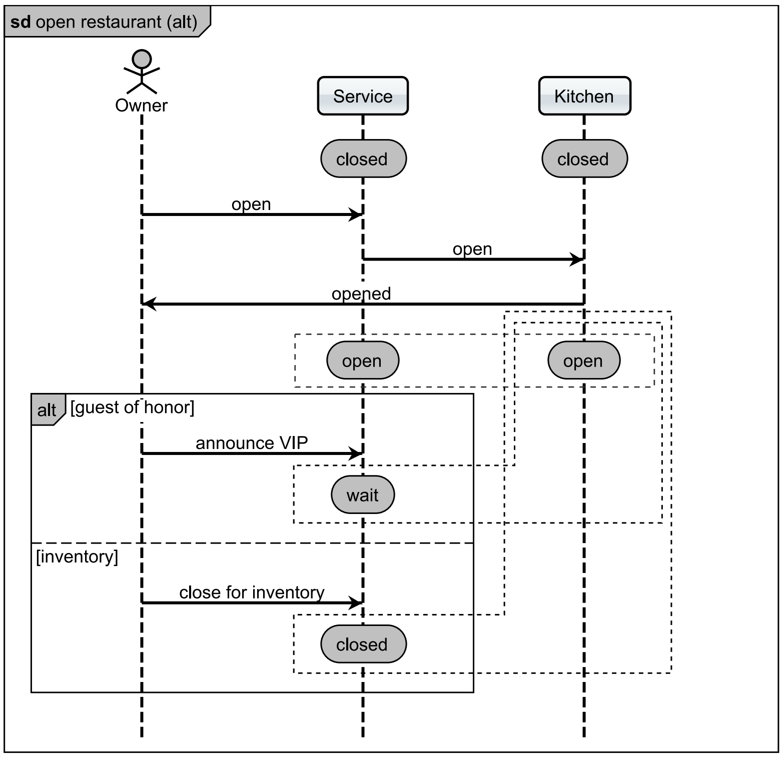


Figure 2.11.: Different end states through an alt fragment.

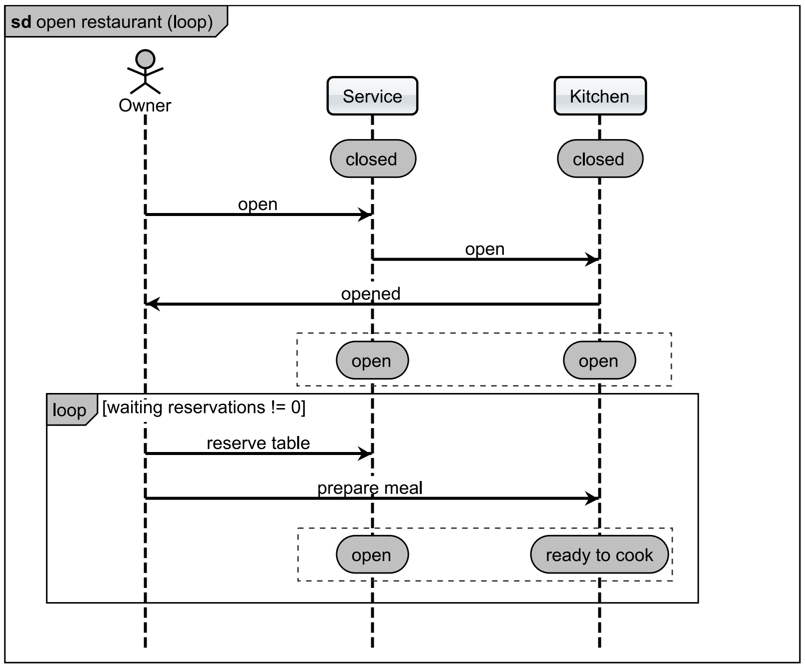


Figure 2.12.: Different end states through a loop fragment.

The cutEventSet(E, f) contains all events from E excluding the fragment f with all events in all operands from f, *i.e.*,

$$\begin{aligned}\text{cutEventSet} : E \times O_F &\to E,\\ \text{cutEventSet}(E, f) :=\ & E \setminus \left\{ \bigcup_{o \in O_O : \eta(o) = f} \text{operandEventSet}(E, o) \right.\\ & \left. \cup \{e \in E_{OB} \cup E_{OE} \mid \exists\ i \in I : \iota(e) = (i, o)\} \right\}.\end{aligned}$$

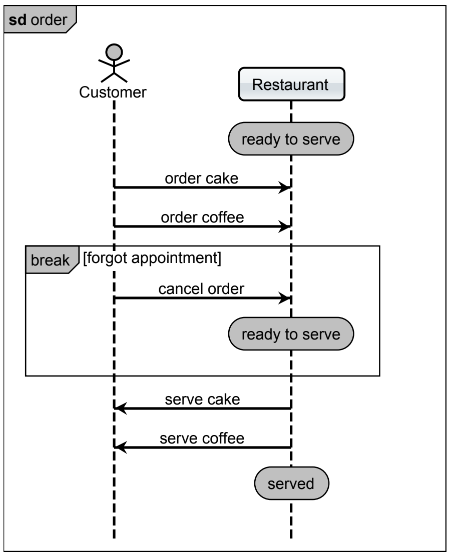
The next event after e according to the event order on the same lifeline is contained in next(e), *i.e.*,

$$\begin{aligned}\text{next} : E &\to E\\ \text{next}(e) :=\ & \{e' \in E \mid \exists\ i \in I, o \in O : \iota(e) = (i, o)\\ & \wedge \exists\ o' \in O : \iota(e') = (i, o') \wedge e' > e\\ & \wedge \forall\ e'' \in E|_i : e'' \geq e' \vee e'' < e\}.\end{aligned}$$

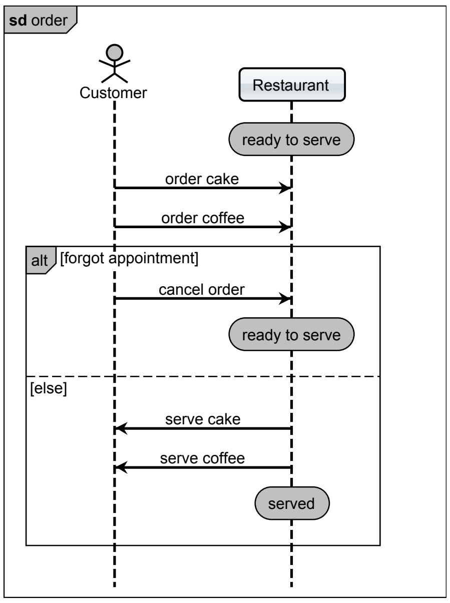
The previous event before e according to the event order on the same lifeline is contained in prev(e), *i.e.*,

$$\begin{aligned}\text{prev} : E &\to E\\ \text{prev}(e) :=\ & \{e' \in E \mid \exists\ i \in I, o \in O : \iota(e) = (i, o)\\ & \wedge \exists\ o' \in O : \iota(e') = (i, o') \wedge e' < e\\ & \wedge \forall\ e'' \in E|_i : e'' \leq e' \vee e'' > e\}.\end{aligned}$$

The following definition describes whether a given sequence diagram is consistent with the start and end state information needed to create the level state machine. To simplify the definition and the following algorithm, an alt fragment replaces any break fragment. This is no semantically correct transformation, but in case of end states every break fragment is analogous to an alt fragment with an else-case. Furthermore, all references have to be expanded while all gates and parameters have to be resolved.
Figure 2.13(a) shows an order sequence during which the Customer orders a cake and a coffee. If the Customer remembers an important appointment, the Customer prematurely cancels the transaction. Figure 2.13(b) shows the same sequence where an alt fragment replaces the break fragment. The interaction below the break fragment forms the second case for the alt fragment. Since the condition for the second operand is else there is always one executable operand. The transformation is, however, not semantically equivalent, since the Customer can notice the forgotten appointment before the Restaurant receives the coffee order. The break fragment prematurely terminates the surrounding interaction and, thus, the coffee order may not be received. In Figure 2.13(b), this case is not possible, since the alt fragment cannot cancel an interaction and the coffee order will always be received. The set of possible end state vectors of both sequence diagrams, however, remains the same.



(a) If the Customer remembers an important appointment, the sequence will be prematurely canceled.



(b) The corresponding diagram with an alt fragment.

Figure 2.13.: Replace a break fragment with an alt fragment.

Definition 2.8 (Sequence Diagram State Consistency):
A sequence diagram $SD = (I, E, O, \iota, <, \eta, t, \tau)$ is *state consistent*, if start and end states are present for each component's lifeline, *i.e.*

$$\forall\ i \in I_c : \text{checkStartStates}\left(E|_i\right) \wedge \text{checkEndStates}\left(E|_i\right).$$

The start (end) states are present for lifeline i if the first (last) element of the total order for i is a state or an alt, opt, or loop fragment, which contains start (end) states, *i.e.*

$$\begin{aligned}\text{checkStartStates}(E) \coloneqq\ & (\min(E) \in E_s) \\ & \vee \text{checkStartFragments}(E),\end{aligned}$$

$$\begin{aligned}\text{checkEndStates}(E) \coloneqq\ & (\max(E) \in E_s) \\ & \vee \text{checkEndFragments}(E).\end{aligned}$$

The maximum (max) and minimum (min) are applicable since the event set is restricted to the total order of one lifeline. If the first (last) element of the total order of i is an alt, opt, or loop fragment, each operand of the fragment has to contain start (end) states. For the fragments that introduce optional sequences, *i.e.*, opt, loop, and alt without an else-case, start (end) states have to exist after (before) the fragment, *i.e.*

$$\begin{aligned}\text{checkStartFragments}(E) :=\ & \exists\ i' \in I, o \in O_O : \iota(\min(E)) = (i', o)\\ & \wedge t(\eta(o)) \in \{\mathsf{alt}, \mathsf{opt}, \mathsf{loop}\}\\ & \wedge ((t(\eta(o)) = \{\mathsf{alt}\} \wedge\\ & \quad \forall\ o' \in O_O : \eta(o') = \eta(o) \Rightarrow \tau(o') \neq \text{"}else\text{"})\\ & \quad \vee t(\eta(o)) \in \{\mathsf{opt}, \mathsf{loop}\} \Rightarrow\\ & \quad \text{checkStartStates}(\text{cutEventSet}(E, \eta(o))))\\ & \wedge \forall\ o' \in O_O : \eta(o') = \eta(o) \Rightarrow\\ & \quad \text{checkStartStates}(\text{operandEventSet}(E, o')),\end{aligned}$$

$$\begin{aligned}\text{checkEndFragments}(E) :=\ & \exists\ i' \in I, o \in O_O : \iota(\max(E)) = (i', o)\\ & \wedge t(\eta(o)) \in \{\mathsf{alt}, \mathsf{opt}, \mathsf{loop}\}\\ & \wedge ((t(\eta(o)) = \{\mathsf{alt}\} \wedge\\ & \quad \forall\ o' \in O_O : \eta(o') = \eta(o) \Rightarrow \tau(o') \neq \text{"}else\text{"})\\ & \quad \vee t(\eta(o)) \in \{\mathsf{opt}, \mathsf{loop}\} \Rightarrow\\ & \quad \text{checkEndStates}(\text{cutEventSet}(E, \eta(o))))\\ & \wedge \forall\ o' \in O_O : \eta(o') = \eta(o) \Rightarrow\\ & \quad \text{checkEndStates}(\text{operandEventSet}(E, o')).\end{aligned}$$

Finally, all possible start and end state vectors have to be calculated. This task is separated in three steps:

- Algorithm 2.1 firstly determines all start states recursively for each lifeline according to the consistency definition. The algorithm to determine the end states is analogous to Algorithm 2.1 when changing min to max.
- Secondly, Algorithm 2.2 creates a Petri net with all start (end) states and their relations, *i.e.*, common operands.
- Finally, this Petri net is used to calculate the resulting state vectors the sequence diagram can start or end with, respectively. This step calculates the reachability graph for the created Petri net. Due to the structure of the Petri net, the states of the reachability graph are valid start (end) state vectors for the sequence diagram.

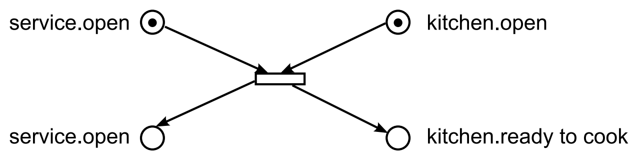
The three steps are performed once for the start states and once for the end states of the sequence diagram. Figure 2.14 depicts the Petri net representations for the end states of Figure 2.10 and Figure 2.11, respectively.

Algorithm 2.1 Calculate all start (end) states of a sequence diagram
Inputs: $sd = (I, E, O, \iota, <, \eta, t, \tau)$

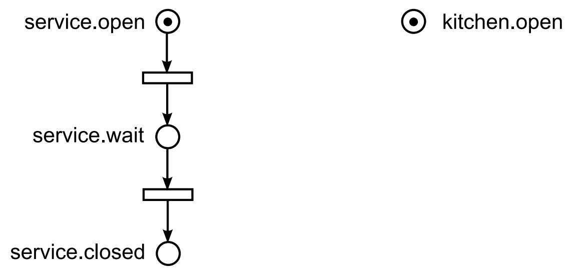
```
function calcStartStates($E$)
    curStates ← {}
    $(i, o) \leftarrow \iota(\min(E))$
    if $\min(E) \in E_S$ then
        curStates ← $\{\min(E)\}$
    else if $\min(E) \in E_{OB} \cup E_{OE}$ then
        elseOperand ← false
        for all $o' \in O_O$ do
            if $\eta(o') = \eta(o)$ then
                curStates ← curStates ∪ calcStartStates(operandEventSet($E, o'$))
                if $\tau(o')$ ="else" then
                    elseOperand ← true
                end if
            end if
        end for
        if elseOperand = false $\vee$ $t(\eta(o)) \in \{\text{loop}, \text{opt}\}$ then
            curStates ← curStates ∪ calcStartStates(cutEventSet($E, \eta(o)$))
        end if
    end if
    return curStates
end function

startStates ← {}
for all $i \in I_c$ do
    startStates ← startStates ∪ calcStartStates($E|_i$)
end for
return startStates
```

Algorithm 2.2 Create Petri net to determine start (end) state combinations

```
Require: state contains the start [end] state events according to Algorithm 2.1
  $P \leftarrow \{\}$
  $T \leftarrow \{\}$
  $F \leftarrow \{\}$
  $m_0 \leftarrow \{(x,1) \mid x \in \text{state}\}$
  for all $i \in I_c$ do
    for all $s \in \text{state}|_i$ do
      $P \leftarrow P \cup \{s\}$
      $e \leftarrow s$
      while $next(e) \neq \varnothing \ \wedge \ next(e) \notin \text{state}$ do
        $\{e\} \leftarrow next(e)$
        if $e \in E_{OE}$ [$e \in E_{OB}$] then
          $(i,o) \leftarrow \iota(e)$
          $T \leftarrow T \cup \{o\}$
          $F \leftarrow F \cup \{(s,o)\}$
        end if
      end while
      if $next(e) \neq \varnothing$ then
        $F \leftarrow F \cup \{(o, next(e))\}$
      end if
      for all $s' \in \text{state}|_i$ do
        $\{e'\} \leftarrow \{e'' \in E_s \mid \iota(e'') = (i,s)\}$
        if $e' < e$ then
          $m_0(s) \leftarrow 0$
        end if
      end for
    end for
  end for
  return $(P,T,F,m_0)$
```



(a) Petri net representation of the end states of Figure 2.10.



(b) Petri net representation of the end states of Figure 2.11.

Figure 2.14.: Petri net representations.

2.2.4. Petri Net Representation for the Level State Machine

Lischke [17] used an algorithm for the generation of the level state machine that matches the sequence diagram formalization used by her. By using Petri nets, the same results are achieved with established and well analyzed methods. Thus, the level state machine is a result of a common Petri net algorithm, if the following Petri net representation is used. Figure 2.15 depicts the Petri net representation to generate the level state machine from the example of this section (cf. Figure 2.8). Each place represents a state of one component in the given level of abstraction and each transition represents one sequence diagram from one start state combination to one end state combination. In this example each sequence diagram corresponds to only one transition, because each of them has exactly one start state and end state combination. The states in the initial state vector of the level state machine description are initially marked in the Petri net. The reachability graph of this representation equals the level state machine, because each reachable marking of the Petri net is equivalent to one state vector in the level state machine. Consequently the edges in the reachability graph are annotated with the labels of the transitions to obtain the same presentation.
Algorithm 2.3 converts a set of sequence diagrams for a given refinement level to the introduced Petri net representation.

2.3. State Refinement

Section 2.1 introduced the inter- and intra-level refinement for instances and the impact on the presence of messages in related sequence diagrams. Section 2.2 presented the level state machine concept, which reveals how complete a specification level is,

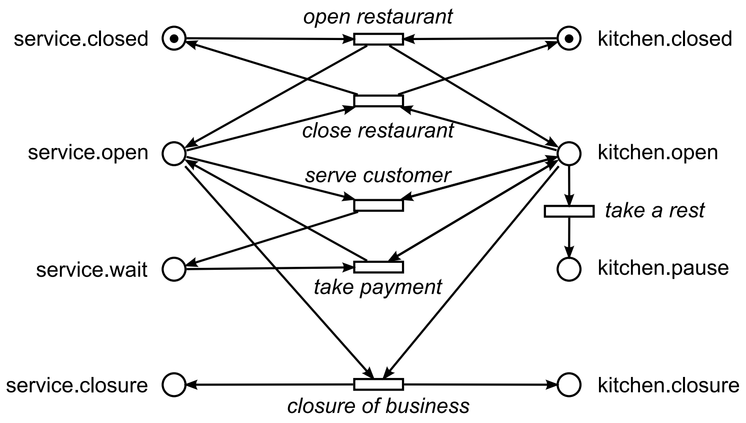


Figure 2.15.: Petri net representation of Figure 2.8.

Algorithm 2.3 lsmTransformation
Inputs: n, initialStates

```
$P \leftarrow \{\}$
$T \leftarrow \{\}$
$F \leftarrow \{\}$
for all $sd \in SD^n$ with $sd = (I, E, O, \iota, <, \eta, t, \tau)$ do
  startVectors $\leftarrow$ calcStartVectors($E$)
  endVectors $\leftarrow$ calcEndVectors($E$)
  for all startVector $\in$ startVectors do
    for all endVector $\in$ endVectors do
      $T \leftarrow T \cup sd^{\text{startVector}}_{\text{endVector}}$
      for all state $\in$ startVector do
        $P \leftarrow P \cup \{\text{state}\}$
        $F \leftarrow F \cup \{(\text{state}, sd^{\text{startVector}}_{\text{endVector}})\}$
      end for
      for all state $\in$ endVector do
        $P \leftarrow P \cup \{\text{state}\}$
        $F \leftarrow F \cup \{(sd^{\text{startVector}}_{\text{endVector}}, \text{state})\}$
      end for
    end for
  end for
end for
$m_0 \leftarrow \{(x,1)) \mid x \in \text{initialStates}\} \cup \{(x,0) \mid x \in P \setminus \text{initialStates}\}$
return $(P, T, F, m_0)$
```

by using sequence diagrams that are annotated with state information. These two concepts are independent, because the instance refinement does not include state information. Thus, a refinement concept for states in sequence diagrams is needed to combine instance refinement and level state machines.
States of different components are only weakly related, even if they are in a refinement relation, such as described in Section 2.1. Figure 2.16 depicts a small example to illustrate some cases for the state refinement. The master instance Restaurant is refined to the Waiter and the Kitchen. Each instance has several states, which are listed below the instance.

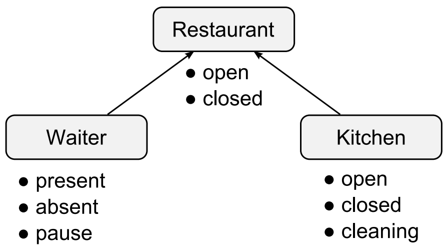


Figure 2.16.: Motivating example for the state refinement.

A one-to-one match of states between master and refining instance is not possible, since the Restaurant has two states and the refining instances have three states. The Waiter does not even have a state with the same name as the Restaurant. A relation between the states, however, exists. The open state of the Restaurant relates to the open state of the Kitchen and the present state of the Waiter. Likewise, the closed state of the Restaurant relates with the closed state of the Kitchen and the absent state of the Waiter. The open state of the Restaurant also relates to the additional states of the Waiter and the Kitchen, namely pause and cleaning. Nonetheless, the Kitchen can be closed while the Restaurant is open or the Kitchen can be cleaning while the Restaurant is closed. Thus, the relations between states are weak, but they exist and should be made explicit in the model. All this is captured in the following definition.

Definition 2.9 (State Relation):
Let $h = (I^0, I^1, \ldots, I^n, \mu, \pi)$ be an instance hierarchy. S_i designates the states of instance i and S refers to the set of all states for all instances, *i.e.*, $S := \bigcup_{i \in I} S_i$. The relation

$$\mathrm{SR} \subseteq S \times S$$

named *State Relation* holds all relations between states of subsequent instances according to the refinement.

According to this definition, the state relation for the example from Figure 2.16 can be:

$$\begin{aligned} SR = \{&(\text{Restaurant.open}, \text{Kitchen.open}), (\text{Restaurant.open}, \text{Kitchen.cleaning}),\\ &(\text{Restaurant.open}, \text{Waiter.present}), (\text{Restaurant.open}, \text{Waiter.pause}),\\ &(\text{Restaurant.closed}, \text{Kitchen.closed}), (\text{Restaurant.closed}, \text{Waiter.absent})\} \end{aligned}$$

On the one hand, every state of the master instance needs a relation to at least one state of each refined instance, because being in one state has to have a counterpart in every refined instance. On the other hand, each state of a refined instance needs a relation to at least one master state, because each state combination of the refined instances has to reflect one master state.
The following definition formalizes these conditions.

Definition 2.10 (State Consistency):
Let $h = (I^0, I^1, \dots, I^n, \mu, \pi)$ be a instance hierarchy and SR the state relation for h. SR is *state consistent*, iff

$$\begin{aligned} \forall\ i \in I\ \exists\ i' \in I : \mu(i) = i' \Rightarrow &(\forall\ s \in S_{i'}\ \exists\ s' \in S_i : (s, s') \in \text{SR})\\ \wedge\ &(\forall\ s \in S_i\ \exists\ s' \in S_{i'} : (s', s) \in \text{SR}). \end{aligned}$$

2.4. Message Refinement

During the development process messages can change their names and other details such as their documentation or communication ports. The reason for that is that messages are refined from an abstract information transport- and synchronization-medium to concrete protocols and bus systems. Thus, messages in the requirement levels can have other names than in the implementation levels, since they are refined to physical entities. Hence, messages between different levels of abstraction are not the same object, but they are directly related. The following definition presents an one-to-one refinement for messages

Definition 2.11 (Message Refinement):
Let $n, m \in O_M$ be messages and $G : O_M \to O_M$. The partial function G relates messages with their refinements, and if message n refines message m, there is no other message n' which also refines message m, *i.e.*,

$$\begin{aligned} (m, n) \in G :\Leftrightarrow\ &\text{message } n \text{ refines message } m \text{ and}\\ &\text{for all } n' \in O_M : (m, n') \in G \Rightarrow n = n'. \end{aligned}$$

3. Model/View Concept

This chapter presents the constructive model/view concept, which incorporates the refinement and consistency notions from Chapter 2. Section 3.1 introduces the concept informally and describes the benefits for the software development process. Section 3.2 describes the underlying data model and Section 3.3 presents operations on the model.

3.1. Introduction

During the development process many sequence diagrams are created and several of them might show parts of the same behavior from different levels of abstraction or from different points of view. Consequently, there is a large share in redundant information, *e.g.*, the same message in a master diagram and in its refinement. If an element in one diagram is changed, the modification has to be repeated in all diagrams that also refer to that element. The model/view concept abstracts from this issue by decoupling the data elements from their concrete appearance in the diagrams. For structural UML diagrams such as class diagrams, a model/view concept is already used and implemented in commercial CASE tools, *e.g.*, the Rational Systems Developer [12] or ARTiSAN Studio [1]. By contrast, the model/view support for behavioral diagrams such as sequence diagrams is very limited and does not include refinement. The model/view concept is based on the Model-View-Controller pattern.

Model-View-Controller *Model-View-Controller* (*MVC*) is an architectural design pattern, which was firstly described in the context of the object-oriented language *Smalltalk* [15]. It separates an application into the domain specific data representation (Model), the user interface (View), and the control logic (Controller). As a result of this separation, each part consists only of the necessary information and logic. The resulting classes are easier to maintain and reuse due to the reduced amount of dependencies. Figure 3.1 depicts the design pattern and the relations between the different parts. The view visualizes a part from the model, so there is a connection from the view to the model. The controller handles the interaction between view and model, which is shown by the remaining two connections.

A slight modification of this pattern is the model/view pattern that distinguishes only between model and view. The controller is part of the view for handling the user interaction and part of the model for the model interactions.
Section 3.2 describes the realization of this pattern in detail, which consists of all

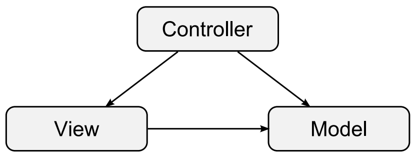


Figure 3.1.: Model-View-Controller design pattern

relevant information for the sequence diagrams, the refinement concept, and the level state machine. Due to the software development process mentioned in Section 1.1, the main views are sequence diagrams, but there are several other important views onto the model. Figure 3.2 depicts a collection of different views and their relation to the model. Each arrow represents the possible information flow between the adjacent views and the model.

Sequence Diagram Sequence diagrams are the main views to add new scenarios and to modify or delete existing scenarios.

Level State Machine Each level of abstraction corresponds to a level state machine (cf. Section 2.2). It shows all reachable state combinations and all scenarios between them. The level state machine view allows to create templates for new sequence diagrams between state combinations. These templates consist of the necessary components, start states, and end states.

Instance Hierarchy The instance hierarchy (cf. Definition 2.2) establishes the basement of the refinement concept. New instances can be added through this view and existing instances can be refined or removed.

State Refinement The state refinement view depicts the current state relation (cf. Section 2.3) and allows the user to add new states to the model, relate states to each other, or remove existing states.

Derived Views Derived views visualize a part from the model in a different way, *e.g.*, default scenarios without optional behavior or scenarios with expanded references. These views are well-suited to communicate with different stakeholders or to get a better understanding of the scenario. Derived views in Figure 3.2 have only a connection from the model to the view, because derived views may use aggregated values. Changes in those views maybe hard or even impossible to integrate into the model and, thus, derived views are read-only.

Several benefits arise from the constructive model/view concept: The number of integrations against a model is lower than the required number of integrations if all diagrams are independent. The constructive approach integrates each change to all parts of the model and resolves all ambiguities. Without the constructive approach, several views may be changed at the same time. Consequently, all views have to be integrated with each other. If n diagrams show partly the same behavior, there are

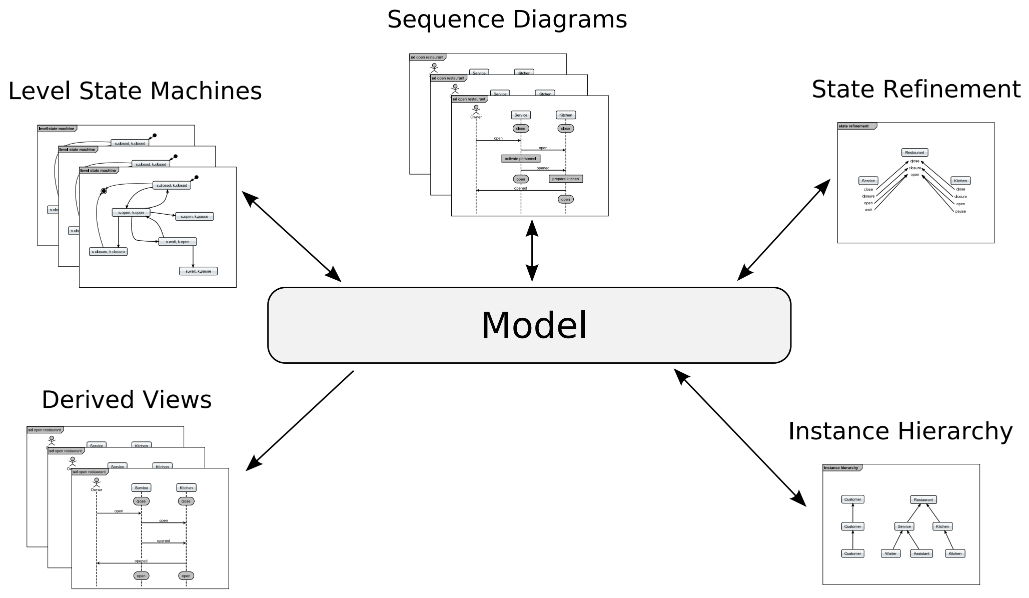
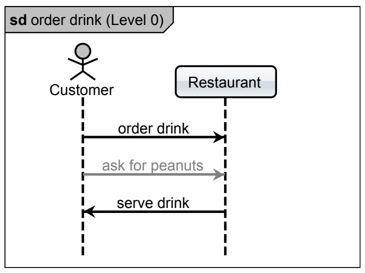


Figure 3.2.: Different views depict partial model information.

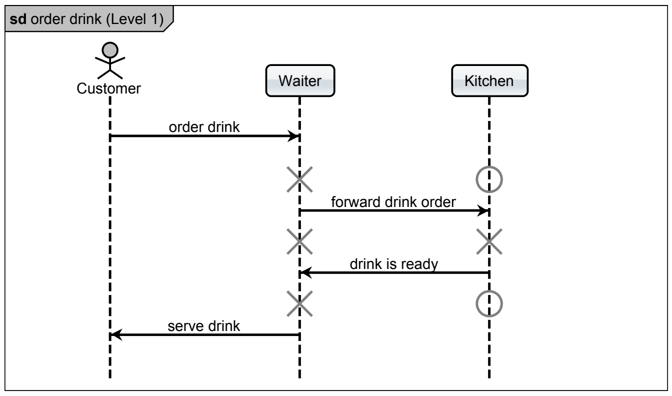
$n * (n-1)/2$ integrations to get a consistent set of diagrams (cf. [5]). Furthermore, the constructive approach allows maintaining model properties, *e.g.*, there will be no state that is not consistent with the state refinement concept in Section 2.3.
Redundancy is reduced to a minimum, since diagrams showing the same behavior are different views on the same model objects. Thus, trivial changes, such as renamed messages, affect directly all diagrams showing the modified object. The central data base of all diagram elements allows fast derivation of new information, such as derived views, aggregated values, or statistics. We refer to this as *model queries* as an analogy to database queries.

Integration of view modifications Changes cannot be integrated automatically into other levels of abstraction, since several different integrations might be valid. Thus, only the developer can decide which integration is desired in such a situation. Figure 3.3(a) depicts a simple example. The Customer orders a drink, which is served by the Restaurant. A new message, namely ask for peanuts, is inserted between the already existing messages. This change has to be integrated with the inter-level refinement, which is shown in Figure 3.3(b). The Waiter and the Kitchen replace the Restaurant. The Waiter communicates with the Customer, and the Kitchen mixes the drink. The position of the send event of the new message is the same as in the source diagram, because the adjacent events, *i.e.*, the send event for order drink and the receive event for serve drink, are directly related. By contrast, there are six possibilities for the receive event of the message ask for peanuts, which are emphasized in Figure 3.3(b). Waiter and Kitchen are both refining the Restaurant and, thus, may both receive the new message. Due to the internal communication between

Waiter and Kitchen, there are three insertion points on both lifelines. Some of the insertion points are, however, of lower quality of integration, since the topmost and bottommost points of the Kitchen's lifeline are not consistent with all traces of the high-level diagram, see also Section 3.2.2. For example, if the receive event of the message ask for peanuts will be inserted as the first event on the Kitchen's lifeline, then there is a trace in which the receive event can occur before the receive event of order drink. This situation cannot occur in the high-level diagram, because the two receive events are ordered there. Thus, the valid inter-level insertion points have to be classified according to the quality of integration.



(a) Insertion of a new message ask for peanuts into the sequence diagram.



(b) Valid insertion points for the receive event of ask for peanuts.

Figure 3.3.: Integration of a new message to the next refinement level.

3.2. Model

The model constitutes the essential part of the concept, because it has to reflect all concepts of Chapter 2. Consequently, the model consists of a data representation and a set of operations to encapsulate this data representation. The operations integrate changes from a view into the model and, thus, these operations have to maintain the consistency of the model. The following definition introduces the data model as the foundation for the model operations and the views.

Definition 3.1 (Data Model):
A *data model* is a tuple

$$M = (h, O, S, G, \sigma, \eta, t, \tau, \text{Seqs})$$

where

h	$= (I^0, I^1, \dots, I^n, \mu, \pi)$ is an instance hierarchy with $I = \bigcup_{k \in \mathbb{N}_{\leq n}} I^k$,
O	is a set of sequence diagram objects,
S	is a state relation,
G	is a message refinement relation,
σ	$: O_S \to I$ maps states to instances,
η	$: O_O \to O_F$ maps operands to fragments,
t	$: O_F \to \{\text{alt}, \text{break}, ...\}$ determines fragment type,
τ	$: O_O \to \Sigma$ assigns a constraint in an alphabet to an operand, and
Seqs	is a set of sequences (cf. Definition 3.2).

As part of the data model, the following definition presents a *sequence*, which is a scenario or a set of scenarios in different levels of abstraction. The term sequence might be misleading, since a sequence, as used here, includes more than one trace. Moreover, it represents a set of traces and also the refinement of these traces. The name sequence is, however, appropriate, because it reflects the UML understanding of a sequence diagram with extensions that reflect the refinement principles.

Definition 3.2 (Sequence):
A *sequence* is a tuple

$$\text{Seq} = (E = E^0 \uplus \dots \uplus E^n, \iota, f)$$

where

E	$= E^0 \uplus \dots \uplus E^n$ is a set of events,
ι	$: E \to I \times O$ maps events to lifelines and objects, and
f	$: \mathbb{N}_{\leq n} \to \mathcal{P}(I) \times \mathcal{P}(E \times E)$
	maps levels of abstraction to a set of instances and an event order.

A sequence consists of events, which are each assigned to a level of abstraction. The function f assigns each level of abstraction to a set of instances from the data model and an event order on the corresponding events. The data model subsumes every relation described in Chapter 2.
The following definition presents how a sequence diagram, as used in Chapter 2, can be extracted from the data model. Therefore, all sets and relations, which represent a sequence diagram, have to be limited to the objects that correspond to a specific level of abstraction in a given sequence. Consequently, all definitions from that chapter are applicable to a sequence in a given level of abstraction.

Definition 3.3 (Sequence to Sequence Diagram):
Let $M = (h, O, S, G, \sigma, \eta, t, \tau, \text{Seqs})$ be a data model and $\text{Seq} = (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence in Seqs. Further be $l \in \mathbb{N}_{\leq n}$ with $f(l) = (I_l, <_l)$.

$$SD_s^l = (I_l, E^l, O', \iota', \eta', t', \tau')$$

is the corresponding sequence diagram according to Definition 2.1 with:

$$\begin{aligned}
O' &= \{o \in O \mid \exists\ e \in E^l, i \in I_l : \iota(e) = (i, o)\} \\
&\quad O'_\alpha = O_\alpha \cap O' \text{ for } \alpha \in \{M, F, O, R, S, C, A, CO\} \\
\iota' &= \iota|_{E^l \times I_l \times O'} \\
\eta' &= \eta|_{O'_O \times O'_F} \\
t' &= t|_{O'_F \times \{\text{alt,break,...}\}} \\
\tau &= \tau|_{O'_O \times \Sigma}
\end{aligned}$$

References in a sequence diagram as seen in Figure 1.4 are a shorthand for inserting the content of another sequence diagram. In the context of the data model, the diagram that would be inserted is a sequence at a given level of abstraction. Since the referenced diagram is a sequence, it may also be refined and reused in more detailed sequence levels. The reference object, *i.e.*, an element from O_R, is a link to a sequence at a given level of abstraction and a reference event represents the occurrence of the reference object on an instance's lifeline.
The data model with the set of sequences is the foundation for the sequence diagram views. Thus, the following definition presents how sequence diagram views are represented according to the data model.

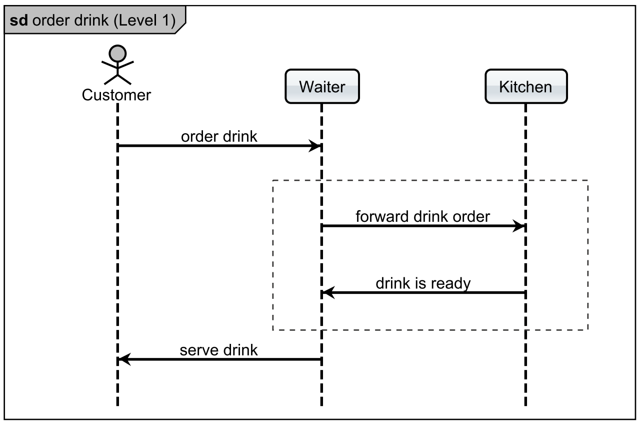
Definition 3.4 (Sequence Diagram View):
Let $M = (h, O, S, G, \eta, t, \tau, \text{Seqs})$ be a data model. A *sequence diagram view* is a tuple

$$SDV = (\text{Seq}, l, VI, \xi)$$

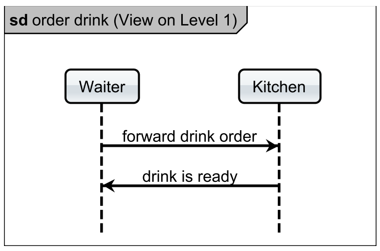
where

Seq	is a sequence,
l	$\in \mathbb{N}_{\leq n}$ is a level of abstraction,
VI	$\subseteq I^l$ is a set of instances, and
ξ	$: VI \rightarrow E \times E$ maps each instance to a start- and end event.

A sequence diagram view consists of a set of instances and for each instance a start event and end event, which represent the start and the end of the instance's lifeline. Thus, a sequence diagram view is a projection onto a sequence. This definition allows a range of sequence diagram views, since it represents a clipping of the instances and the event set for a sequence. Figure 3.4(a) shows the second level of abstraction from a small sequence order drink already known from Figure 3.3. Figure 3.4(b) depicts a sequence diagram view onto that level and it contains only the Waiter and Kitchen. The first event for the Waiter is the send event of the message forward drink order and the last event is the message receive event for drink is ready. Likewise, the first event for the Kitchen is the message receive event for forward drink order and the last event is the send event for drink is ready. The dashed rectangle in Figure 3.4(a) frames the events and objects that are visible in the adjoining view.



(a) Figure 3.3(b) without marker.



(b) A view onto Figure 3.4(a).

Figure 3.4.: Sequence diagram from Figure 3.3(b) and a view onto that sequence.

Views are only valid together with the underlying model and each model operation that affects a view, *e.g.*, remove an instance that is part of a view, has to adjust the view object, too. Thus, the following definition presents a structure that combines model and views.

Definition 3.5 (Model/View System):
A *model/view system* (MV) is a tuple

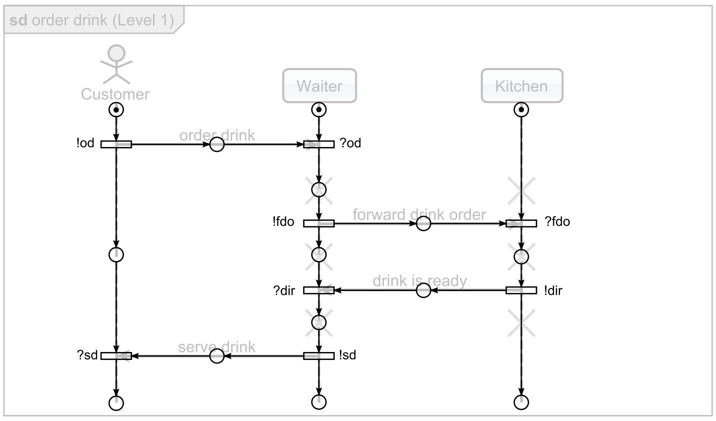
$$MV = (M, V)$$

where

$$\begin{aligned} M &= (h, O, S, G, \eta, t, \tau, \text{Seqs}) \text{ is a data model,} \\ V &\quad \text{is a set of sequence diagram views of } M. \end{aligned}$$

3.2.1. Identify possible event orderings

Figure 3.3 in the introduction of this chapter presents a sample integration for the insertion of a new message. There are six insertion points in Figure 3.3(b) that



(a) Petri net with initial marking.

od	order drink
sd	serve drink
fdo	forward drink order
dir	drink is ready

(b) Abbreviations.

Figure 3.5.: The Petri net representation of Figure 3.3(b).

are valid integrations for the new message receive event. The points between two existing events correspond to event orderings, *i.e.*, tuples from $E \times E$. The position of the message receive event in the source diagram (cf. Figure 3.3(b)) will default the occurrence of the new receive event after the receive event of order drink and before the send event of serve drink. Both of these events have counterparts in the refining diagram and, thus, are also a constraint for the position of the new message receive event. A Petri net representation of the sequence diagram determines the possible event orderings by using a special execution model. Each event in the sequence diagram corresponds to a Petri net transition and the Petri net places correspond to possible insertion points for new events. The complete transformation from a given sequence diagram to a Petri net is explained in the remainder of this section. Figure 3.5 depicts the Petri net representation of the sequence diagram in Figure 3.3(b). The main idea of this algorithm is to find a minimal execution of the Petri net that the receive event of order drink is executed and afterwards, to execute the Petri net maximal until the send event of serve drink is the only possible execution. During the maximal execution phase, all insertion points are stored.

To generalize the problem, let $s = (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence and k a level of abstraction in s. If a new event x in a sequence level k is "framed" by an event a upwards and an event b downwards, which event orderings are valid insertion points in the next/previous level of abstraction?

Function 3.1:
The following steps are necessary to identify the set of valid insertion points:

1. Transform the next sequence level $k+1$ (the previous sequence level $k-1$) into a Petri net $PN = (P, T, F, m_0)$
2. Identify events a' and b' that represent the ordering constraints in level $k+1$ $(k-1)$

3. Execute the Petri net PN minimal until transition a' was executed
4. From now on, save all places that are currently marked or that will get a marker in the subsequent steps.
5. Execute the Petri net maximal until b' is the only selectable transition.
6. Remove all places from the list that are between message send and receive events of the same message and that belong to instances, which are not allowed as insertion points: the only allowed instances are the master instance for the previous level of abstraction and only child instances for the next level of abstraction, respectively.

Each place from the generated list corresponds to an event ordering for a lifeline and represents a valid insertion point for the event x in the next/previous level of abstraction. The following paragraph describes and presents an algorithm for the Petri net transformation, *i.e.*, for the first step of Function 3.1.

Petri net transformation Petri nets form a well-suited semantic representation of sequence diagrams [22]. The main idea of the algorithm is to transform sequence diagram events into Petri net transitions and preserve the lifeline flow for each instance. Thus, for each lifeline there is at least one initially marked place that represents the start of the corresponding lifeline. Table 3.1 contrasts sequence diagram objects with their corresponding Petri net representation. The dashed line that separates two successive operands in the visual representation is formalized by two events, namely the operand begin event of the first operand and the operand end event of the second operand. Thus, the formalization would allow events between two operands that have no visual representation. To avoid new events between two operands, the transformation ignores the operand end events of all but the last operand. Continuations are not part of the transformation, because continuations have to be the first or last element in an alt fragment and, thus, no new event can be inserted between the continuation and the fragment border. Let $M = (h, O, S, G, \eta, t, \tau, \text{Seqs})$ be a data model, $s = (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence in M and k a level of abstraction in s with $f(k) = (I^k, <^k)$. Algorithm 3.1 represents the transformation for the sequence diagram $sd = (I^k, E, O, \iota, <^k, \eta, t, \tau)$.

The first step generates a Petri net for the next/previous sequence level and the later steps execute the Petri net with the ordering constraints from the source sequence level. These ordering constraints are adjacent events in the source sequence level. Many events do not have to be in the next or previous level, though, *e.g.*, references or actions. The next paragraph describes which events can be identified and how they can be identified.

Identify events in the next/previous sequence level Some events from a given level can be identified with events in the successive level or in the previous level, respectively. Those events are important, because they pose a constraint for valid

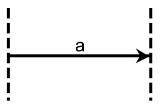
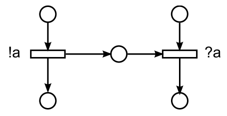
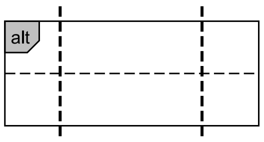
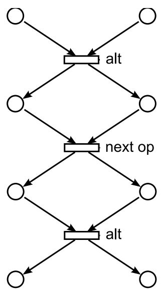
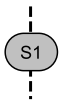
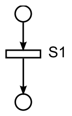
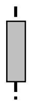
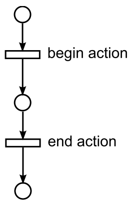
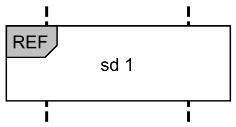
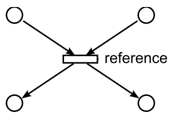
Object	Sequence Diagram Representation	Petri Net Representation
Message		
Fragment		
State Invariant		
Local Action (Execution Occurrence)		
Reference		

Table 3.1.: Petri net transformation for sequence diagram objects.

Algorithm 3.1 Petri net transformation
Inputs: Sequence Diagram $sd = (I, E, O, \iota, <, \eta, t, \tau)$

```
$P \leftarrow \{\}$, $T \leftarrow \{\}$, $F \leftarrow \{\}$
for all $i \in I$ do
  $e \leftarrow \min(E|_i)$
  curPlace $\leftarrow$ create new place
  $P \leftarrow P \cup \{\text{curPlace}\}$
  $m_0(\text{curPlace}) \leftarrow 1$
  while $\text{next}(e) \neq \bot$ do
    $(i, o) \leftarrow \iota(e)$
    if $e \in E_{MS} \cup E_{MR} \cup E_{CB} \cup E_{CE} \cup E_S \cup E_{AB} \cup E_{AE}$ then
      $T \leftarrow T \cup \{e\}$
      $F \leftarrow F \cup \{(\text{curPlace}, e)\}$
      if $e \in E_{MS}$ then
        $P \leftarrow P \cup \{o\}$, $m_0(o) \leftarrow 0$
        $F \leftarrow F \cup \{(e, o)\}$
      end if
      if $e \in E_{MR}$ then
        $P \leftarrow P \cup \{o\}$, $m_0(o) \leftarrow 0$
        $F \leftarrow F \cup \{(o, e)\}$
      end if
      curPlace $\leftarrow$ create new place
      $P \leftarrow P \cup \{\text{curPlace}\}$
      $m_0(\text{curPlace}) \leftarrow 0$
      $F \leftarrow F \cup \{(e, \text{curPlace})\}$
    end if
    if $\text{next}(e) \neq \bot$ then
      $(i', o') \leftarrow \iota(\text{next}(e))$
    else
      $(i', o') \leftarrow (\bot, \bot)$
    end if
    if $e \in E_{OB} \vee (e \in E_{OE} \wedge \eta(o) \neq \eta(o'))$ then
      $T \leftarrow T \cup \{o\}$
      $F \leftarrow F \cup \{(\text{curPlace}, o)\}$
      curPlace $\leftarrow$ create new place
      $P \leftarrow P \cup \{\text{curPlace}\}$
      $m_0(\text{curPlace}) \leftarrow 0$
      $F \leftarrow F \cup \{(o, \text{curPlace})\}$
    end if
    $e \leftarrow \text{next}(e)$
  end while
end for
return $(P, T, F, m_0)$
```

event orderings when adding new elements to the model. The number of identifiable events varies depending on the target level of abstraction. In general, message events and operand events are identifiable. Message send and receive events can be identified, since messages are in a one-to-one refinement relation (cf. Section 2.4). Fragments and operands are unique for all levels, thus, a fragment in the current level is the same as in the successive level. For that reason each operand begin/end event can be identified. The set of messages and operands is monotonically increasing with the level of abstraction. Thus, should events be identified relative to the next more detailed sequence level, the nearest message or operand event can be used. Identifying events for the next more abstract level is more complex, since the target level might not contain all message and operand events. Both algorithms can be found in the appendix, namely *indentifyDownwards* to identify the next possible event in the successive level of abstraction and *identifyUpwards* for the next previous level of abstraction.

Following the introduction on how to identify possible event orderings, the following paragraph describes the Petri net execution model. This execution model uses the Petri net from the first step and the two identified events a' and b'.

Petri net execution model The Petri net represents the order of events from the given sequence level and, thus, allows determining the set of event orderings between an ordering constraint, namely between two events a', b'. To maximize the size of the event ordering set, the first constraint a' has to be minimally fulfilled. Thus, a minimal execution sequence of transitions has to be found, which begins with the initial marking and ends by enabling transition a'. Algorithm 3.2 generates a set of transitions beginning at transition a'. The algorithm traces the shortest way from a' to the initial marking and finds each transition that has to be fired before a' will be enabled.

After creation of the backtrack set the Petri net has to be executed to get the marking that represents the ordering constraint a'. Algorithm 3.3 executes the set by going through the set and firing each enabled transition. If the event a' is undefined, Algorithm 3.2 and Algorithm 3.3 do not have to be executed and, thus, the Petri net remains in its initial marking. The marking as generated by Algorithm 3.3 conforms to the first ordering constraint a'. From now on, each marked place represents a new possibility to insert the new event x. To get all valid possibilities according to the event order, the Petri net has to fire all transitions until b' remains the only enabled transition. Algorithm 3.4 takes the marking m from Algorithm 3.3 and generates the set of event orderings. Therefore, the algorithm loops over all transitions and fires each enabled transition except b' until there are no enabled transitions left. The algorithm adds each place that has a marker according to m or gets a marker in the execution sequence to a set of places. Each place in this set represents an event ordering in the sequence diagram. If the event b' is undefined, the Petri net will be executed until the end. Since there are no loops in the Petri net, it will terminate if all lifeline tokens are in the last place of the lifeline.

Algorithm 3.2 Petri net Backtracking

Inputs: Petri Net pn, Transition t

```
new $\leftarrow \{t\}$
backtrackSet $\leftarrow \{\}$
while new $\neq \emptyset$ do
  $t \leftarrow$ any element from new
  new $\leftarrow$ new $\setminus \{t\}$
  backtrackSet $\leftarrow$ backtrackSet $\cup \{t\}$
  for all $p \in \bullet t$ do
    for all $k \in \bullet p$ do
      if $k \notin$ backtrackSet then
        new $\leftarrow$ new $\cup \{k\}$
      end if
    end for
  end for
end while
return backtrackSet
```

Algorithm 3.3 Petri net execute Backtracking - Set

Inputs: Petri Net $pn = (P, T, F, m_0)$, Set backtrackSet

```
$m \leftarrow m_0$
while backtrackSet $\neq \emptyset$ do
  for all $t \in$ backtrackSet do
    if $t$ is enabled then
      $m \xrightarrow{t} m'$
      $m \leftarrow m'$
      backtrackSet $\leftarrow$ backtrackSet $\setminus \{t\}$
    end if
  end for
end while
return $m$
```

Algorithm 3.4 Compute valid insertion points — Petri net full execution
Inputs: Petri Net $pn = (P, T, F, m_0)$, Marking m, Transition t

```
markedPlaces = empty list
for all $p \in P$ do
  if $m(p) = 1$ then
    add(markedPlaces, $p$)
  end if
end for
deadlock = false
while deadlock = false do
  deadlock = true
  for all $t' \in T$ do
    if $t'$ is enabled $\wedge$ $t' \neq t$ then
      deadlock = false
      $m \xrightarrow{t'} m'$
      $m \leftarrow m'$
      for all $p \in t'\bullet$ do
        add(markedPlaces, $p$)
      end for
    end if
  end for
end while
return markedPlaces
```

3.2.2. Classify possible event orderings

The Petri net approach determines all insertion points that are consistent with the event order of the sequence diagram as defined in Section 2.1. Some of these points are, however, of different integration quality. The introduction of this chapter presented an exemplary integration for a new message ask for peanuts. The possible insertion points in the inter-level refinement are determined by the Petri net approach. For the first and the last insertion point of the Kitchen there are traces of the refining diagram that are not possible in the master diagram. Thus, these two insertion points are of lower integration quality. Algorithm 3.5 also determines a set of insertion points, which in contrast are valid according to the traces of the refined diagram. To distinguish these different qualities, we call the insertion points computed by Algorithm 3.5 *trace valid*, which is a subset of the set of *valid* insertion points determined by the Petri net approach, *i.e.*, Algorithm 3.4. A restriction to trace valid insertion points only is not appropriate since the every day development often reveals additional constraints that avoid the problematic traces. UML offers, with general orderings, a syntactic possibility to make these ordering constraints explicit. General orderings are dashed lines between two events in a sequence diagram with an arrowhead in the middle, which depicts the direction of the additional or-

dering constraint.
The following approach classifies the set of insertion points according to the integration quality and automatically adds general orderings. To resume the example,

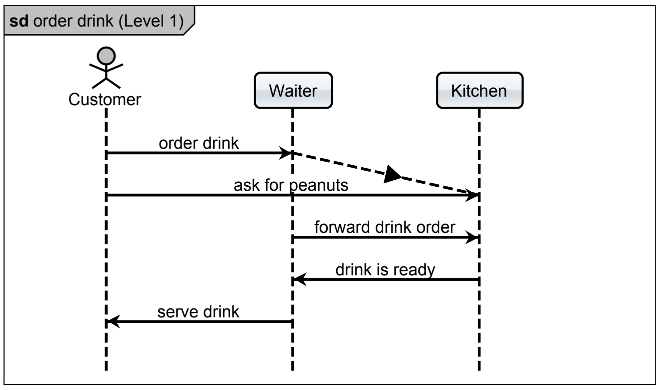


Figure 3.6.: Continuation of Figure 3.3(b) with general ordering.

Figure 3.6 depicts the integration of the message ask for peanuts with the general ordering that corrects the ordering between the receive event of order drink and the new receive event of ask for peanuts.
The procedure to generate and classify the set of insertion points is:

1. P_1 = set of insertion points according to the Petri net execution (Function 3.1).
2. P_2 = set of insertion points according to Algorithm 3.5.
3. Valid insertion points: $P_V = P_1 \cap P_2$.
4. Insertion points with general ordering: $P_G = P_1 \setminus P_2$.

When these insertion points are presented to the developer, the classification is depicted with different colors or different symbols (cf. Figure 3.3(b)).
As for the Petri net approach, the base of the classification are the Petri net transformation and the identified ordering constraints a' and b'. Algorithm 3.5 executes two breadth first searches (BFS) on the Petri net representation of the sequence diagram. The starting point of the first search is the transition or place that represents a' and the final point is the transition or place that represents b' and vice versa for the second search. The algorithm determines all places of direct paths between these two Petri net elements and, thus, each sequence of places that has a strict ordering between the ordering constraints a' and b'.
If the first ordering constraint a' is undefined, the breadth first search has to be done for each initial place of all refining instances and the given b' as endpoint. The result of the modified algorithm is the union of all result sets. If the second ordering constraint b' is undefined, the end place of each refined lifeline is a second ordering constraint. If both ordering constraints are undefined, the algorithm does not have

Algorithm 3.5 Compute trace valid insertion points — Petri net BFS
Inputs: Petri net $pn = (P, T, F, m_0)$, element $a \in P \cup T$, element $b \in P \cup T$

```
function traceValidIPs( Petri net $pn$, element $a \in P \cup T$, element $b \in P \cup T$, direction $d \in \{\text{up}, \text{down}\}$)
    visited : $P \cup T \rightarrow \{\textbf{true}, \textbf{false}\}$
    $\forall\ x \in P \cup T$ : visited($x$) ← false
    visited($a$) ← true
    toVisit ← [$a$]
    while toVisit ≠ [] do
        $e$ ← head(toVisit)
        toVisit ← tail(toVisit)
        if $e \neq b$ then
            $T \leftarrow e\bullet$
            if $d$ = up then
                $T \leftarrow \bullet e$
            end if
            for all $x \in T$ do
                if visited($x$) = false then
                    visited($x$) ← true
                    append(toVisit, $x$)
                end if
            end for
        end if
    end while
    return $\{p \in P \mid \text{visited}(p) = \textbf{true}\}$
end function

$L_1$ ← traceValidIPs($pn, a, b$, down)
$L_2$ ← traceValidIPs($pn, b, a$, up)
return $L_1 \cap L_2$
```

to be executed, since there is no restriction.
General orderings in the data model are an additional entry in the event order. If a general ordering should be inserted into a given level of abstraction in a sequence, a new tuple is inserted into the event order of that level. The automatic insertion of general orderings can be done between the identified events a', b' and the new inserted event x. If there is no direct path between a' and x, the new general ordering is (a', x). Analogous for b', if there is no direct path between x and b', the new general ordering is (x, b').

3.2.3. Refinement

Section 2.1 introduced informally the ideas of the refinement concept, which this thesis is based on. Consequently, the following definitions present the pending structural refinement rules in the context of the data model. There are several differences between the rules defined here and the original definitions by Ohlhoff [21]. These differences are explained along with the corresponding definition. In the data model each level of a sequence corresponds to a sequence diagram of the original definitions (cf. Definition 3.3).
The first definition describes which instances have to appear in successive sequence levels.

Definition 3.6 (Inter-level Instance Refinement):
Let Seq = $(E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence and $s, t \in \mathbb{N}_{\leq n}$ levels of abstraction in Seq with $t = s + 1$, $f(s) = (I_s, <_s)$, and $f(t) = (I_t, <_t)$. Sequence level t is an *inter-level instance refinement* of sequence level s, if s contains all master instances from instances in t and each instance in t that does not have a master according to the instance hierarchy has a parent instance in t, *i.e.*,

$$\begin{aligned} &\forall\ i \in I_t : \mu(i) \neq \bot \Rightarrow \mu(i) \in I_s \\ \wedge &\ \forall\ i \in I_t : \mu(i) = \bot \Rightarrow \pi(i) \in I_t \end{aligned}$$

cf. [21, p. 19].

A separate structural refinement rule for intra-level instance refinement as in the diploma thesis of Ohlhoff [21] is not necessary, since sequences as used here are complete, *i.e.*, each level of abstraction is an inter-level instance refinement of the previous level. The concept of this thesis allows no decoupled diagrams that depict a clipping of the behavior, since these diagrams would be views on a complete sequence.
The following definition prescribes which messages have to appear in successive sequence levels of abstraction.

Definition 3.7 (Inter-level Message Refinement):
Let Seq = $(E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence and $s, t \in \mathbb{N}_{\leq n}$ levels of abstraction in Seq with $t = s + 1$, $f(s) = (I_s, <_s)$, and $f(t) = (I_t, <_t)$. Further be sequence level t an inter-level instance refinement of sequence level s. t is an *inter-level message refinement* of s, if the following holds for all messages in s and t. For each message m in s, $G(m)$ also appears in t. If the sending and receiving instances of a message m in t are refinements of different instances of s, $G^{-1}(m)$ also appears in s. Furthermore, the sending and receiving instances of messages that appear in both sequence

levels are compatible with the instance hierarchy, *i.e.*,

$$\begin{aligned}&\forall\ m \in M_t : \bot \neq \mu(p(!m)) \neq \mu(p(?m)) \neq \bot \Rightarrow G^{-1}(m) \in M_s\\ \wedge\ &\forall\ m \in M_s\ \exists\ m' \in M_t :\\ &\qquad G(m) = m' \wedge p(!m) = \mu(p(!m')) \wedge p(?m) = \mu(p(?m'))\end{aligned}$$

where:

- $M_k := \{o \in O_M \mid \exists\ i \in I^k, e \in E^k : \iota(e) = (i,o)\}$ for all $k \in \mathbb{N}_{\leq n}$ is the set of messages in sequence level k and
- $p : E \to I$ with $\forall\ e \in E, i \in I : p(e) = i \Leftrightarrow \exists\ o \in O : \iota(e) = (i,o)$ is a mapping from events to instances

cf. [21, p. 20].

Compared to the original definition, messages from the master diagram have to appear in the next level. Consequently, the set of messages between successive levels is monotonically increasing.
Definition 3.8 describes which fragments have to appear in successive levels of abstraction.

Definition 3.8 (Fragment Complete):
Let Seq $= (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence and $s, t \in \mathbb{N}_{\leq n}$ levels of abstraction in Seq with $t = s+1$, $f(s) = (I_s, <_s)$, and $f(t) = (I_t, <_t)$. Further be sequence level t an inter-level instance refinement of sequence level s. s and t are *fragment complete* if t contains all combined fragments from s that cover instances whose refinements appear in t, *i.e.*,

$$\begin{aligned}&\forall\ o \in O_O, i \in I_s :\\ &(\exists\ e_B \in E_{OB}, e_E \in E_{OE} : \iota(e_B) = (i,o) = \iota(e_E)) \Rightarrow\\ &[\forall\ i' \in I_t : \mu(i') = i \Rightarrow \exists\ e'_B \in E_{OB}, e'_E \in E_{OE} : \iota(e'_B) = (i',o) = \iota(e'_E)].\end{aligned}$$

and s contains all fragments from t whose covered instances are refinements of different instances of s, *i.e.*,

$$\begin{aligned}&\forall\ o \in O_O :\\ &(\exists\ i, i' \in I_t, e, e' \in E : \iota(e) = (i,o) \wedge \iota(e') = (i',o) \wedge \mu(i) \neq \bot \neq \mu(i')) \Rightarrow\\ &[\forall\ i \in I_t :\\ &\quad (\exists\ e_B \in E_{OB}, e_E \in E_{OE} : \iota(e_B) = (i,o) = \iota(e_E) \wedge \mu(i) \in I_s) \Rightarrow\\ &\quad \exists\ e'_B \in E_{OB}, e'_E \in E_{OE} : \iota(e'_B) = (\mu(i),o) = \iota(e'_E)]\end{aligned}$$

cf. [21, p. 21].

In the original definition, the occurrence of an operand in successive levels of abstraction depends on the messages that are covered by the operand. The definition used here prescribes the appearance of operands based on the instances in that level.

The new definition is more rigorous, since all operands have to appear in each level of abstraction according to the contained instances. Just as the set of messages, the set of operands is also monotonically increasing.
The structural refinement rules describe nothing about the event order between two successive levels. The following notation introduces the restriction of the event set and event order to events covered by the refinement concept and how the event order of a high-level diagram brought forward to the next more detailed level of abstraction. The operation *identDownwards* can be found in the appendix. Furthermore, it defines for each event a set of operands that encapsulate the event. If the event does not belong to an operand, the operand set is empty.

Notation 3.1:
Let Seq $= (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence. Then we define

$$\text{Ident}(E) := E_{MS} \cup E_{MR} \cup E_{OB} \cup E_{OE}$$

For each level of abstraction $s \in \mathbb{N}_{\leq n}$ in Seq with $f(s) = (I_s, <_s)$, we define

$$\text{Ident}(<_s) := <_s |_{\text{Ident}(E) \times \text{Ident}(E)}$$

to be the restriction of the event order $<_s$ to identifiable events.
For each event $e \in E$ with $\iota(e) = (i, o')$ it is

$$\begin{aligned} \forall\ o \in O_O :\ & o \in \text{operandSet}(e) :\Leftrightarrow \\ & \exists\ e_B \in E_{OB}, e_E \in E_{OE} : \iota(e_B) = (i, o) = \iota(e_E) \wedge e_B < e \wedge e < e_E \end{aligned}$$

the set of operands that e belongs to.
For two levels of abstraction $s, t \in \mathbb{N}_{\leq n}$ in Seq with $t = s + 1$, $f(s) = (I_s, <_s)$, and $f(t) = (I_t, <_t)$ we define $\text{Ident}_t : E \times E \to \text{Ident}(E) \times \text{Ident}(E)$ as the corresponding event order of $<_s$ with events from level t:

$$\begin{aligned} \forall\ & e, e' \in \text{Ident}(E) : (e, e') \in <_s \Rightarrow \\ & (\text{identDownwards}(\text{Seq}, e), \text{identDownwards}(\text{Seq}, e')) \in \text{Ident}_t(<_s). \end{aligned}$$

The following definition presents a further consistency term for combined fragments and operands. The first part prescribes that fragments cannot overlap each other in a sequence and the second part prescribes that each identifiable event in a refining level of abstraction has to belong to at least the same operands as the corresponding event in the refinement master.

Definition 3.9 (Fragment Consistent):
Let Seq $= (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence and $s \in \mathbb{N}_{\leq n}$ a level of abstraction in Seq. We say s is *fragment consistent* if all operand begin and operand end events that belong to a single operand are nested in the same set of operands, *i.e.*,

$$\begin{aligned} \forall\ o \in O_O, e, e' \in E, i, i' \in I : \iota(e) = (i, o) \wedge \iota(e') = (i', o) \Rightarrow \\ \text{operandSet}(e) = \text{operandSet}(e') \end{aligned}$$

Let t be a further level of abstraction in Seq with $t = s + 1$ and let s and t be fragment complete, where s and t are each fragment consistent. We say s and t are *fragment consistent* if each identifiable event e from s resides in a subset of the operands that the corresponding event in t resides in, *i.e.*,

$$\forall\ e \in \mathrm{Ident}(E) : \mathrm{operandSet}(\mathrm{identDownwards}(\mathrm{Seq}, e)) \subseteq \mathrm{operandSet}(e)$$

cf. [21, p. 22].

Definition 3.10 combines the preceding definitions and specifies when two event orders between successive levels are consistent.

Definition 3.10 (Inter-level Refinement):
Let $\mathrm{Seq} = (E = E^0 \uplus \ldots \uplus E^n, \iota, f)$ be a sequence and $s, t \in \mathbb{N}_{\leq n}$ levels of abstraction in Seq with $t = s + 1$, $f(s) = (I_s, <_s)$, and $f(t) = (I_t, <_t)$. t is an *inter-level refinement* of s, if t is an inter-level message refinement of s, t and s are fragment consistent, and $\mathrm{Ident}_t(<_s) \cup \mathrm{Ident}(<_t)$ is acyclic

cf [21, p. 26].

Function 3.1 determines the valid insertion points for new messages and operands. The ordering constraints in that function are those events, which maintain the acyclicity of the unions of both event orders.
These refinement rules define constraints, which are fulfilled by the corresponding model operations in the following section.

3.3. Model Operations

The model operations provide the interface between the data representation and the user. Each operation maintains the consistency of the model. According to the example in the introduction of this chapter, some operations might not be resolved unambiguously since it is possible that several valid resolutions exists in other levels of abstraction. Only the developer can choose the desired option. Furthermore, maintaining the model consistency might involve a couple of changes to the data representation or even several model operations. All these changes to the data model have to be atomic, consistent, isolated, and durable for each operation. These four mentioned characteristics are well-known as ACID properties in the database transaction theory. Consequently, each model operation has to be encapsulated in a transaction, which may only be committed if each change is made and each ambiguity is resolved. Figure 3.7 depicts the flow of a generic model operation. Firstly, the user initiates an operation on a view. Secondly, the view creates a new transaction and delivers the operation with the corresponding parameters to the model. The model interacts with the data representation to get the necessary information and to perform the relevant changes. Finally, the view commits the transaction if the operation is complete or performs a rollback if an error occurred in between. The

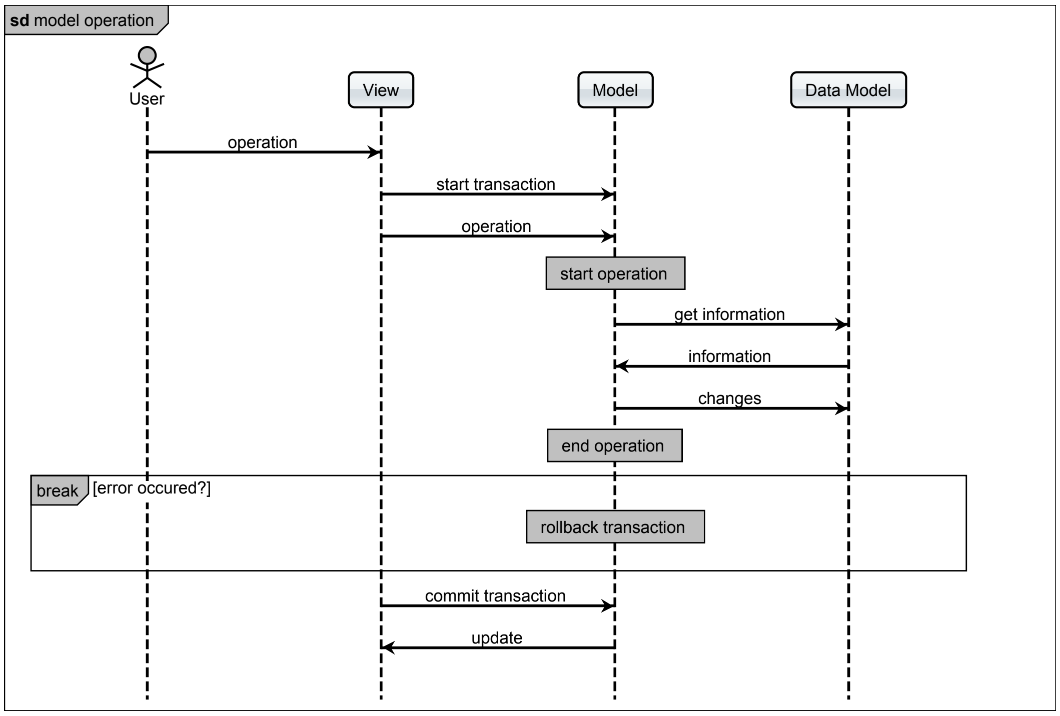


Figure 3.7.: A generic model operation.

rollback can also be triggered by the model, *e.g.*, if the error occurred inside the model.

The remainder of this section introduces all important model operations to create a new model and to work on it. Table 3.2 outlines the operations of this section. Let MV be a model/view system. Each operation is given as a list of input parameters, a list of preconditions, and the operation body. An operation body is only executed if all preconditions are fulfilled. Several operations share helping functions, *e.g.*, *addToEventOrder* to modify the event order, which can be found in the appendix. Furthermore, each operation that removes an event from a sequence has to assure the view integrity, *i.e.*, that the start and end event in the view definition are replaced with the next or previous event if the defined event is removed.

Ask User In several model operations a call will occur to an operation *askUser*, which represents that the developer has to resolve ambiguities or to add missing information. Even if the operation often can choose a valid solution itself, it cannot choose the desired solution, though. The operation *askUser* is a callback from the model with a specified interface and, thus, an oracle can take the role of the developer for non-interactive operations such as automated tests.

Operation	Description
create sequence	create a new sequence in the data model
delete sequence	delete an existing sequence from the data model
create level	create a new level of abstraction in a sequence
delete level	delete a level of abstraction in a sequence
create view	create a sequence diagram view
delete view	delete a sequence diagram view
create instance	create instance in the instance hierarchy
delete instance	delete instance from the instance hierarchy
add sequence instance	add an instance to a sequence
remove sequence instance	remove an instance from a sequence
add view instance	add an instance to a sequence diagram view
remove view instance	remove an instance from a sequence diagram view
add message	add a new message to a sequence
delete message	delete a message from a sequence
add fragment	add a new fragment to a sequence
delete fragment	remove a fragment from a sequence
add operand	add a new operand to a sequence
delete operand	delete an operand to a sequence
create state	create a new state for an instance
delete state	delete a state from an instance
relate states	relate two states of related instances
unrelate states	unrelate two states
add state	add a new state to a sequence
remove state	remove a state from a sequence
add reference	add a reference to a sequence
delete reference	delete a reference from a sequence
add local action	add a local action to a sequence
delete local action	delete a local action from a sequence
add continuation	add a continuation to a sequence
delete continuation	delete a continuation from a sequence
add general ordering	add a general ordering to a sequence
delete general ordering	delete a general ordering to a sequence

Table 3.2.: Overview of the model operations in Section 3.3.

3.3.1. Sequence

The user creates a new sequence to describe a new flow of the system or to define a new source for a reference. After creating a new sequence, the event set and the functions ι and f are empty, because there are no objects in the sequence.

create sequence

Input: —

Body:
```
$f(0) \leftarrow (\{\},\{\})$
$\text{Seqs} \leftarrow \text{Seqs} \cup \{(\{\},\{\},f)\}$
```

The delete operation has to remove all objects from the object set, which only belong to the events of that sequence. For that reason, it is sufficient to remove each level of abstraction. States are independent of the sequences and, thus, remain in the object set if a state event is removed.

delete sequence

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$

Require: – the sequence is not used as reference in another sequence

Body:
```
**for** $l = n$ to $0$ **do**
  **if** $f(l) \neq \bot$ **then**
    delete level$(s, l)$
  **end if**
**end for**
$\text{Seqs} \leftarrow \text{Seqs} \setminus \{s\}$
```

The following operation adds a new level of abstraction to a sequence. The instances for the new level of abstraction have to exist in the instance hierarchy before adding the new level. For messages and operands, the refinement concept defines which messages and operands have to be in the new level (cf. message operations and fragment operations). This operation adds the messages and fragments automatically into the new level, to maintain the model consistency. Other objects, such as actions, states, or continuations, are not transferred to the new level, since there is no constraint defined on how to add these objects. If an instance from the previous level is substituted by more than one instance in the new level, a proxy instance will be selected, which takes over the communication from the previous level.

create level

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$

Require:
- $l > 0$
- $f(l) = \bot \wedge f(l-1) \neq \bot$

– $f(l-1) = (I_{l-1}, <_{l-1}) \wedge \forall i \in I_{l-1}\ \exists\ i' \in I : \mu(i') = i$

Body:

```
$(I_{l-1}, <_{l-1}) \leftarrow f(l-1)$
$I' \leftarrow \{i \in I \mid \exists\ i' \in I_{l-1} : \mu(i) = i'\} \cup \{i \in I \mid \exists\ i' \in I_l : \pi(i) = i'\}$
for all $i \in I_{l-1}$ do
  $i' \leftarrow$ any instance from $\{i' \in I' \mid \mu(i') = i\}$
  $e \leftarrow \min(E|_i)$
  while $e \neq \bot$ do
    $(i, o) \leftarrow \iota(e)$
    if $e \in E_{MS} \cup E_{MR}$ then
      if $G(o) = \bot$ then
        $m \leftarrow$ new message with the same type as $o$
        $O_M \leftarrow O_M \cup \{m\}$
        $G(o) \leftarrow m$
      else
        $m \leftarrow G(o)$
      end if
      if $e \in E_{MS}$ then
        $e' \leftarrow$ message send event for $m$
        $E_{MS} \leftarrow E_{MS} \cup \{e'\}$
      else
        $e' \leftarrow$ message receive event for $m$
        $E_{MR} \leftarrow E_{MR} \cup \{e'\}$
      end if
      $\iota(e') \leftarrow (i', m)$
      $<_l \leftarrow \text{addToEventOrder}(e', (\max(E|_{i'}), \bot), <_l)$
    end if
    if $e \in E_{OB} \cup E_{OE}$ then
      for all $i'' \in \{i' \in I' \mid \mu(i') = i\}$ do
        if $e \in E_{OB}$ then
          $e' \leftarrow$ new operand begin event for $o$
          $E_{OB} \leftarrow E_{OB} \cup \{e'\}$
        else
          $e' \leftarrow$ new operand end event for $o$
          $E_{OE} \leftarrow E_{OE} \cup \{e'\}$
        end if
        $\iota(e') \leftarrow (i'', o)$
        $<_l \leftarrow \text{addToEventOrder}(e', (\max(E|_{i''}), \bot), <_l)$
      end for
    end if
    $e \leftarrow \text{next}(e)$
  end while
end for
$f(l) \leftarrow (I', <_l)$
```

To delete a level of abstraction, it has to be the lowest defined level, so that there is no dependency to another level. The operation removes all instances from that level, since each element that belongs to a sequence's level of abstraction in particular belongs to an instance.

delete level

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$

Require:
- $f(l+1) = \perp$
- the sequence is not used as reference in another sequence

Body:

```
$(I_l, <_l) \leftarrow f(l)$
for all $i \in I_l$ do
   remove sequence instance($s, i, l$)
end for
$f(l) \leftarrow \perp$
```

3.3.2. Views

A sequence diagram view depicts a clipping of the instances and events of a sequence. Thus, a new sequence diagram view consists of a sequence as information base.

create view

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$

Require:
- $f(l) \neq \perp$

Body: $V \leftarrow V \cup \{(s, \{\}, l, \{\})\}$

Views provide no information for the model and no other elements reference them. Hence, the delete operation for a sequence diagram view only has to remove the view structure.

delete view

Input: view $v \in V$

Body: $V \leftarrow V \setminus \{v\}$

3.3.3. Instances

Instances are part of the instance hierarchy, all sequences, and the sequence diagram views. Thus, the model needs at least six operations to add instances and remove them, respectively. The instance hierarchy is the only structure that consists of instances while sequences and sequence diagram views refer only to a subset of the instance hierarchy. The following model operation creates a new instance and adds it to the instance hierarchy.

create instance

Input: type $t \in \{\text{actor}, \text{component}\}$, level $l \in \mathbb{N}_{\leq n}$, master $m \in I \cup \{\bot\}$, parent $p \in I \cup \{\bot\}$

Require:

- only first level instances do not require a master or a parent, *i.e.*, if $l > 0 : m \neq \bot \ \vee \ p \neq \bot$
- if $m \neq \bot$: $m \in I^{l-1}$
- if $p \neq \bot$: $p \in I^l$

Body:

```
$i \leftarrow$ new instance with type $t$
$I^l \leftarrow I^l \cup \{i\}$
$\mu(i) \leftarrow m$
$\pi(i) \leftarrow p$
```

To delete an instance from the model, each object depending on that instance has to be removed first. The next operation calls the remove operation for each sequence and sequence diagram view. Afterwards it removes the instance from the instance hierarchy. The precondition requires that all refining instances are removed before the master or parent can be removed.

delete instance

Input: instance $i \in I$, level $l \in \mathbb{N}_{\leq n}$

Require:

- there are no inter-level refining instances in the instance hierarchy, *i.e.*, $\{i' \in I^{l+1} \mid \mu(i') = i\} = \varnothing$
- there are no intra-level refining instances in the instance hierarchy, *i.e.*, $\{i' \in I^l \mid \pi(i') = i\} = \varnothing$

Body:

```
for all $v \in V$ do
  remove view instance($v$, $i$)
end for
for all $s \in$ Seqs do
  remove sequence instance($s$, $i$, $l$)
end for
$\mu(i) \leftarrow \bot$
$\pi(i) \leftarrow \bot$
$I^l \leftarrow I^l \setminus \{i\}$
```

The set of instances in each level of abstraction in a sequence has to be well-formed according to Definition 2.4, *i.e.*, all instances have to belong to the same level of abstraction. Furthermore, the refinement concept defines with Definition 3.6, which instances have to belong to subsequent levels. In conclusion, the following operation adds an existing instance to a sequence.

add sequence instance

Input: sequence $s \in \text{Seq}$, instance $i \in I$, level $l \in \mathbb{N}_{\leq n}$

Require:
- the level of abstraction of instance is l, *i.e.*, $i \in I^l$
- if $\mu(i) \neq \perp \Rightarrow (I_{l-1}, <_{l-1}) = f(l-1) \wedge \mu(i) \in I_{l-1}$
- if $\pi(i) \neq \perp \Rightarrow (I_l, <_l) = f(l) \;\wedge\; \pi(i) \in I_l$

Body:
```
$(I_l, <_l) \leftarrow f(l)$
$f(l) \leftarrow (I_l \cup \{i\}, <_l)$
```

To remove an instance from a sequence, the operation needs to delete all events that belong to that instance. Consequently, it also needs to delete those objects that belong exclusively to the instance's lifeline and cannot remain in the model. More precisely, if the instance's lifeline contains a message send or receive event then this message would be incomplete without the event and has to be deleted, too. Coregions and actions belong to only one instance's event set and, thus, the operation has to delete these objects together with the instance itself. States are independent from a sequence and the operation only removes the state from the sequence but not from the model. Each reference usage in a sequence has to cover the same instances as in the reference definition. Consequently, the operation has to delete all reference objects that cover the instance that should be deleted. Fragments and continuations, however, have only to be deleted if the object covers the instance exclusively. Otherwise, it is sufficient to delete the corresponding events.

remove sequence instance

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, instance $i \in I$, level $l \in \mathbb{N}_{\leq n}$

Require:
- the level of abstraction of instance is l, *i.e.*, $i \in I^l$
- there are no inter-level refining instances for the given instance in seq, *i.e.*, $(I_{l+1}, <_{l+1}) = f(l+1), I_{l+1} \cap \{i \in I \mid \mu(i) = i\} = \varnothing$
- there are no intra-level refining instances for the given instance in seq, *i.e.*, $(I_l, <_l) = f(l), I_l \cap \{i \in I \mid \mu(i) = i\} = \varnothing$
- the sequence is not used as reference in another sequence

Body:
```
$(I_l, <_l) \leftarrow f(l)$
$E_i \leftarrow \{e \in E \mid \exists\, o \in O : \iota(e) = (i, o)\}$
**for all** $e \in E_i$ **do**
  $(i, o) \leftarrow \iota(e)$
  **if** $o \in O_M$ **then**
    delete message$(s, l, o)$
  **else if** $o \in O_R$ **then**
    delete reference$(s, l, o)$
  **else if** $o \in O_S$ **then**
```

```
        remove state$(s, l, o)$
    **else if** $o \in O_C$ **then**
        delete coregion$(s, l, o)$
    **else if** $o \in O_A$ **then**
        delete local action$(s, l, o)$
    **else if** $o \in O_O \cup O_{CO}$ **then**
        **if** $\{i' \in I_l \mid \exists\, e \in E : \iota(e) = (i'', o') \wedge i' = i'' \wedge o' = o\} = \{i\}$ **then**
            **if** $o \in O_O$ **then**
                delete fragment$(s, l, \eta(o))$
            **else**
                delete continuation$(s, l, o)$
            **end if**
        **else**
            $E \leftarrow E \cap \{e \in E \mid \iota(e) = (i, o)\}$
            $<_l \leftarrow <_l \setminus \{(e, e') \mid \iota(e) = (i, o) \vee \iota(e') = (i, o)\}$
        **end if**
    **end if**
**end for**
```

Each sequence diagram view depicts a clipping of the instances and event set for a given sequence and a level of abstraction. Thus, the operation to add an instance to a view requires two events besides the instance itself. These two events have to belong to the instance that should be added and have to be given in ascending order.

add view instance

Input: view $v = (\mathrm{Seq}, l, VI, \xi) \in V$, instance $i \in I$, event $e_1 \in E$, event $e_2 \in E$

Require:

- $VI \cup \{i\} \subseteq I^l$
- e_1, e_2 belong to instance i and $e_1 < e_2$

Body:
$VI \leftarrow VI \cup \{i\}$
$\xi(i) \leftarrow (e_1, e_2)$

The operation to remove an instance from a view has only to modify the instance set and the event mapping.

remove view instance

Input: view $v = (\mathrm{Seq}, l, VI, \xi) \in V$, instance $i \in I$

Body:
$VI \leftarrow VI \setminus \{i\}$
$\xi(i) \leftarrow \perp$

3.3.4. Messages

Messages consist of a message send event and a message receive event. Thus, the operation to add a new message requires a send event order and a receive event order. After inserting the message into the intended level of abstraction, a corresponding message possibly has to be integrated into the other levels of the same sequence. Definition 3.7 specifies whether new messages have to be integrated into the next higher and lower level. If the message has to be integrated into another level of abstraction, the Petri net algorithm from the previous section finds the appropriate event orderings. Afterwards the breadth first search allows the classification between insertion points that are completely covered by the traces of the next more abstract level and the insertion points allowed by the event order. The following model operation adds a new message into the sequence. Since the integration algorithm is long, the operation presented here is only an abstraction of the full operation. The complete version can be found in the appendix.

add message

Input: sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, send order $so \in <$, receive order $ro \in <$

Require:

- $so = (e, e')$: e, e' belong to the same instance and e, e' are subsequent according to the event order, *i.e.*, $e < e' \wedge \forall\ e'' \in E : e \neq e'' \neq e' \Rightarrow e'' < e \vee e' < e''$
- $ro = (e, e')$ analogous to so
- ro is a valid message receive order according to so, *i.e.*, the event order $<$ is acyclic after inserting the new message events.

Body:

```
**function** traverseUpwards(sequence $s \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, message $m \in O_M$, send order $so \in <$, receive order $ro \in <$)
  **if** $\perp \neq \mu(\text{getInstance}(so)) \neq \mu(\text{getInstance}(so)) \neq \perp$ **then**
    $m' \leftarrow$ new message with relation to $m$
    $(P, P') \leftarrow$ determine a set of valid insertion points upwards for $(so, ro)$
    $(eo_S, eo_R) \leftarrow (\text{askUser}(P), \text{askUser}(P'))$
    insert message $m'$
    traverseUpwards$(s, l-1, m', eo_S, eo_R)$
  **end if**
**end function**

**function** traverseDownwards(sequence $s \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, message $m \in O_M$, send order $so \in <$, receive order $ro \in <$)
  **if** $\exists\ i \in I : \mu(i) \in \{\text{getInstance}(so), \text{getInstance}(ro)\}$ **then**
    $m' \leftarrow$ new message with relation to $m$
```

```
            $(P, P') \leftarrow$ determine a set of valid insertion points downwards for
            $(so, ro)$
            $(eo_S, eo_R) \leftarrow (\text{askUser}(P), \text{askUser}(P'))$
            insert message $m'$
            traverseDownwards$(s, l+1, m', eo_S, eo_R)$
        end if
    end function

    $m \leftarrow$ new message
    insert message $m$
    traverseUpwards($s$, $l$, $m$, $so$, $ro$)
    traverseDownwards($s$, $l$, $m$, $so$, $ro$)
```

To remove an existing message from all levels of abstraction in a sequence s, the operation has to iterate over the message refinement relation to get the message objects in each level. Section 3.2 describes the classification of insertion points, where some of the insertion points are only valid with an additional general ordering. Likewise, the delete message operation has to verify that the deletion of the message does not invalidate the already existing event order according to the master diagram. For example if a message that should be deleted synchronizes two lifelines, the operation has to add new general orderings to resolve the event order according to the more abstract higher level.

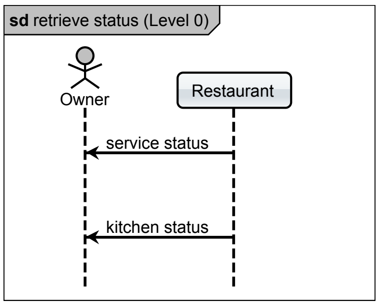
Function 3.2:
The following steps are used to find out whether the deleted message with send event e_{MS} and receive event e_{MR} has to be replaced by a general ordering after the message was deleted. For this purpose, let $<^+$ denote the transitive closure of $<$.

1. $e_1 = \text{findUpwards}(s, e_{MS}, \text{UP})$

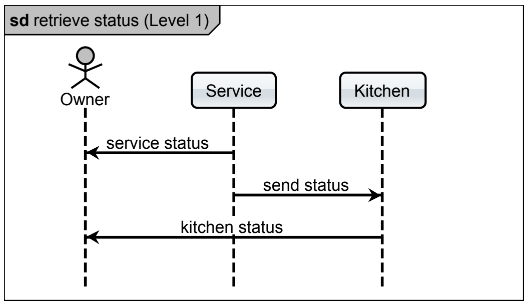
2. $e_2 = \text{findUpwards}(s, e_{MR}, \text{DOWN})$

3. add general ordering (e_1, e_2), if $\text{identUpwards}(s, e_1) <^+_{l-1} \text{identUpwards}(s, e_2)$ and $e_1 \not<^+_l e_2$

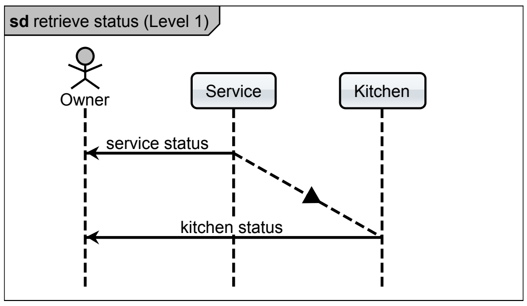
Figure 3.8 depicts an example where the deletion of a message implicates a new general ordering. The master diagram in Figure 3.8(a) shows the Owner retrieving the status of the Service and the Kitchen. The refining diagram in Figure 3.8(b) reveals the internal behavior of the Restaurant and depicts that Service and Kitchen send their own status. The message send status maintains the message order of the master diagram, since without send status both send events could occur in arbitrary order. In the following, the message send status should be deleted. The first step of Function 3.2 finds the next identifiable event e_1 above the send event of send status, namely the send event of service status. The second step finds the next identifiable event e_2 below the receive event of send status, namely the send event of kitchen status. Moreover, the operation *identUpwards* returns for both found events the counterpart in level 0. The condition of the third step of Function 3.2 is fulfilled, since e_1 and e_2 are not ordered in level 1 after the deletion of send status, while the identified counterparts in level 0 are ordered. Consequently, a new general ordering has to be inserted into the diagram between e_1 and e_2, which is depicted in Figure 3.8(c).



(a) The Owner retrieves the status of the service and the kitchen.



(b) The message send status synchronizes the unrelated send events in level 1.



(c) A general ordering undertakes the task of send status.

Figure 3.8.: The deletion of send status implicates a new general ordering.

delete message

```
Input:    sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\le n}$, message $m \in O_M$
Body:       $(I_l, <_l) \leftarrow f(l)$
            $\{e_{MS}\} \leftarrow \{e \in E_{MS} \mid \exists\, i \in I_l : \iota(e) = (i, m)\}$
            $\{e_{MR}\} \leftarrow \{e \in E_{MR} \mid \exists\, i \in I_l : \iota(e) = (i, m)\}$
            $\iota(e_{MS}) \leftarrow \bot$
            $\iota(e_{MR}) \leftarrow \bot$
            $E \leftarrow E \setminus \{e_{MS}, e_{MR}\}$
            $<_l \leftarrow <_l \setminus \{(e_1, e_2) \mid e_1 \in \{e_{MS}, e_{MR}\} \wedge e_2 \in E\}$
            $<_l \leftarrow <_l \setminus \{(e_1, e_2) \mid e_2 \in \{e_{MS}, e_{MR}\} \wedge e_1 \in E\}$
            **if** $l > 0$ **then**
              $e_1 \leftarrow \text{findUpwards}(s, e_{MS}, \text{UP})$
```

```
        $e_2 \leftarrow$ findUpwards$(s, e_{MR}, \text{DOWN})$
        $(I_{l-1}, <_{l-1}) \leftarrow f(l-1)$
        **if** $(e_1, e_2) \notin <_l \wedge$ identUpwards$(s, e_1) <_{l-1}$ identUpwards$(s, e_2)$ **then**
            $<_l \leftarrow <_l \cup \{(e_1, e_2)\}$
        **end if**
    **end if**
    **if** $G^{-1}(m) \neq \perp$ **then**
        $m' \leftarrow G^{-1}(m)$
        $G(m') \leftarrow \perp$
        delete message$(m', l-1)$
    **end if**
    **if** $G(m) \neq \perp$ **then**
        $m' \leftarrow G(m)$
        $G(m) \leftarrow \perp$
        delete message$(m', l+1)$
    **end if**
    $O_M \leftarrow O_M \setminus \{m\}$
```

3.3.5. Fragments

Fragments are shared between all levels of abstraction in the same sequence. Depending on the type, fragments consist of one or more operands. Since fragments and operands are different objects, the first two operations add and delete a new fragment and the following two operations add and delete operands. Operands just like messages have to be integrated into the other levels of abstraction in the same sequence. Definition 3.8 describes which fragments and which operands have to be integrated.

The following operation adds a new fragment to a sequence. Since each fragment has at least one operand, the operation calls the *add operand* operation. Thus, the parameter list includes all values which are needed to add an operand.

add fragment

Input: sequence $s \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, type $p \in$ { alt, assert, break, consider, critical, ignore, loop, neg, opt, par, seq, strict }, assertion $a \in \Sigma$, instances $I' \subseteq I$, event order eo $\subseteq <$

Require: – see add operand

Body:

```
$o_F \leftarrow$ new fragment of type $t$
$O_F \leftarrow O_F \cup \{o_F\}$
$t(o_F) \leftarrow p$
add operand$(o_F, a, I', eo)$
```

The operation to delete a fragment has to remove all operands that belong to the fragment before the fragment object can be removed.

delete fragment

Input: sequence $s \in \text{Seqs}$, fragment $o_F \in O_F$

Require: – see delete operand

Body:

```
$Q \leftarrow \{o \in O_O \mid \eta(o) = o_F\}$
**for all** $o \in Q$ **do**
  delete operand$(s, o)$
**end for**
$t(\mathrm{o_F}) \leftarrow \bot$
$O_F \leftarrow O_F \setminus \{o_F\}$
```

The following operation adds a new operand to a fragment. A parameter with a subset of the instances in that level of abstraction specifies the covered lifelines. An event ordering for each lifeline determines the insertion point for the operand. Furthermore, operands are drawn as rectangles in the sequence diagram. Hence, the given event orderings have to allow that. To check if the event orderings fulfill this restriction, the current level of abstraction has to be converted to the corresponding Petri net representation (cf. Algorithm 3.1). Each event ordering equals a marking of one place in the Petri net. Consequently, the condition for the check is, if the marking that represents the event ordering is reachable in the Petri net. Since an operand does not have to cover each lifeline, the event orderings represent no complete marking. This problem is called *submarking reachability* [20].
Definition 3.8 describes which fragments have to be in which levels of abstraction. Thus, the operation iterates over the levels and inserts the corresponding operand events in the appropriate levels. The Petri net execution model is used to determine valid insertion points in other levels. Since the integration algorithm is long, the operation presented here is only an abstraction of the complete operation that can be found in the appendix.

add operand

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, fragment $o_F \in O_F$, assertion $a \in \Sigma$, instances $T \subseteq I$, event orders $eos \subseteq <$

Require:

- if $t(o_F) \in \{\text{loop}, \text{opt}, \text{break}, \text{neg}\} \Rightarrow |\eta^{-1}(o_F)| = 0$
- for each $i \in T$ there is an $(e, e') \in eos$: e, e' belongs to instance i and there is no other $(e, e') \in eos$ that belongs to i
- for each $(e, e') \in eos$: e, e' are subsequent according to the event order, *i.e.*, $e < e' \wedge \forall\ e'' \in E : e \neq e'' \neq e' \Rightarrow e'' < e \vee e' < e''$
- check event orders (see above)

Body:

```
**function** traverseUpwards(sequence $s \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, operand $o_O \in O_O$, instances $T \subseteq I$, event orders $eos \subseteq <$)
```

```
        $M \leftarrow \{i \in I \mid \exists\ i' \in T : \mu(i') = i\}$
        **if** $|M| > 1$ **then**
            **for all** $m \in M$ **do**
                identify events from $eos$ upwards
                execute Petri net for the identified events
                $eo \leftarrow$ chosen event ordering
                $eos' \leftarrow eos' \cup \{eo\}$
                $T' \leftarrow T' \cup \{m\}$
            **end for**
            traverseUpwards($s, l - 1, o_O, T', eos'$)
        **end if**
    **end function**

    **function** traverseDownwards(sequence $s \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, operand
    $o_O \in O_O$, instances $T \subseteq I$, event orders $eos \subseteq <$)
        **if** $\{i \in I \mid \mu(i) \in T\} \neq \varnothing$ **then**
            **for all** $(e_1, e_2) \in eos$ **do**
                identify events $e_1$, $e_2$ downwards
                execute Petri net for the identified events
                **for all** $c \in \{i' \in I \mid \mu(i') = \text{getInstance}((e_1, e_2))\}$ **do**
                    $eo \leftarrow$ chosen event ordering for instance $c$
                    $eos' \leftarrow eos' \cup \{eo\}$
                    $T' \leftarrow T' \cup \{c\}$
                **end for**
            **end for**
            traverseDownwards($s, l + 1, o_O, T', eos'$)
        **end if**
    **end function**

    $o_O \leftarrow$ new operand
    insert operand $o_O$
    traverseUpwards($s, l, o_O, T, eos$)
    traverseDownwards($s, l, o_O, T, eos$)
```

To remove an operand, each operand begin and end event that belongs to the operand has to be removed from the event set and the event order of all levels of abstraction. Afterwards the operand object can be removed from the object set.

delete operand

Input: sequence $s = (E, \iota, f) \in$ Seqs, operand $o_O \in O_O$

Require: – operand is empty in all levels of abstraction

Body: **for** $l = 1$ to n **do**

```
    if $f(l) \neq \perp$ then
      $(I_l, <_l) \leftarrow f(l)$
      $E' = \{e \in E \mid \exists\ i \in I_l : \iota(e) = (i, o_O)\}$
      for all $e_O \in E'$ do
        $\iota(e_O) \leftarrow \perp$
        $<_l \leftarrow <_l \setminus \{(e_1, e_2) \in <_l \mid e_1 = e_O \vee e_2 = e_O\}$
      end for
      $E \leftarrow E \setminus E'$
    end if
  end for
  $\eta(o_O) = \perp$
  $\tau(o_O) = \perp$
  $O_O \leftarrow O_O \setminus \{o_O\}$
```

3.3.6. States

Each state belongs to an instance and is independent from the set of sequences since states are used in possibly many sequences. Moreover, states are in a refinement relation (cf. Section 2.3). Overall, there are six operations to handle states: The first two operations are responsible to create and delete states for an instance. Existing states can be put into relation or removed from the state relation with the *relate* operation and the *unrelate* operation. The last two operations add and remove states to a sequence as a local state event.
If the user creates a new state in the model, this state has to be put in relation with other states to retain a state consistent model (cf. Definition 2.10). These other states need not exist already and the operation makes sure that all necessary states are created to comply with Definition 2.10.

create state

Input: instance $i \in I$

Body:

```
  function insert state (instance $i \in I$, state $o'_S \in O_S$, state $o''_S \in O_S$)
    $o_S \leftarrow$ new state for instance $i$
    $\sigma(o_S) \leftarrow i$
    $O_S \leftarrow O_S \cup \{o_S\}$
    if $o'_S \neq \perp$ then
      $S \leftarrow S \cup \{(o_S, o'_S)\}$
    end if
    if $o''_S \neq \perp$ then
      $S \leftarrow S \cup \{(o''_S, o_S)\}$
    end if
    $T = \{o \in O_S \mid \sigma(o) = \mu(i)\}$
    if $\mu(i) \neq \perp \wedge (\exists\ o \in T : (o, o_S) \in S) = $ false then
      $d \leftarrow$ askUser(add new state for $\mu(i)$ or relate existing state?)
```

```
        **if** $d$ = add new state for instance $\mu(i)$ **then**
            insert state($\mu(i), o_S, \perp$)
        **else**
            $o \leftarrow$ askUser(select state from $T$)
            $S \leftarrow S \cup \{(o, o_S)\}$
        **end if**
    **end if**
    $C \leftarrow \{i' \in I \mid \mu(i') = i\}$
    **for all** $i' \in C$ **do**
        $T \leftarrow \{o \in O_S \mid \sigma(o) = i'\}$
        **if** $(\exists\ o \in T : (o_S, o) \in S)$ = **false then**
            $d \leftarrow$ askUser(add new state for $i'$ or relate existing state?)
            **if** $d$ = add new state for instance $i'$ **then**
                insert state($i', \perp, o_S$)
            **else**
                $o \leftarrow$ askUser(select state from $T$)
                $S \leftarrow S \cup \{(o_S, o)\}$
            **end if**
        **end if**
    **end for**
**end function**

insert state($i, \perp, \perp$)
```

The operation to delete a state o_S has to remove all relations to the state and from the state. All states that are no longer consistent with the state refinement after removing o_S have to be reintegrated to the state refinement to maintain the state consistency. For each removed relation the user has to choose between three possibilities: The related state can also be deleted, an existing state takes over the relation, or a new state for the same instance may take over the relation.

delete state

Input: state $o_S \in O_S$

Require: – no sequence has a state event for o_S

Body:

```
$K \leftarrow \{o \in O_S \mid (o_S, o) \in S\}$
**for all** $o \in K$ **do**
    **if** $(\exists\ o' \in O_S : (o', o) \in S \wedge o' \neq o_S)$ = **false then**
        $S \leftarrow S \setminus \{(o_S, o)\}$
        $d \leftarrow$ askUser(create new state, delete state, relate state?)
        **if** $d$ = create new state **then**
            insert state($\sigma(o_S), o, \perp$)
        **else if** $d$ = delete state **then**
```

```
            delete state($o$)
        else
            $o_N \leftarrow$ askUser(select state from instance $\sigma(o_S)$)
            relate state($o_N, o$)
        end if
    else
        $S \leftarrow S \setminus \{(o_S, o)\}$
    end if
end for
$K \leftarrow \{o \in O_S \mid (o, o_S) \in S\}$
for all $o \in K$ do
    if $(\exists\ o' \in O_S : (o, o') \in S \wedge o' \neq o_S)$ = false then
        $S \leftarrow S \setminus \{(o, o_S)\}$
        $d \leftarrow$ askUser(create new state, delete state, relate state?)
        if $d$ = create new state then
            insert state($\sigma(o_S), \perp, o$)
        else if $d$ = delete state then
            delete state($o$)
        else
            $o_N \leftarrow$ askUser(select state from instance $\sigma(o)$)
            relate state($o, o_N$)
        end if
    else
        $S \leftarrow S \setminus \{(o, o_S)\}$
    end if
end for
$O_S \leftarrow O_S \setminus \{o_S\}$
```

According to the state refinement, each state may be related to a set of other states. Thus, the following operation relates two existing states. The precondition assures that only states of successive instances in the instance hierarchy can be added to the state refinement relation.

relate states

Input:	state $o_S \in O_S$, state $o'_S \in O_S$
Require:	– $\mu(\sigma(o'_S)) = \sigma(o_S)$
Body:	$S \leftarrow S \cup \{(o_S, o'_S)\}$

State relations can be removed as long as there remains at least one relation to maintain the state refinement consistency.

unrelate states

Input: state $o_S \in O_S$, state $o'_S \in O_S$

Require:
- $\exists\, o \in O_S : o \neq o'_S \wedge (o_S, o) \in S \wedge \sigma(o) = \sigma(o'_S)$
- $\exists\, o' \in O_S : (o', o'_S) \in S$

Body: $S \leftarrow S \setminus \{(o_S, o'_S)\}$

States are used in sequences to define state invariants and in particular to determine the start and end states. Only sequences that are annotated with start states and end states can appear in the level state machine.

add state

Input: sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, state $a \in O_S$, eventOrdering $eo \in <$

Require:
- eo is a valid event order
- l is a level in s, *i.e.*, $f(l) \neq \bot$

Body:
```
$e_S \leftarrow$ new state event for $a$
$(I_l, <_l) \leftarrow f(l)$
$E_S \leftarrow E_S \cup \{e_S\}$
$<_l \leftarrow \text{addToEventOrder}(e_S, eo, <_l)$
$\iota(e_S) \leftarrow (\text{getInstance}(eo), a)$
```

Each state of a sequence can be removed without a precondition, since there is no condition for states. If the sequence is no longer state consistent after the state was removed (cf. Definition 2.8), it would not be used to generate the level state machine.

remove state

Input: sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, state event $e_S \in E_S$

Body:
```
$(I_l, <_l) \leftarrow f(l)$
$E_S \leftarrow E_S \setminus \{e_S\}$
$\iota(e_S) \leftarrow \bot$
$<_l \leftarrow <_l \setminus \{(e_1, e_2) \mid e_1 = e_S \vee e_2 = e_S\}$
```

3.3.7. References

References refer to a sequence at a given level of abstraction. The following operation takes two sequences, a level of abstraction, and a set of event orderings to insert a reference for the second sequence into the first sequence. The new reference object has to cover each instance that is used in the referred sequence.

add reference

Input: sequence $s = (E_s, \iota_s, f_s) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, event orderings $eos \subseteq <$, reference $r = (E_r, \iota_r, f_r) \in \text{Seqs}$

Require:
- check event orders
- eos contains all lifelines for the reference, *i.e.*, $(I_r, <_r) = f_r(l),\ \forall\ i \in I_r\ \exists (e_1, e_2) \in eos : \text{getInstance}((e_1, e_2)) = i$

Body:

```
$(I_s, <_s) \leftarrow f_s(l)$
$o_R \leftarrow$ new reference for sequence $r$, level $l$
$O_R \leftarrow O_R \cup \{o_R\}$
for all $eo \in eos$ do
  $e_R \leftarrow$ reference event for $o_R$
  $E_s \leftarrow E_s \cup \{e_R\}$
  $\iota(e_R) \leftarrow (\text{getInstance}(eo), o_R)$
  $<_s \leftarrow \text{addToEventOrder}(e_R, eo, <_s)$
end for
```

The following operation deletes a reference from a sequence's level of abstraction. The operation *delete sequence* can be used to delete the sequence the reference refers to.

delete reference

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, reference $o_R \in O_R$

Body:

```
$E' \leftarrow \{e \in E \mid \iota(e) = (i, o) \wedge o = o_R\}$
$(I_l, <_l) \leftarrow f(l)$
for all $e \in E'$ do
  $\iota(e) \leftarrow \bot$
  $<_l \leftarrow <_l \setminus \{(e_1, e_2) \mid e_1 = e \vee e_2 = e\}$
end for
$E = E \setminus E'$
$O_R = O_R \setminus \{o_R\}$
```

3.3.8. Actions

Each local action consists of a start event and an end event on the same lifeline. Thus, both event orderings have to define insertion points on the same lifeline. Moreover, these insertion points have to reside in the same operand scope, *i.e.*, the insertion points belong to the same innermost operand or both belong to no operand.

add local action

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, actionBegin $ab = (ab_1, ab_2) \subseteq <$, actionEnd $ae = (ae_1, ae_2) \subseteq <$

```
Require:  – $\text{getInstance}(ab) = \text{getInstance}(ae)$
          – $\text{operandSet}(ab_1) \cup \text{operandSet}(ab_2) =$
            $\text{operandSet}(ae_1) \cup \text{operandSet}(ae_2)$
Body:     $o_A \leftarrow$ new local action
          $O_A \leftarrow O_A \cup \{o_A\}$
          $e_{AB} \leftarrow$ local action begin event for $o_A$
          $e_{AE} \leftarrow$ local action end event for $o_A$
          $E \leftarrow E \cup \{e_{AB}, e_{AE}\}$
          $\iota(e_{AB}) \leftarrow (\text{getInstance}(ab), o_A)$
          $\iota(e_{AE}) \leftarrow (\text{getInstance}(ae), o_A)$
          $(I_l, <_l) \leftarrow f(l)$
          $<_l \leftarrow \text{addToEventOrder}(e_{AB}, ab, <_l)$
          $<_l \leftarrow \text{addToEventOrder}(e_{AE}, ae, <_l)$
```

To delete a local action, the following operation removes the start and end event from the event set as well as from the event order and eventually the local action object.

delete local action

```
Input:  sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, local action $a \in O_A$
Body:   $E' \leftarrow \{e \in E \mid \iota(e) = (i, o) \wedge o = a\}$
        $(I_l, <_l) \leftarrow f(l)$
        for all $e_A \in E'$ do
          $\iota(e_A) \leftarrow \bot$
          $<_l \leftarrow <_l \setminus \{(e_1, e_2) \mid e_1 = e_A \vee e_2 = e_A\}$
        end for
        $E \leftarrow E \setminus E'$
        $O_A \leftarrow O_A \setminus \{a\}$
```

3.3.9. Continuations

A continuation has to cover all lifelines in the enclosing operand and has to be either the very first element in the operand or the very last. Thus, the operation to create a new continuation takes an operand and the information if it should be placed at the top or the bottom of the operand.

add continuation

```
Input:  sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, operand $o \in O_O$, mark $m \in$
        $\{\text{START}, \text{END}\}$
Body:   $o_{CO} \leftarrow$ new continuation
        $O_{CO} \leftarrow O_{CO} \cup \{o_{CO}\}$
        $(I_l, <_l) \leftarrow f(l)$
```

```
if $m = \text{START}$ then
  $E' \leftarrow \{e \in E_{OE} \mid \exists\, i \in I_l : \iota(e) = (i, o)\}$
else
  $E' \leftarrow \{e \in E_{OB} \mid \exists\, i \in I_l : \iota(e) = (i, o)\}$
end if
for all $e \in E'$ do
  $e_{CO} \leftarrow$ continuation event for $o_{CO}$
  if $m = \text{START}$ then
    $eo \leftarrow (\text{prev}(e), e)$
  else
    $eo \leftarrow (e, \text{next}(e))$
  end if
  $<_l \leftarrow \text{addToEventOrder}(e_{CO}, eo, <_l)$
  $\iota(e_{CO}) \leftarrow (\text{getInstance}(eo), o_{CO})$
end for
```

To delete a continuation, the operation removes the continuation object and the corresponding events on all lifelines.

delete continuation

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, continuation $o_{CO} \in O_{CO}$

Body:

```
$E' \leftarrow \{e \in E \mid \exists\, i \in I_l : \iota(e) = (i, o_{co})\}$
$(I_l, <_l) \leftarrow f(l)$
for all $e_{CO} \in E'$ do
  $\iota(e_{CO}) \leftarrow \bot$
  $<_l \leftarrow <_l \setminus \{(e_1, e_2) \mid e_1 = e_{CO} \vee e_2 = e_{CO}\}$
end for
$E \leftarrow E \setminus E'$
$O_{CO} \leftarrow O_{CO} \setminus \{o_{CO}\}$
```

3.3.10. General Orderings

General orderings can be used to put events into order that are not ordered yet. The message add and remove operations automatically use this mechanism to order new message events if needed. The following operation adds a new general ordering into the event order of a sequence's level of abstraction.

add general ordering

Input: sequence $s = (E, \iota, f) \in \text{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, event $e_1 \in E$, event $e_2 \in E$

Body:

```
$(I_l, <_l) \leftarrow f(l)$
$<_l \leftarrow <_l \cup \{(e_1, e_2)\}$
```

The operation to delete a general ordering has to assure that no regular event orderings are deleted. Thus, the first precondition requires that both events belong to different instances to maintain the total order of the lifelines and further that both events belong to different objects. The second condition maintains the order between message send and receive events of the same message object and prevents the ordering of operand events of the same fragment.

delete general ordering

Input: sequence $s = (E, \iota, f) \in \mathrm{Seqs}$, level $l \in \mathbb{N}_{\leq n}$, event $e_1 \in E$, event $e_2 \in E$

Require:

- both events belong to different instances and different objects, *i.e.*, $\iota(e_1) = (i_1, o_1), \iota(e_2) = (i_2, o_2)$: $i_1 \neq i_2, o_1 \neq o_2$
- there is direct path between e_1 and e_2 without $(e_1, e_2) \in <_l$

Body:

$(I_l, <_l) \leftarrow f(l)$
$<_l \leftarrow <_l \setminus \{(e_1, e_2)\}$

4. Implementation

This chapter describes the prototype implementation of the model as described in Chapter 3. The remainder of this chapter is organized into the following sections:

Architecture The architecture section describes the structure of the implementation, the relations between the components, and the basic principles that were used.

Realization The realization section presents the concrete technologies that were used and describes the implementation itself.

Evaluation The evaluation section presents a complexity analysis for the Petri net algorithms that were used to determine start and end states, to generate the level state machine, and to calculate the possible insertion points for new events.

4.1. Architecture

4.1.1. Data Storage

The information in the data model has to be saved persistently. A well-suited possibility to save structured information is a relational database. *Database management systems* (*DBMS*) allow efficient information queries and the usage of proven mechanisms such as database transactions, views, and triggers. Thus, the model transactions as mentioned in Section 3.3 can be mapped to database transactions. DBMS range from in-memory or file based database systems to multi-user client/server based database systems. Different DBMS are suitable for different use cases: File based database systems are fast, require little or even no administration and the database files can be version-controlled through a classical file based version control system. Client/Server based database systems are appropriate if many developers work on the same model, since the centralized database assures the ACID properties of transactions for all connections to the database. Most of the complex software systems nowadays are developed in an object-oriented language such as *Java*, *C++*, or *C#*. An object-relational mapping (O/R mapping) handles the connection between objects in the programming language and the tuples in the relational database. Furthermore, it abstracts from the concrete database queries and, thus, allows switching between different DBMS according to the desired use case. Consequently, no relational schema is given in this section, since it depends on the DBMS and is task of the O/R mapping system.

4.1.2. Model/View System

The prototype saves the model/view system in a relational database with help of an O/R mapping. Each element type in the model/view system is a class with a mapping description, which represents the names and tables of the class attributes. Figure 4.1 depicts the class structure and the connections between the classes of the data model element types (cf. Definition 3.1) and the sequence diagram views (cf. Definition 3.4). The mapping function ι of the data model is included in the events, since each event references the corresponding instance and object. Likewise, references between the corresponding objects realize the functions η that maps operands to fragments and σ that maps states to instances. The fragment type given by t and the operand constraint given by τ are attributes of the fragment and operand, respectively. Furthermore, the instance hierarchy with the master relation for inter-level instance refinement and the parent relation for intra-level instance refinement are references between the instances, namely masterInstance and parentInstance. A separate level object realizes one level of abstraction in a sequence. View objects represent sequence diagram views, which consist of view instances that reference the lifelines in the view, the start event, and the end event.

4.1.3. Prototype

The implementation consists of the model/view system to store the information and the model that contains the model operations to manipulate the model/view system. Figure 4.2 depicts the class structure of the prototype implementation. Several model operations use a callback to a function *askUser* to involve the developer if the model operation has to resolve ambiguities or if information is missing. An interface UserCallback represents the *askUser* function that is realized by the graphical user interface, namely UserInterface. The second interface ModelChangeListener allows any object to be informed if there was a change in the model. This listener mechanism is used by the user interface and the views to update their content. On the right side of the diagram is the model/view system, which is in detail shown by Figure 4.1, and the O/R mapping framework with connection to the database and the model/view system. The Model class has, besides the model operations from Section 3.3, methods for the model listeners, the callback, and—most important—for the model transactions. Before a series of model operations begins, a new transaction has to be initiated with the method startTransaction. This method starts a transaction in the O/R mapping framework and, thus, in the database. For that reason, all model operations can directly work on the objects and relations from the model/view system and the transaction mechanism encapsulates the change set. After all model operations have been executed, the transaction can be committed with commitTransaction, which makes the changes permanent or a rollback can be performed with rollbackTransaction, which undoes the changes.

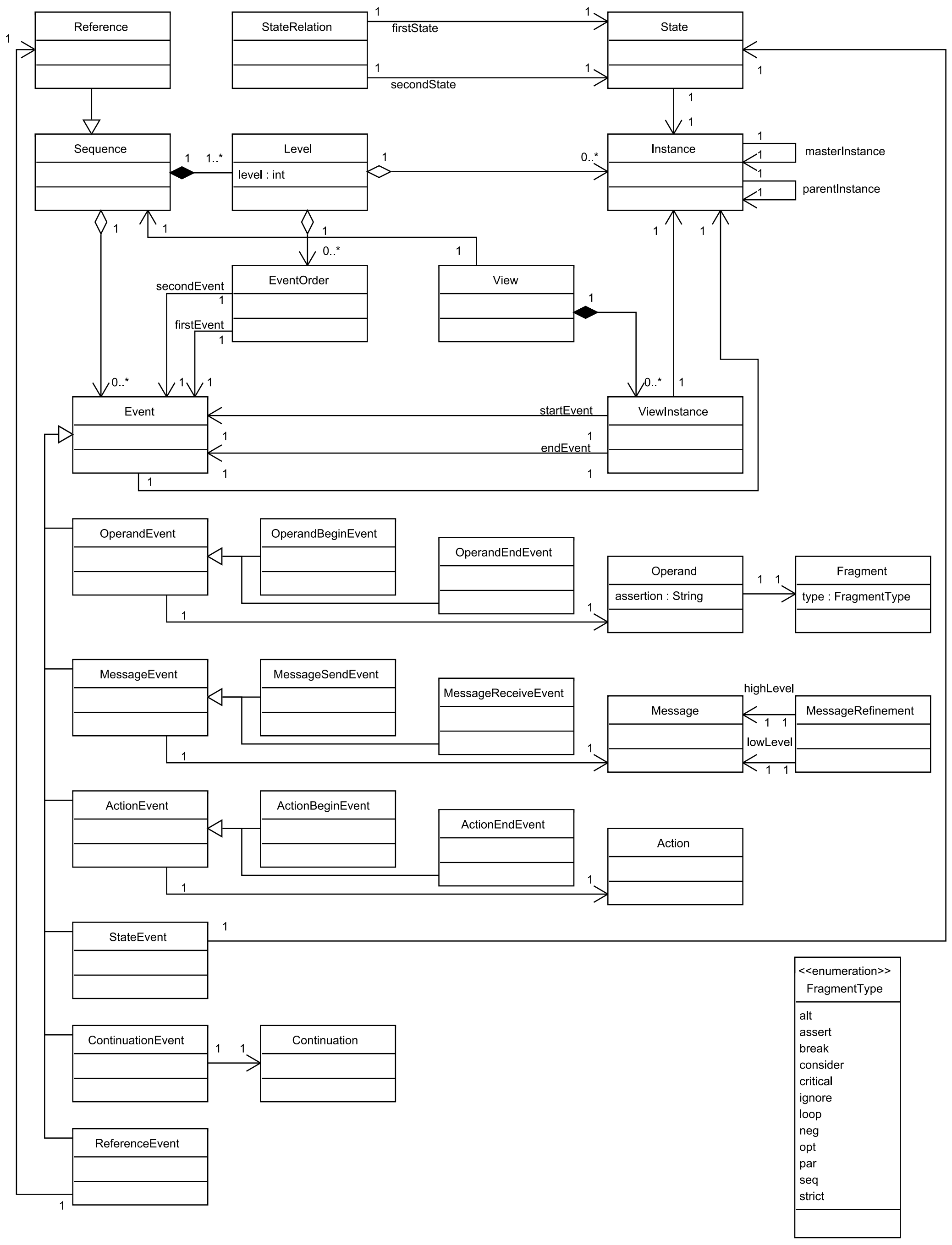


Figure 4.1.: Class diagram for the model/view system (cf. Definition 3.5).

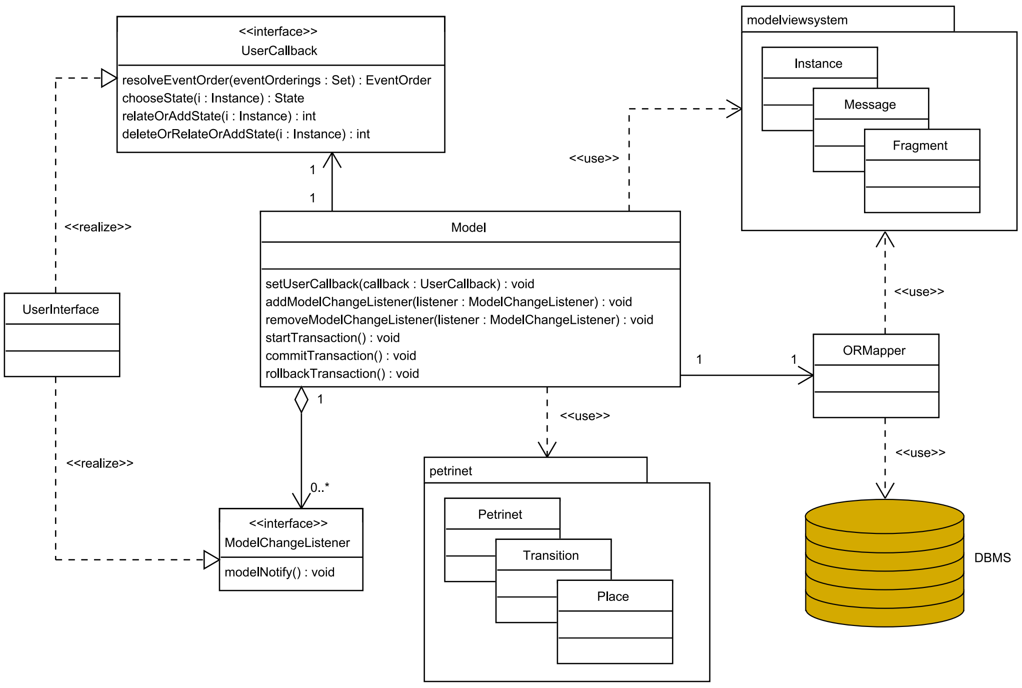

<<interface>>
UserCallback
resolveEventOrder(eventOrderings : Set) : EventOrder
chooseState(i : Instance) : State
relateOrAddState(i : Instance) : int
deleteOrRelateOrAddState(i : Instance) : int
modelviewsystem
Instance
Message
Fragment
<<use>>
1
1
<<realize>>
Model
setUserCallback(callback : UserCallback) : void
addModelChangeListener(listener : ModelChangeListener) : void
removeModelChangeListener(listener : ModelChangeListener) : void
startTransaction() : void
commitTransaction() : void
rollbackTransaction() : void
UserInterface
<<use>>
1
1
ORMapper
1
<<use>>
<<realize>>
petrinet
<<use>>
0..*
<<interface>>
ModelChangeListener
modelNotify() : void
Petrinet
Transition
Place
DBMS

Figure 4.2.: Class diagram for the implementation.

4.1.4. Petri Net

Several concepts in this thesis are based on Petri nets. For example, the algorithm that determines start state and end state combinations in a sequence diagram creates a Petri net with the states and their relations according to the sequence diagram structure. Afterwards the algorithm calculates the reachability graph of the created Petri net. Thus, the implementation has to support Petri nets and graphs. Figure 4.3 presents the class structure of the Petri net package, which also contains a graph representation. A Petri net consists of places and transitions, which the diagram reflects. Function 3.1 describes how to identify the set of valid insertion points in the refinement concept. In this function, a Petri net will be created and executed. During the execution the set of places is saved, because each place represents an event ordering in the underlying sequence diagram. Moreover, the last step of the mentioned function filters out places that represent a message object, *i.e.*, the place between the send event of a message and the corresponding receive event. For that reason, the place class has two specializations, namely EventOrderingPlace, which includes an event ordering, and MessagePlace, which can easily be filtered. The reachability graph is a directed graph and, thus, the class representation is straightforward. Chapter 2 and Chapter 3 present the Petri net algorithms.

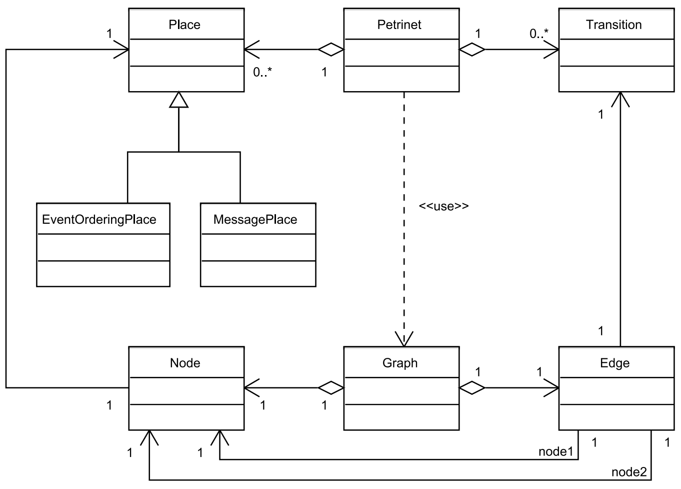


Figure 4.3.: Class diagram for the Petri net package.

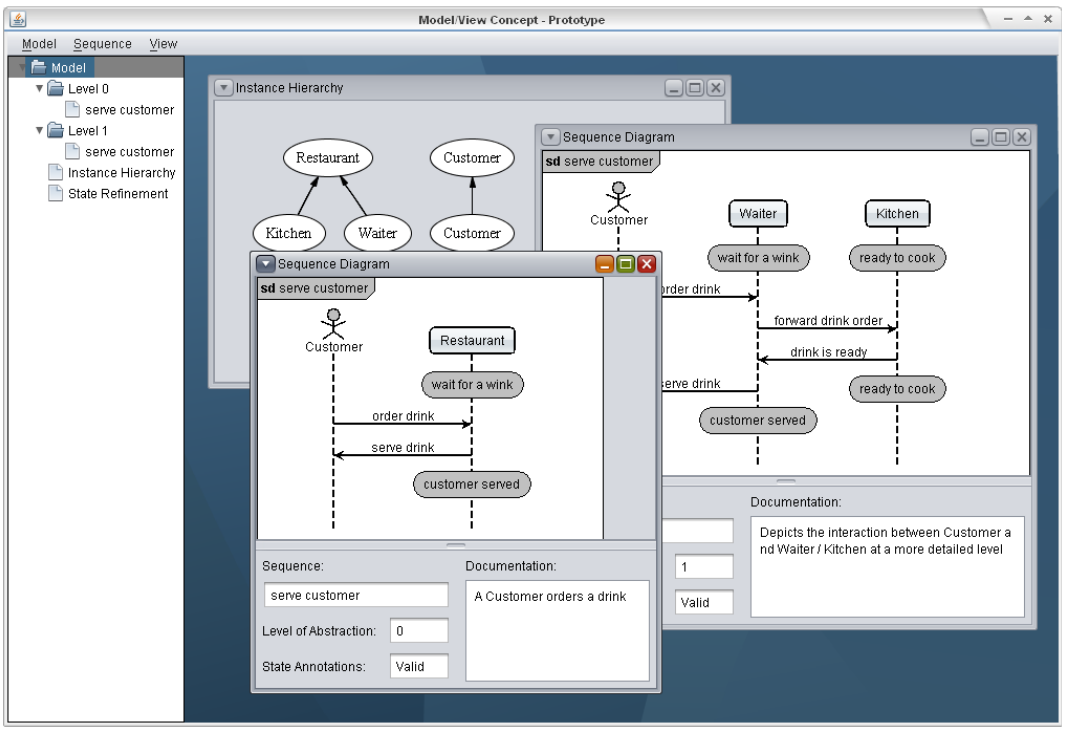
4.2. Realization

4.2.1. Technologies

The prototype is written in Java. At Philips PMS Java is the favored language for writing tools and tool extensions. Furthermore, with *Eclipse* [3] there is a comfortable development environment to write and test Java applications.
As mentioned in the previous section, the data storage in the prototype is realized with an O/R mapping framework. *Hibernate* [10] is such an O/R mapping and persistence framework that is available in native Java. It provides a special object-oriented query language to access database objects similar to SQL for relational databases. The prototype uses the small and fast HSQLDB [11]; a file based database system also written in Java. For visualization and layout of the graphs, *i.e.*, the instance hierarchy, the level state machine, and the state refinement, the prototype uses the *Graphviz* toolkit [7], which offers automatic graph layout and a Java interface to depict the graphs.
All algorithms and model operations are implemented straightforward to the definitions in Chapter 2 and Chapter 3.

4.2.2. Functionality

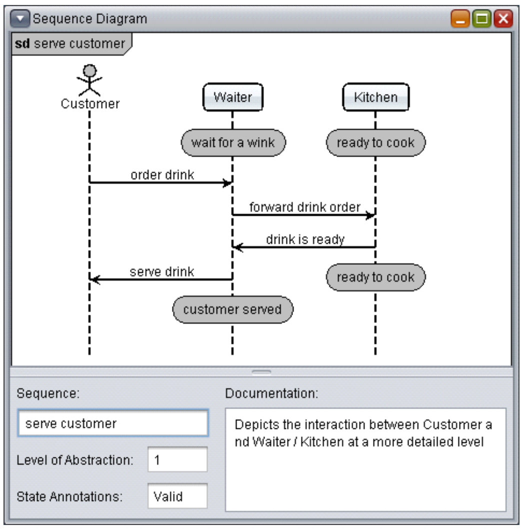
The implementation consists of the classes described in the previous section and of a graphical user interface to work with the model and the views. It is a standalone application and, thus, independent of any CASE-tool. The user interface allows to start with an empty model and to add the instance hierarchy, state refinement, the sequences with their abstraction levels, and to work with the objects and events in the sequence's levels of abstraction. Moreover, any amount of views can be defined for the sequences. Each view registers itself to the model and updates the view on model changes. The prototype implements all but derived views from Figure 3.2. A sequence diagram component, which is developed for this thesis, depicts the sequence diagram views. The sequence diagram layout will be the same as for the sequence diagrams in this thesis, since both are generated by the same component. The sequence diagram component generates an automatic layout for sequence diagrams and depicts the diagram according to the generated layout. An automatic layout reduces the diagram to the semantically important information given by the partial event order. Furthermore, it allows generating dynamic views that combine elements without compatible layout information defined previously by the developer. The layout algorithm tries to depict each event topmost on the corresponding lifeline and to draw all possible messages horizontal. The sequence diagram elements and their order can be defined within a XML file, which was done for all sequence diagrams of this thesis. Moreover, the component can be inserted into each Java application as part of the graphical user interface. The component can add different marker types on the lifelines that are clickable by the user to choose insertion points for new messages, actions, fragments and states. Figure 3.3 shows a sequence diagram with markers that present valid insertion points. Furthermore, each sequence diagram element can be colored to graphically highlight it. Figure 4.4 depicts a screenshot



Sequence: serve customer
Level of Abstraction: 0
State Annotations: Valid
Documentation: A Customer orders a drink

1
Valid
Documentation: Depicts the interaction between Customer a nd Waiter / Kitchen at a more detailed level

Figure 4.4.: Screenshot of the graphical user interface.



Sequence: serve customer

Level of Abstraction: 1

State Annotations: Valid

Documentation: Depicts the interaction between Customer and Waiter / Kitchen at a more detailed level

Figure 4.5.: Screenshot of a sequence diagram view.

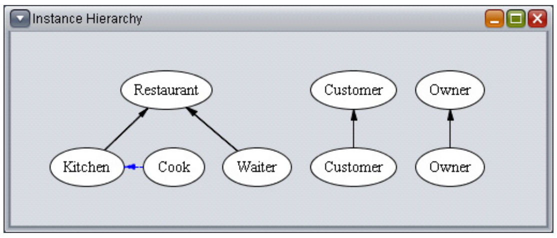


Figure 4.6.: Screenshot of the instance hierarchy view.

of the graphical user interface with three frames showing the same scenario on two levels of abstraction and the instance hierarchy. On the left side in the screenshot is a tree containing the views for each level, *i.e.*, sequence diagram views and the level state machine, and the views that belong to no concrete level such as the instance hierarchy and the state refinement view are below the last level. Figure 4.5 presents a sequence diagram view with the sequence diagram in the upper half and a panel with further information in the lower half. This panel contains a documentation field, the sequence name, the level of abstraction and a status field that depicts whether the current state annotations in the diagram are valid according to Definition 2.8. If these annotations are valid, the corresponding sequence will be used for generation of the level state machine. Figure 4.6 depicts a screenshot of the instance hierarchy view with two levels of abstraction. Analogous to Figure 2.6, the arrows point upwards to the master instances in case of inter-level refinement, *e.g.*, from the Waiter to the Restaurant, and at the same level to the parent instance in case of intra-level refinement, *e.g.*, from the Cook to the Kitchen. Figure 4.7 presents a level state machine for five sequences each with one start state combination and one end state combination, such as the diagram in Figure 4.5. The black bullet marks the initial state combination ⟨*Waiter.closed, Kitchen.closed*⟩. The last view is the state

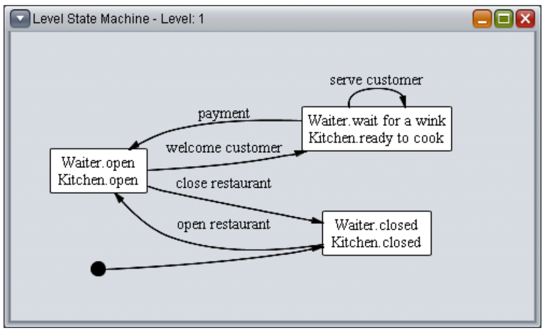


Figure 4.7.: Screenshot of the level state machine view.

refinement view, which is depicted by Figure 4.8. It presents all states for each instance grouped by a rectangle and the relations between the states according to Definition 2.9.

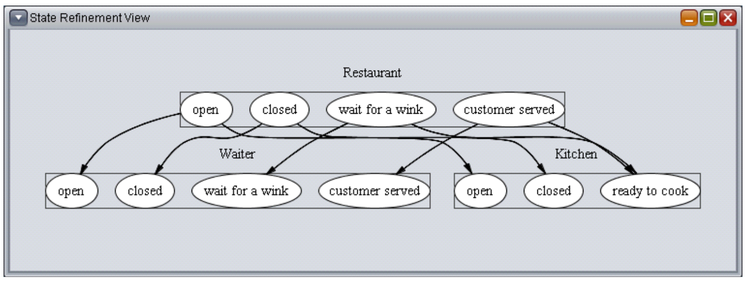


Figure 4.8.: Screenshot of the state refinement view.

4.3. Evaluation

4.3.1. Complexity

Petri Nets The Petri net definition (cf. Definition 2.5) presents a subclass of the standard Place/Transition nets, since each place can hold at most one token. This subclass is called *1-safe nets*, which contains Petri nets that are well-suited to represent logical conditions. Esparza [6] presents a good overview about decidability and complexity of Petri net questions. Most questions about the behavior of standard Place/Transitions nets are EXPSPACE-hard [6] and, thus, in particular in EXPTIME, which makes the algorithms unattractive for implementations. The limitation to 1-safe nets reduces the complexity class for the same questions to PSPACE-complete [6].

Level State Machine During the creation of the level state machine (cf. Section 2.2), all start state and end state combinations of all participating sequences have to be determined. The used Petri net representation is 1-safe, acyclic, and *conflict-free*, *i.e.*, each place has at most one outgoing transition. Due to these properties, the complexity is reduced once more to polynomial time. After all start and end state combinations were determined, the reachability graph of another Petri net results in the level state machine. This Petri net is just 1-safe and, thus, the calculation belongs to the complexity class PSPACE.

Petri Net Execution To determine the set of valid insertion points, *e.g.*, for a new message that should be integrated, a Petri net execution model results in the insertion points. This execution model consists of three algorithms described in Section 3.2. The backtracking set is calculated in $\mathcal{O}(|P| + |T|)$, where P is the set of places and T is the set of transitions in the Petri net, since each path has to be visited once in the worst case. The execution of the backtracking set has to execute each transition of the set if it is enabled in the current Petri net marking. Let $p_{\max}$ denote the maximal number of incoming and outgoing places for a transition. The backtracking set is executed in $\mathcal{O}(\frac{n\cdot(n+1)}{2} \cdot p_{\max})$ where n is the cardinality of the backtracking set. The third algorithm executes the Petri net from a given marking either until a certain transition is the only enabled transition or until the end places are reached. Thus, in the worst case all transitions are fired once and the algorithm calculates the set of insertion points in $\mathcal{O}(|P| + |T|^2 \cdot p_{\max})$.

Petri net BFS The classification of insertion points uses two breadth first searches based on a Petri net (cf. Algorithm 3.5). A breadth first search has to visit in the worst case each place and transition once, *i.e.*, $\mathcal{O}(|P|+|T|)$. After both BFS created a set of places, the cut of these two sets is the result of Algorithm 3.5. Thus, the whole complexity is $\mathcal{O}(2 * (|P| + |T|) + |P|)$. If the first ordering constraint is not defined for the algorithm, the breadth first search has to be done for each refining instance, *i.e.*, with m as the number of refining instance, the runtime complexity is $\mathcal{O}(m * (|P| + |T|))$. If the second ordering constraint is not defined, the complexity remains the same.

5. Related Work

This chapter presents selected related work and points out relations to this thesis. The concepts of this thesis touch several topics:

- Refinement and consistency of sequence diagrams
- Model/View concept for sequence diagrams
- Constructive modeling
- Interactive resolution of ambiguities while integrating changes

Comparisons and related work for refinement and consistency of sequence diagrams can be found in the related work sections of Ohlhoff [21] and Lischke [17]. In general, most of the concepts that are state-of-the-art try to find inconsistencies between existing views instead of preventing inconsistent operations.

UML and View Integration

Egyed presented several publications [5] including his dissertation [4] on the topic of integrating UML views. In that work, he described problems that may arise between different UML views and presented a framework that exposes inconsistencies between these views. Therefore, Egyed identified a list of 51 inconsistency types that are classified in the abstract, the generic, and the behavioral dimension. Like in this thesis, he distinguishes the development process into different levels of abstraction, *i.e.*, a refinement relation.
The integration process as described in Egyed's dissertation is separated into three disciplines:

- Mapping: identifies elements that describe overlapping information in different views;
- Transformation: extracts and manipulates model elements to support differentiation;
- Differentiation: analyzes the model and identifies inconsistencies.

The integration framework, however, checks views retroactively and, thus, reveals existing inconsistencies rather then avoid them constructively. Even if the list of inconsistencies covers some consistency terms for sequence diagrams, at state of the dissertation the framework just supports class and object diagrams.

Refinement and Consistency

The sequence diagram refinement concept used in this thesis is based on the diploma thesis of Ohlhoff [21]. He introduced a set of syntactical refinement rules based on the principle that the *same things should look the same*, which means that the same behavior in a refining diagram has to be realized by the same syntactical elements. The advantage of a syntactical approach is the high performance of validating these rules, which allows constructive modeling. In this thesis, the rules were adapted and realized even more rigorously. The original rules allow different operand ordering for fragments in case of par fragments or alt fragments, since the set of traces will remain the same. The definitions here require the same operand order in each refinement step. Another mentionable difference is the handling of coregions; Ohlhoff included the handling of coregions into the partial order that is used to check the message order consistency. This thesis does not uses a special mechanism for coregions, since coregions are according to the UML 2.0 specification a shorthand for a par fragment where each event belongs to an own operand. Thus, coregions are handled in the same way as each other fragment. Moreover, Ohlhoff introduced intra-level refinement diagrams (cf. Section 2.1) to support iterative modeling of diagrams in the same level of abstraction. This concept is maintained for instances, because it contains a restriction of the valid messages. Intra-level refined instances can only be accessed by the parent instance, which may interact as a proxy for other instances. The corresponding intra-level refinement concepts for messages and fragments are covered by the model/view concept, since each sequence of the model contains all instances, messages, and fragments that may be used in a level of abstraction. Thus, intra-level refinement diagrams can be achieved by defining a view that contains the appropriate instances.

Level State Machine

Section 2.2 presented the level state machine as concept to check the overall consistency between sequence diagrams of the same level of abstraction. That section is founded on the diploma thesis of Lischke [17]. She presented several consistency terms for single sequence diagrams, for sequence diagrams of the same level of abstraction and for refinement relations between sequence diagrams of different abstraction levels. Due to the supported CASE tool, Lischke used UML 1.4 and furthermore neglected coregions. For single sequence diagrams, she introduced the usage of state invariants to express the start and end states of a sequence diagram and presented a consistency definition for the state annotation. Sequence diagrams in UML 1.4 have only one start and end state combination. This thesis picked up the idea of annotating sequence diagrams and extended it to the possibilities of UML 2.0. The new combined fragments lead to possibly many state combinations for a single sequence diagram as described in Section 2.2. The level state machine as used in this thesis is the same as defined by Lischke, but the generation of the level state machine is different. Here, the level state machine is the reachability graph of a Petri net, which

is presented in Section 2.2.
Moreover, Lischke presented a refinement concept for sequence diagrams based on traces of master and refinement diagrams. That approach is applicable for the made restrictions, which make the amount of traces for one sequence diagram controllable. Instances in successive abstraction levels can remain the same or can be substituted by sub-instances that model the internal behavior in more detail. States are refined according to the instances and, thus, there is a one-to-many relation from higher to lower levels of abstraction. This thesis uses a many-to-many relation (cf. Section 2.3) that allows more freedom to model the instances. Even if it is a weak relation, Chapter 6 outlines a consistency notion as further work.

6. Conclusion and Further Work

Describing intended and unintended behavior in case of sequence diagrams is a field-tested communication medium to address different stakeholders. It is applicable throughout all disciplines of a software development process such as the Rational Unified Process. Furthermore, consistent refinement from the requirements to the implementation discipline helps to handle the complexity problem of today's software. Due to the large number of diagrams for every refinement step, the overall consistency is hard to maintain. Thus, the overall consistency is a new part of the development complexity.

This thesis defines a constructive model/view concept, with operations that maintain the efficient structural refinement relationships introduced by Ohlhoff [21] (cf. Section 2.1). The model is well-suited to handle redundant information and, thus, to manage a large number of diagrams. The refinement concept was adapted to the constructive approach, which allows being more strict, *i.e.*, operands have to appear in the same order in all diagrams. In addition, the model/view concept supports the completeness of each refinement level introduced by Lischke [17], which is described and extended in Section 2.2. The level state machine concept [17] was extended to support the variety of start and end vectors arising through combined fragments that were introduced in UML 2.0.

When a new element is added to an existing sequence diagram, the refinement concept describes which changes involve other sequence diagrams that refer to the same scenario on another refinement level. These changes often cannot be resolved automatically, since the possibly high number of valid integrations. The presentation of only valid integrations to the developer allows comfortable ambiguity resolving in other refinement levels. Chapter 3 presented how a Petri net representation of the sequence diagram identifies valid integrations, by using a special execution model. Furthermore, Chapter 3 describes how these possibilities can be classified according to the quality of the integration, *i.e.*, if traces of the master diagram are obtained. Beside sequence diagrams, other views can be used to modify and interact with the model. For example, the developer can structure the refinement of instances with an instance hierarchy view. A prototype implementation of the model/view concept is presented in Chapter 4. The database foundation of the data model allows system development separately for each developer with a local database or in a team with a central database server.

Even if the objectives that were mentioned in the introduction (cf. Section 1.3) are fulfilled, several new interesting concepts arise from the constructive model/view principle. Additionally, the tasks to turn the prototype into a well usable development environment remain. The following section briefly describes further work.

6.1. Concept

This section outlines some conceptual extensions and new concepts that benefit from the model/view concept.

6.1.1. Hierarchical Scenario Refinement

A new consistency notion can be introduced called *Hierarchical Scenario Refinement*, which benefits from the state refinement concept (cf. Section 2.3).
The idea behind this consistency notion is that every scenario at a given level should be executable at every lower level in the instance hierarchy. Thus, the set of scenarios is monotonically increasing from top to bottom. According to the state refinement, every scenario runnable from state s at a given level should be executable at every lower level from any refined states of s. That is every scenario should be executable from every state combination reflecting the original start state of the scenario.
Furthermore, each scenario at a given level ending in state t has to end at any lower level in at least one refined state of t. In contrast to the first condition a scenario does not need to end in all refined states of t, because a subsequent scenario has to be executable in all refined states and in particular in t.
The following example revisits the instance hierarchy from Figure 2.16 and the state relation defined for Figure 2.16 in the same section. Assume that for the Restaurant a scenario named *close restaurant* was defined at level 0, which starts from the state *open* and ends in the state *closed*. The corresponding scenario at level 1 has to start in the state combinations:

- ⟨*Kitchen.open, Waiter.present*⟩
- ⟨*Kitchen.open, Waiter.pause*⟩
- ⟨*Kitchen.cleaning, Waiter.present*⟩
- ⟨*Kitchen.cleaning, Waiter.pause*⟩

The end state combination for the *close restaurant* scenario at abstraction level 1 is ⟨*Kitchen.closed, Waiter.absent*⟩.

6.1.2. Using Change Logs for Merges / Differential Comparison

A common practice in software development is that every developer has an own copy of the model or source code from the system under development to work independent from other developers. The source of this copy often is a version control system such as *Subversion*, *Microsoft Visual SourceSafe*, or *Rational ClearCase*. After the work is done, the developer has to integrate the changes into the version control system. This task is easy, if the software is separated into independent small parts and the developer, who integrates the changes, is the only one who made changes in the corresponding part, because then the part can be overwritten with the new part from

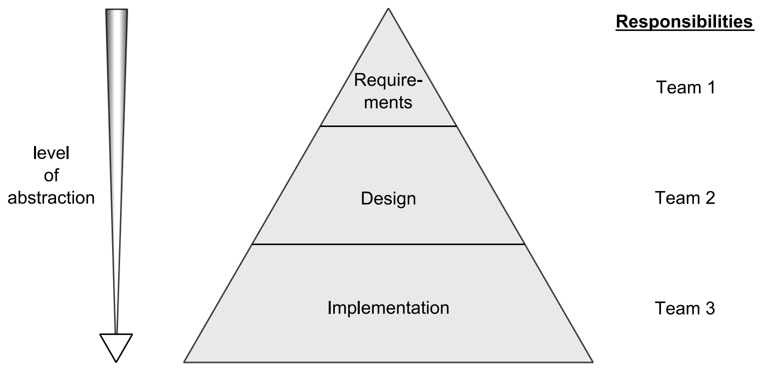


Figure 6.1.: Different responsibilities in different levels of abstraction.

the developer. In case that the affected parts were also changed by other developers, the integration becomes nontrivial. These merges of different model versions are difficult, because the changes made by different developers may be incompatible, *e.g.*, one developer added the new functionality to sequence diagram a and another developer moved the existing functionality from sequence diagram a to sequence diagram b and deleted sequence diagram a.
Each model operation results in a *Change Log* or *Delta List*, which is a list of changes made since a given version, *i.e.*, since the last model integration. A Change Log in context of this model / view principle is the operation called by the user, the parameters of the operation and the integration decisions the user made, rather then the actual model changes after a successful operation. These change sets are useful for comparing two models on higher level than comparing visual differences or textual representations. Furthermore, a model integrator that should combine two models a and b can start from the point where both models were equal and has to find a good sequence of applying both change logs. Since each change log entry contains the former integration decisions, the integrator can reuse these decisions, if the chosen possibility is still a valid integration.

6.1.3. Responsibilities for different levels of abstraction

In large software projects, different roles are responsible for different levels of abstraction. For example, a requirement team manages the requirement levels, a software architect creates the design level, and the software developers are responsible for the remaining implementation levels. If a role changes a diagram in a level that he/she is responsible for and the consequence is that a diagram has to be changed that belongs to another responsibility level, the question is how to handle this. Thus, the concept introduced in this thesis can be extended to support responsibilities for different refinement levels or even responsibilities for different components or sub trees of the instance hierarchy.
Since model operations are transactions, an operation which has impact on another, higher responsibility scopes can be saved and given over to the corresponding role,

who can decide to allow the model operation, *i.e.*, to commit the transaction or to disallow it, *i.e.*, to rollback the transaction. Furthermore, changes on a higher level, which have an impact on a lower levels in a different responsibility scope, the appropriate role can be informed about the changes, *e.g.*, changes can be highlighted in the corresponding diagram.
Every model operation has to check whether the executed changes belong to the same responsibility scope and a transaction can only be committed if all changes are in the same scope or if the appropriate roles have allowed the change.

6.1.4. Hierarchical States

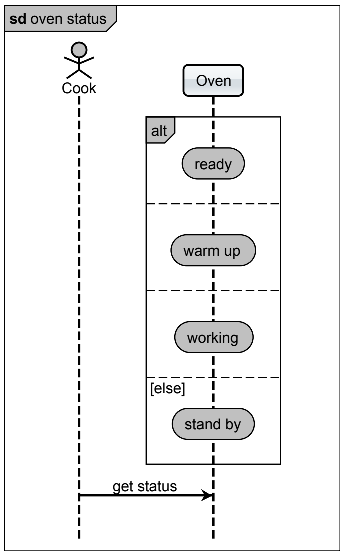
Chapter 2 describes the annotation of lifelines in a sequence diagram with state invariants to express the possible start and end states of the corresponding sequence. If a component might start or end in a set of states, an alt fragment in the corresponding sequence diagram can express that. Each operand in the alt fragment represents one desired start/end state. Imagine a use case where the scenario represents a status request that can occur in a large number of state combinations. The corresponding sequence diagram will be cluttered and hard to maintain, since the large number of operands and each new state has to be worked in as another operand. A way to overcome this problem is the introduction of hierarchical states. This hierarchy is independent from the state refinement concept presented in Chapter 2. It introduces new pseudo states for a component. Each pseudo states represents a set of states already defined for that component, which are hierarchically below the pseudo state. For example, consider a pseudo state *on* that contains the states *ready*, *warm up*, *working*, and *stand by*. Figure 6.2 depicts two diagrams, both with the beginning of a sequence where the Cook wants to get the status of the Oven. Figure 6.2(a) shows the sequence with the enumeration of all states. In comparison, Figure 6.2(b) shows the sequence with a hierarchical pseudo state, which is a strong simplification of the diagram.
In addition to hierarchical states, other pseudo states such as *previous state* or *unchanged* can highly simplify the diagrams. Furthermore, those pseudo states also introduce the concept of history from state charts into sequence diagrams. History states in state charts allow resuming of a complex state with the last active configuration of that state.

6.2. Implementation

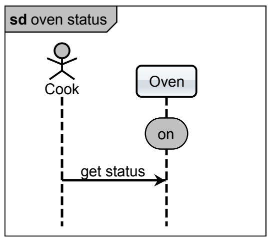
The following section presents several ideas and extensions towards an implementation of the concepts presented in this thesis.

6.2.1. Integration into a commercial CASE Tool

In order to transfer the concepts presented in this thesis to the day-by-day development process, there has to be support for a rich CASE tool with the corresponding



(a) An alt fragment expresses multiple start states.



(b) Syntactical simplification through a hierarchical pseudo state.

Figure 6.2.: alt fragment for multiple start states in comparison to a hierarchical pseudo state *on*.

tool-chain, *e.g.*, version control system, project and requirement tracking. The corresponding CASE tool has to be highly customizable; since the consistency definitions presented here necessitate that several operations can be refused or modified to enforce the overall consistency. Eclipse [3] based CASE tools, such as the *Rational Systems Developer* (*RSD*) [12], often have a rich *Application Programming Interface* (*API*) to extend the tools to customer needs. The Eclipse workbench consists of views, perspectives, editors and wizards, which can be provided by different plug-ins, thus, special views such as the level state machine can be provided by an own plug-in.

6.2.2. Semantic Plug-Ins

The integration of new information into other levels of abstraction is often an interactive process, if there is more than one valid integration. The introduction of Chapter 3 presented the integration of a new message into the next more detailed level of abstraction and presented the different insertion points for the receive event of the new message. These insertion points were of different integration quality. Two of the insertion points in the example may lead to inconsistent traces of master and refinement diagram, which was resolved with a general ordering that relates the receive event with the previous event according to the master diagram. In a software development process, several integration possibilities might be excluded through coding conventions or the actual application and their constraints. These restrictions of valid options can be achieved with semantic plug-ins. The prototype from Chapter 4 used a user callback interface, namely UserCallback, for each decision that the user has to take. A semantic plug-in in that architecture is an implementation of the interface UserCallback, which acts as proxy between the graphical user interface and the model. It gets the set of valid choices from the model, filters this input according to the coding conventions, and presents the filtered inputs to original implementation of the interface.

A. Bibliography

[1] Artisan Software. UML Modeling Tools for Real-time Embedded Systems Modeling and Systems Engineering, 2007. `http://www.artisansw.com/`.

[2] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[3] Eclipse. An Open and Extensible IDE, 2007. `http://www.eclipse.org/`.

[4] Alexander Egyed. Heterogeneous View Integration and its Automation. Phd dissertation, University of Southern California, Faculty of the Graduate School, August 2000.

[5] Alexander Egyed. Integrating Architectural Views in UML. Technical Report USC-CSE-99-514, Center for Software Engineering, University of Southern California, 1999. URL `http://sunset.usc.edu/publications/TECHRPTS/1999/usccse99-514/usccse99-514.pdf`.

[6] Javier Esparza. Decidability and complexity of petri net problems - an introduction. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, pages 374–428, London, UK, 1998. Springer-Verlag. ISBN 3-540-65306-6.

[7] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1234, 2000. ISSN 00380644. URL `http://www.research.att.com/sw/tools/graphviz/GN99.pdf`.

[8] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[9] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine.* Springer-Verlag New York, Inc., 2003.

[10] Hibernate. Relational Persistence for Java and .NET, 2007. `http://www.hibernate.org/`.

[11] HSQLDB. Open-source Java Database, 2007. `http://www.hsqldb.org/`.

[12] IBM. Rational Systems Developer (RSD), 2006. `http://www.ibm.com/developerworks/rational/products/rsd/`.

[13] IBM. Rational Unified Process, 2007. `http://www-306.ibm.com/software/awdtools/rup/`.

[14] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC99). Geneva, 1999.

[15] Gleen E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming (JOOP)*, 1(3):26–49, 1988.

[16] Dean Leffingwell. Calculating Your Return on Investment from More Effective Requirements Management. Whitepaper, Rational Software Corporation, 1997.

[17] Andrea Lischke. Consistency Checking of Sequence Diagrams. Diploma thesis, Brandenburgische Technische Universität Cottbus, September 2005.

[18] Björn Lüdemann. Synthesis of human-readable Statecharts from Sequence Diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2005.

[19] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246, New York, NY, USA, 1981. ACM. doi: http://doi.acm.org/10.1145/800076.802477.

[20] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[21] André Ohlhoff. Consistent Refinement of Sequence Diagrams in the UML 2.0. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, November 2006. `http://rtsys.informatik.uni-kiel.de/~rbiblio/downloads/theses/aoh-dt.pdf`.

[22] Olaf Kluge. Petri Nets as a Semantic Model for Message Sequence Chart Specifications. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, pages 138–147, 2002.

[23] Philips Medical Systems (PMS). PMS Homepage, 2007. `http://www.medical.philips.com/de/`.

[24] Ragnhild Kobro Runde and Øystein Haugen and Ketil Stølen. How to transform UML neg into a useful construct. In *Proceedings of the Norwegian Informatics Conference (Norsk Informatikkonferanse)*, 2005.

[25] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994. ISBN 0-471-59917-4.

[26] The Object Management Group. UML Homepage, 2007. `http://www.uml.org/`.

B. Pseudocode

This appendix contains the pseudocode for the identification of events in different levels of abstraction, for several model operations, and for several small commonly used auxiliary operations. These operations are in the appendix, because they are not important to understand the concepts from the corresponding sections. Moreover, the ideas are explained in the thesis or an abstract version of the pseudocode can be found in the corresponding section.

B.1. Identify events in the next/previous sequence level

Section 3.2 describes the identification of events between different levels of abstraction without giving a concrete algorithm. This section provides the two described algorithms for identifying events according to the next more abstract or the next more detailed level of abstraction. These are two different algorithms, since the amount of identifiably events depends on the identification direction (cf. Section 3.2).

B.1.1. identify events downwards

Let $s = (E, \iota, f)$ be a sequence and $e \in E$ an event in s. The following operation identifies the given event e according to the next more detailed level of abstraction. The direction parameter defines whether the next event, incipient by e, should be searched upwards or downwards according to the lifeline order.

```
identifyDownwards (sequence $s = (E, \iota, f) \in \text{Seqs}$, event $e \in E$, direction $d \in \{\text{UP}, \text{DOWN}\}$)
  $e' \leftarrow \text{findDownwards}(s, e, d)$
  $e'' \leftarrow \text{identDownwards}(s, e')$
  return $e''$
```

The operation is separated into two operations: *findDownwards* to find the next event in the same level of abstraction as e that can be identified with an event of the next more detailed level and *identDownwards* that uses the refinement concepts to return the corresponding event in the adjacent level of abstraction. This split-up makes the whole operation easier to understand and allows other model operations to access the interim result, *i.e.*, the found event that has a corresponding event in the adjacent level of abstraction.

findDownwards (sequence $s = (E, \iota, f) \in \mathrm{Seqs}$**, event** $e \in E$**, direction** $d \in \{$ UP, DOWN$\}$**)**

```
  $e' \leftarrow e$
  while $e' \neq \bot$ do
    if $d = \mathrm{DOWN}$ then
      $e' \leftarrow \mathrm{next}(e')$
    else
      $e' \leftarrow \mathrm{prev}(e')$
    end if
    if $e' \in E_{MS} \cup E_{MR}$ then
      return $e'$
    end if
    if $e' \in E_{OB} \cup E_{OE}$ then
      return $e'$
    end if
  end while
  return $\bot$
```

identDownwards (sequence $s = (E, \iota, f) \in \mathrm{Seqs}$**, event** $e \in E$**)**

```
  $(i, o) \leftarrow \iota(e')$
  $e' \leftarrow \bot$
  if $e \in E_{MS}$ then
    $m \leftarrow G(o)$
    $\{e'\} \leftarrow \{e_s \in E_{MS} \mid \iota(e_s) = (i, m)\}$
  end if
  if $e \in E_{MR}$ then
    $m \leftarrow G(o)$
    $\{e'\} \leftarrow \{e_s \in E_{MR} \mid \iota(e_s) = (i, m)\}$
  end if
  if $e' \in E_{OB}$ then
    $I' \leftarrow \{i' \in I \mid \mu(i') = i\}$
    $E' \leftarrow \{e_o \in E_{OB} \mid \iota(e_o) = (i', o) \wedge i' \in I'\}$
    $e' \leftarrow$ any element from $E'$
  end if
  if $e' \in E_{OE}$ then
    $I' \leftarrow \{i' \in I \mid \mu(i') = i\}$
    $E' \leftarrow \{e_o \in E_{OE} \mid \iota(e_o) = (i', o) \wedge i' \in I'\}$
    $e' \leftarrow$ any element from $E'$
  end if
  return $e'$
```

B.1.2. identify events upwards

Let $s = (E, \iota, f)$ be a sequence and $e \in E$ an event in s. Analogous to *identifyDownwards*, the following operation identifies the given event e in the next more abstract level. Again, the direction parameter defines whether the next event, incipient by e, should be searched upwards or downwards according to the lifeline order.

identifyUpwards (sequence $s = (E, \iota, f) \in \text{Seqs}$**, Event** $e \in E$**, direction** $d \in \{\text{ UP, DOWN}\}$**)**

```
  $e' \leftarrow \text{findUpwards}(s, e, d)$
  $e'' \leftarrow \text{identUpwards}(s, e')$
  **return** $e''$
```

The operation is separated into two operations: *findUpwards* to find the appropriate event and *identUpwards* that uses the refinement concepts to return the corresponding event in the adjacent level of abstraction. This split-up makes the whole operation easier to understand and allows other model operations to access the interim result, *i.e.*, the found event that has a corresponding event in the adjacent level of abstraction.
In comparison to *findDownwards*, the operation *findUpwards* is more complex, since not all events of the current level of abstraction have a corresponding event in the next more abstract level (cf. Section 3.2).

findUpwards (sequence $s = (E, \iota, f) \in \text{Seqs}$**, event** $e \in E$**, direction** $d \in \{\text{ UP, DOWN}\}$**)**

```
  $(i, o) \leftarrow \iota(e)$
  $\{k\} \leftarrow \{k \in \mathbb{N}_{\leq n} \mid i \in I^k\}$
  $(I_{k-1}, <_{k-1}) \leftarrow f(k-1)$
  $I' \leftarrow \{i' \in I \mid \mu(i') = \mu(i)\}$
  $e' \leftarrow e$
  **while** $e' \neq \bot$ **do**
    **if** $d = \text{DOWN}$ **then**
      $e' \leftarrow \text{next}(e')$
    **else**
      $e' \leftarrow \text{prev}(e')$
    **end if**
    $(i, o) \leftarrow \iota(e')$
    **if** $e' \in E_{MS} \cup E_{MR}$ **then**
      **if** $e' \in E_{MS}$ **then**
        $\{e_M\} \leftarrow \{e_M \in E_{MR} \mid \exists\, i' \in I : \iota(e_M) = (i', o)\}$
      **else**
        $\{e_M\} \leftarrow \{e_M \in E_{MS} \mid \exists\, i' \in I : \iota(e_M) = (i', o)\}$
      **end if**
      $(i', o') \leftarrow \iota(e_M)$
      // If the corresponding send/receive event for $e'$ belong to same instance,
      // further search is required. Otherwise the event can be identified.
      if $i' \in I'$ then
        $e_1 \leftarrow$ findUpwards$(s, e', d)$
        $e_2 \leftarrow$ findUpwards$(s, e_M, d)$
        if $e_1 = \perp$ then
          return $e_2$
        else if $e_2 = \perp$ then
          return $e_1$
        else if identUpwards$(s, e_1) <^{k-1}$ identUpwards$(s, e_2)$ then
          return $e_2$
        end if
        return $e_1$
      else
        return $e'$
      end if
    end if
    if $e' \in E_{OB} \cup E_{OE}$ then
      // Determine the set of instances that are covered by $o$
      $I'' \leftarrow \{i' \in I_k \mid \exists\ e'' \in E : \iota(e'') = (i', o)\}$
      // If $o$ covers at least one instance that has another parent,
      // the event $e'$ can be identified
      if $I'' \cap I' \neq \varnothing$ then
        return $e'$
      end if
    end if
  end while
  return $\perp$
```

```
identUpwards (sequence $s = (E, \iota, f) \in$ Seqs, event $e \in E$)
  $(i, o) \leftarrow \iota(e')$
  $e' \leftarrow \perp$
  if $e \in E_{MS}$ then
    $m \leftarrow G^{-1}(o)$
    $\{e'\} \leftarrow \{e_s \in E_{MS} \mid \iota(e_s) = (\mu(i), m)\}$
  end if
  if $e \in E_{MR}$ then
    $m \leftarrow G^{-1}(o)$
    $\{e'\} \leftarrow \{e_s \in E_{MR} \mid \iota(e_s) = (\mu(i), m)\}$
  end if
  if $e' \in E_{OB}$ then
    $I' \leftarrow \{i' \in I \mid \mu(i') = i\}$
    $\{e'\} \leftarrow \{e_o \in E_{OB} \mid \iota(e_o) = (\mu(i), o)\}$
```

```
**end if**
**if** $e' \in E_{OE}$ **then**
    $I' \leftarrow \{i' \in I \mid \mu(i') = i\}$
    $\{e'\} \leftarrow \{e_o \in E_{OE} \mid \iota(e_o) = (\mu(i), o)\}$
**end if**
**return** $e'$
```

B.2. Model Operations

This section contains complete versions of the model operations *add message* and *add operand.* Section 3.3 presented only abstract versions of these operations and explained the ideas that are implemented by these operations.

Temporary Views Each insert operation possibly has to make changes to many levels of abstraction and therefore has to check for valid insertion possibilities in those levels. This is achieved by a Petri net execution that represents a subset of the events, which appear in that level. The source of the Petri net transformation is a set of instances and for each instance a start and an end event. This source can be an existing sequence diagram view or a newly generated temporary view. Besides being the source of the Petri net transformation, the temporary view will be presented to the developer if the amount of valid insertion points requires a choice between at least two different possibilities.
There are different possibilities to generate a temporary view. Firstly, the whole sequence in the current level of abstraction can serve as temporary view, since all related model information is in that view. A disadvantage might be the complexity of the view, because the developer might have modeled the containing behavior through several small views and has to orient in the new view. Secondly, the current view might serve as temporary view, since it is known to the developer and belongs to the current operation. A disadvantage might be that the view is too restricted to cover the necessary diagram elements and if the operation has to be integrated into another level of abstraction, a new view has to be created that correspond to the actual view. A third possibility might be a new constructed view that covers exactly the elements, which are necessary to determine the insertion points for the current operation. Since that view is completely new, it is hard to preserve the mental map of the sequence. Finally, none of the listed possibilities is free of disadvantages and, thus, it is up to the implementation to choose the appropriate view or allow the developer to switch between several dynamic views.

The following operation adds a new message to the model and integrates the change into all necessary levels of abstraction. The function *traverseUpwards* recursively integrates the new message into all more abstract levels and *traverseDownwards* integrates the new message into all more detailed levels, if necessary. An abstract version of this operation can be found in Section 3.3.

add message

Input: sequence $s \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, type $t \in \{$ async, sync, answer, create, destroy $\}$, send order $so \in <$, receive order $ro \in <$

Require:

- $so = (e, e')$: e, e' belong to the same instance and e, e' are subsequent according to the event order, *i.e.*, $e < e' \wedge \forall\ e'' \in E : e \neq e'' \neq e' \Rightarrow e'' < e \vee e' < e''$
- $ro = (e, e')$ analogous to so
- ro is a valid message receive order according to so, *i.e.*, the event order $<$ is acyclic after inserting the new message events.

Body:

```
function addMessageEvents(sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, message $m \in O_M$, send order $so \in <$, receive order $ro \in <$)
    $e_{MS} \leftarrow$ message send event for $m$
    $e_{MR} \leftarrow$ message receive event for $m$
    $\iota(e_{MS}) \leftarrow (\text{getInstance}(so), m)$
    $\iota(e_{MR}) \leftarrow (\text{getInstance}(ro), m)$
    $E_{MS} \leftarrow E_{MS} \cup \{e_{MS}\}$
    $E_{MR} \leftarrow E_{MR} \cup \{e_{MR}\}$
    $(I_l, <_l) \leftarrow f(l)$
    $<_l \leftarrow \text{addToEventOrder}(e_{MS}, so, <_l)$
    $<_l \leftarrow \text{addToEventOrder}(e_{MR}, ro, <_l)$
    $<_l \leftarrow <_l \cup \{(e_{MS}, e_{MR})\}$
end function

function determineEventOrderUp(sequence $s = (E, \iota, f) \in$ Seqs, temporary view $d$, event order $eo \in <$, instance $i \in I$)
    $(e_1, e_2) \leftarrow eo$
    $e'_1 \leftarrow \text{identifyUpwards}(s, e_1, \text{UP})$
    $e'_2 \leftarrow \text{identifyUpwards}(s, e_2, \text{DOWN})$
    $p \leftarrow$ execute Petri net for $d$ and $e'_1, e'_2$
    $p' \leftarrow \{(e, e') \in p \mid \text{getInstance}((e, e')) = i\}$
    return getEventOrder($p'$)
end function

function traverseUpwards(sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, message $m \in O_M$, send order $so \in <$, receive order $ro \in <$)
    $i_S \leftarrow \text{getInstance}(so)$
    $i_R \leftarrow \text{getInstance}(ro)$
    if $\perp \neq \mu(i_S) \neq \mu(i_R) \neq \perp$ then
        $m' \leftarrow$ new message with the same type as $m$
        $O_M \leftarrow O_M \cup \{m'\}$
        $G \leftarrow G \cup \{(m', m)\}$
        $d \leftarrow$ generate temporary view for $\{i_S, i_R\}$
        $eo_S \leftarrow$ determineEventOrderUp$(d, so, \mu(i_S))$
        $eo_R \leftarrow$ determineEventOrderUp$(d, ro, \mu(i_R))$
        addMessageEvents$(s, l, m', eo_S, eo_R)$
        traverseUpwards$(s, l-1, m', eo_S, eo_R)$
    end if
end function

function determineEventOrderDown(sequence $s = (E, \iota, f) \in$ Seqs,
temporary view $d$, event $e_1 \in E$, event $e_2 \in E$, instances $R \subseteq I$)
    $e_1' \leftarrow$ identifyDownwards$(s, e_1, \text{UP})$
    $e_2' \leftarrow$ identifyDownwards$(s, e_2, \text{DOWN})$
    $p_1 \leftarrow$ execute Petri net for $d$ and $e_1', e_2'$
    $p_2 \leftarrow$ execute breadth first search on the Petri net for d and $e_1', e_2'$
    $p_V \leftarrow p_1 \cap p_2$ // valid insertion points
    $p_G \leftarrow p_1 \setminus p_2$ // insertion points with general ordering
    $p \leftarrow p_V \cup p_G$
    $p' \leftarrow \{(e, e') \in p \mid \text{getInstance}((e, e')) \in R\}$
    return getEventOrder(p')
end function

function traverseDownwards(sequence $s = (E, \iota, f) \in$ Seqs, level $l \in$
$\mathbb{N}_{\leq n}$, message $m \in O_M$, send order $so = (e_{S1}, e_{S2}) \in <$, receive order
$ro = (e_{R1}, e_{R2}) \in <$)
    $i_S \leftarrow$ getInstance$(so)$
    $i_R \leftarrow$ getInstance$(ro)$
    $R_S \leftarrow \{i \in I \mid \mu(i) = i_S\}$
    $R_R \leftarrow \{i \in I \mid \mu(i) = i_R\}$
    if $R_S \neq \emptyset \vee R_R \neq \emptyset$ then
        $m' \leftarrow$ new message with the same type as $m$
        $O_M \leftarrow O_M \cup \{m'\}$
        $G \leftarrow G \cup \{(m, m')\}$
        $d \leftarrow$ generate temporary view for $R_S \cup R_R$
        $eo_S \leftarrow$ determineEventOrderDown$(d, e_{S1}, e_{S2}, R_S)$
        $eo_R \leftarrow$ determineEventOrderDown$(d, e_{R1}, e_{R2}, R_R)$
        addMessageEvents$(s, l, m', eo_S, eo_R)$
        traverseDownwards$(s, l+1, m', eo_S, eo_R)$
    end if
end function

$m \leftarrow$ new message of type $t$
$O_M \leftarrow O_M \cup \{m\}$
addMessageEvents$(s, l, m, so, ro)$
traverseUpwards$(s, l-1, m, so, ro)$
traverseDownwards$(s, l+1, m, so, ro)$
```

The next operation adds a new operand to a given fragment and integrates the new operand into all necessary levels of abstraction. Analogous to the operation add message, the integration of the new operand into the more abstract levels of abstraction is done by the function *traverseUpwards* and for the more detailed levels of abstraction by the function *traverseDownwards*.

add operand

Input: sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, fragment $o_F \in O_F$, assertion $a \in \Sigma$, instances $T \subseteq I$, event order $eos \subseteq <$

Require:

- if $t(o_F) \in \{\text{loop}, \text{opt}, \text{break}, \text{neg}\} \Rightarrow |\eta^{-1}(o_F)| = 0$
- for each $(e, e') \in eos$: e, e' belongs to one instance $i \in T$ and there is no other $(e, e') \in eos$ that belongs to i
- for each $(e, e') \in eos$: e, e' are subsequent according to the event order, *i.e.*, $e < e' \wedge \forall\ e'' \in E : e \neq e'' \neq e' \Rightarrow e'' < e \vee e' < e''$
- check event orders (described in the Chapter 3)

Body:

```
function addOperandEvents(sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, operand $o_O \in O_O$, event order $eo \in <$)
    $e_{OB} \leftarrow$ operand begin event for $o_O$
    $e_{OE} \leftarrow$ operand end event for $o_O$
    $\iota(e_{OB}) \leftarrow (\text{getInstance}(eo), o_O)$
    $\iota(e_{OE}) \leftarrow (\text{getInstance}(eo), o_O)$
    $E_{OB} \leftarrow E_{OB} \cup \{e_{OB}\}$
    $E_{OE} \leftarrow E_{OE} \cup \{e_{OE}\}$
    $(I_l, <_l) \leftarrow f(l)$
    $<_l \leftarrow \text{addToEventOrder}(e_{OB}, eo, <_l)$
    $<_l \leftarrow \text{addToEventOrder}(e_{OE}, eo, <_l)$
end function

function identifyUpwards* (sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, events $E' \subseteq E$, direction $d \in \{\text{UP}, \text{DOWN}\}$)
    $(I_{l-1}, <_{l-1}) \leftarrow f(l-1)$
    $e_R \leftarrow \bot$
    for all $e \in E'$ do
        $e' \leftarrow \text{identifyUpwards}(e, d)$
        if $e_R = \bot \vee e' <_{l-1} e_R$ then
            $e_R \leftarrow e'$
        end if
    end for
    return $e_R$
end function
```

```
**function** traverseUpwards(sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$,
operand $o_O \in O_O$, instances $T \subseteq I$, event orders $eos \subseteq <$)
  $M \leftarrow \{i \in I \mid \exists\, i' \in T : \mu(i') = i\}$
  **if** $|M| > 1$ **then**
    $d \leftarrow$ generate temporary view for instances $M$
    **for all** $m \in M$ **do**
      $E_1 \leftarrow \{e \in E \mid \exists\ i \in I, o \in O, e' \in E : \iota(e) = (i,o) \wedge \mu(i) = m \wedge (e,e') \in eos\}$
      $e_1 \leftarrow$ identifyUpwards*$(s, l, E_1, \mathrm{UP})$
      $E_2 \leftarrow \{e \in E \mid \exists\ i \in I, o \in O, e' \in E : \iota(e) = (i,o) \wedge \mu(i) = m \wedge (e',e) \in eos\}$
      $e_2 \leftarrow$ identifyUpwards*$(s, l, E_2, \mathrm{DOWN})$
      $P \leftarrow$ execute Petri net for diagram $d$ and $e_1$, $e_2$
      $P' \leftarrow$ restrict possibilities to instance $m$
      $eo \leftarrow$ getEventOrder$(P')$
      addOperandEvents$(s, l, o_O, eo')$
      $eos' \leftarrow eos' \cup \{eo\}$, $T' \leftarrow T' \cup \{m\}$
    **end for**
    traverseUpwards$(s, l, o_O, T', eos')$
  **end if**
**end function**

**function** traverseDownwards(sequence $s = (E, \iota, f) \in$ Seqs, level $l \in \mathbb{N}_{\leq n}$, operand $o_O \in O_O$, instances $T \subseteq I$, event orders $eos \subseteq <$)
  **if** $\{i \in I \mid \mu(i) \in T\} \neq \varnothing$ **then**
    $d \leftarrow$ generate temporary view for instances
    **for all** $(e_1, e_2) \in eos$ **do**
      $(i, o) \leftarrow \iota(e_1)$
      $C \leftarrow \{i' \in I \mid \mu(i') = i\}$
      $e \leftarrow$ identifyDownward$(s, e_1, \mathrm{UP})$
      $e' \leftarrow$ identifyDownward$(s, e_2, \mathrm{DOWN})$
      $P \leftarrow$ execute Petri net for diagram $d$ and $e$, $e'$
      **for all** $c \in C$ **do**
        $P' \leftarrow$ restrict possibilities to instance $c$
        $eo \leftarrow$ getEventOrder$(P')$
        addOperandEvents$(s, l, o_O, eo)$
        $eos' \leftarrow eos' \cup \{eo\}$
        $T' \leftarrow T' \cup \{c\}$
      **end for**
    **end for**
    traverseDownwards$(s, l, o_O, T', eos')$
  **end if**
**end function**
```

```
        $o_O \leftarrow$ new operand
        $O_O \leftarrow O_O \cup \{o_O\}$
        $\tau(\text{op}) \leftarrow a$
        $\eta(\text{op}) \leftarrow o_F$
        **for all** $(e_1, e_2) \in eos$ **do**
            addOperandEvents$(s, l, o_O, e_1, e_2)$
        **end for**
        traverseUpwards$(s, l-1, o_O, T, eos)$
        traverseDownwards$(s, l+1, o_O, T, eos)$
```

B.3. Auxiliary Operations

The following auxiliary operations are used by the model operations (cf. Section 3.3) for recurrent tasks such as adding an event to an event order or determining the instance that corresponds to an event order tuple if it belongs to the total order of one lifeline.

```
function getInstance($eo = (e, e') \in <$)
    if $e \neq \bot$ then
        $(i, o) \leftarrow \iota(e)$
    else if $e' \neq \bot$ then
        $(i, o) \leftarrow \iota(e')$
    else
        $i \leftarrow \bot$
    end if
    return $i$
end function
```

getInstance returns the instance of an event order, which is part of the total order of a lifeline. If the event order represents the point before the first event or below the last element, respectively, one of both events is undefined. Thus, the function uses both events to get the instance. If both events are undefined, the result is also undefined.

```
function addToEventOrder($e \in E,\ (e_1, e_2) \in E \times E,\ < \subseteq E \times E$)
    $< \leftarrow < \cup \{(e, x) \mid \exists\ x \in E, i \in I, o, o' \in O : e_1 < x \wedge \iota(x) = (i, o) \wedge \iota(e_1) = (i, o')\}$
    $< \leftarrow < \cup \{(x, e) \mid \exists\ x \in E, i \in I, o, o' \in O : x < e_2 \wedge \iota(x) = (i, o) \wedge \iota(e_2) = (i, o')\}$
    return $<$
end function
```

The function *addToEventOrder* adds an event e to the event order $<$ between (e_1, e_2). Furthermore it maintains the transitive closure of the total order for the corresponding lifeline.

```
**function** getEventOrder(possibilities $P \subseteq <$)
  **if** $|P| = 1$ **then**
    $(e, e') \in P$
  **else**
    $(e, e') \leftarrow \text{askUser}(P)$
  **end if**
  **return** $(e, e')$
**end function**
```

The function *getEventOrder* takes a set of event ordering possibilities and returns the contained element, if there is only one element or asks the user to choose an event ordering otherwise.