Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Consistent Refinement of Sequence Diagrams in the UML 2.0

cand. inform. Claus-André Ohlhoff

November 26, 2006

Department of Computer Science
Real-Time and Embedded Systems Group

Advised by:
Prof. Dr. Reinhard von Hanxleden
Dipl. Phys. Carsten Ziegenbein*
Dipl. Inf. Björn Lüdemann*

*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

Iterative and incremental development processes are a common way to handle complex systems. Starting from a high-level view of a system, several consecutive abstraction layers are created and refined until the desired level of detail is reached. Sequence diagrams provide a comprehensible communication medium for audiences of different technical knowledge since they are almost self-explanatory due to their rather lean syntax. By using sequence diagrams to capture the requirements of a system, stepwise refinement creates a relationship between the requirements and the implementation. The consistency of each refinement step is crucial to achieve a system that fulfills its requirements. The latest release of the UML sequence diagrams greatly increases their expressive power. The semantical level, however, is raised from partial orders that are easy to handle to trace languages of factorial complexity in the worst case. Thus, existing refinement approaches suffer from very inefficient computations. This thesis introduces a refinement concept—namely intra-level and inter-level refinement—that is based on several rules to limit the structural refinement. With these structural restrictions it is possible to efficiently validate refinement relations in terms of partial orders. The execution of a Petri net representation helps to isolate misordered messages if the refinement is incorrect.

Keywords UML, sequence diagrams, refinement, incremental development, intra-model consistency

Contents

List of Figures

Glossary of Symbols

e, e'	events	6
E, E'	sets of events	6
m, m'	messages	6
M, M'	sets of messages	6
$!m$	the send event of the message m	6
$?m$	the receive event of the message m	6
$E(M)$	set of events that belong to messages in M	6
b, b'	basic sequence diagrams	7
B, B'	sets of basic sequence diagrams	7
i, i'	instances	7
I, I'	sets of instances	7
$\iota(e)$	the instance of the event e	7
$<$	a partial event order	7
s, t	regular sequence diagrams	9
c, c'	combined fragments	9
C, C'	sets of combined fragments	9
f	the root fragment of a regular sequence diagram	9
τ	tree structure of a regular sequence diagram	9
$<^*$	total order on fragments/operands	9
h	instance hierarchy	17
μ	relation between master and refinement instances	17
π	relation between parent and child instances	17
o, o'	operands	21
$O(c)$	operands that belong to the fragment c	21
$M(c)$	messages that belong to the fragment c	21
$\|\| c \|\|$	the partial event order of the fragment c	25
$O^*(c)$	the ordered pairs of operands of the fragment c	25
n, n'	Petri nets	26
p, p'	places	26

P, P'	sets of places	26
t, t'	transitions	26
T, T'	sets of transitions	26
F	flow relation of a Petri net	26
$\bullet t$	input places of transition t	27
$\bullet n$	input places of the Petri net n	27
$t\bullet$	output places of the transition t	27
$n\bullet$	output places of the Petri net n	27
x, x'	markings	27

1. Introduction

The Unified Modeling Language (*UML*) [23] allows to specify systems by various structural and behavioral diagram types. A single diagram usually shows only one part of the system to manage complexity. Thus, specifications of complex systems contain many different diagrams. Furthermore, a common technique is not only to describe all the important parts, but also to describe single parts from several different views. In doing so, it is important to keep the different diagrams consistent [8]. Otherwise a realization of such a system is likely to fail and revising requirements, design, and implementation in more iterations than necessary dramatically increases the overall effort.

This thesis examines a consistent refinement concept for sequence diagrams. Sequence diagrams focus on *inter-object scenario-based behavior* [15], *i.e.*, which instances are involved, which messages are exchanged, and in which order are these messages exchanged. Sequence diagrams are refined by inserting additional instances or by replacing instances with other instances and by inserting new messages (*cf.* Chapter 3). Refinement leads to a set of sequence diagrams that describe the same scenario from different levels of abstraction, which suit different audiences. The highest abstraction level, for example, is very useful to discuss requirements with customers and end-users, and the lowest abstraction level serves as a pattern for the implementation team. Current modeling tools, however, do not support specification of refinement relationships between sequence diagrams, which opens the door for several inconsistencies, such as missing or wrong instances or messages, and—most devious—misordered messages.

1.1. Environment

This diploma thesis is developed in cooperation with a software development team at Philips Medical Systems (*PMS*) [31] in Hamburg, Germany. This team develops embedded real-time systems based on a *ROOM* [3] framework and uses the UML with real-time extensions [4] as modeling language.

The main components are described by so-called *capsules*. Capsules are classes with the stereotype *active*, *i.e.*, all members are private and the only possibility to communicate with other capsules is message passing. Capsules are hierarchical, *i.e.*, they can be arbitrarily nested. A state machine, which can be hierachical too, specifies the internal behavior of each capsule. Therefore, sequence diagrams are particularly suitable to describe the communication of those capsules at almost all development stages. Sequence diagrams are even used to generate test cases.

Project development in the aforementioned software team follows a customized version of the Rational Unified Process (*RUP*) [30]. A detailed description of this process is beyond the scope of this paper, but the distinction between a horizontal and a vertical dimension is important for the refinement concept. The horizontal dimension is separated in consecutive iterations and phases, where each iteration leads to a new version and each phase leads to a *new generation of the product* [35]. The vertical dimension is split into several interdependent *workflows* or *disciplines*, such as *requirements*, *analysis*, *design*, *implementation*, and *testing*.

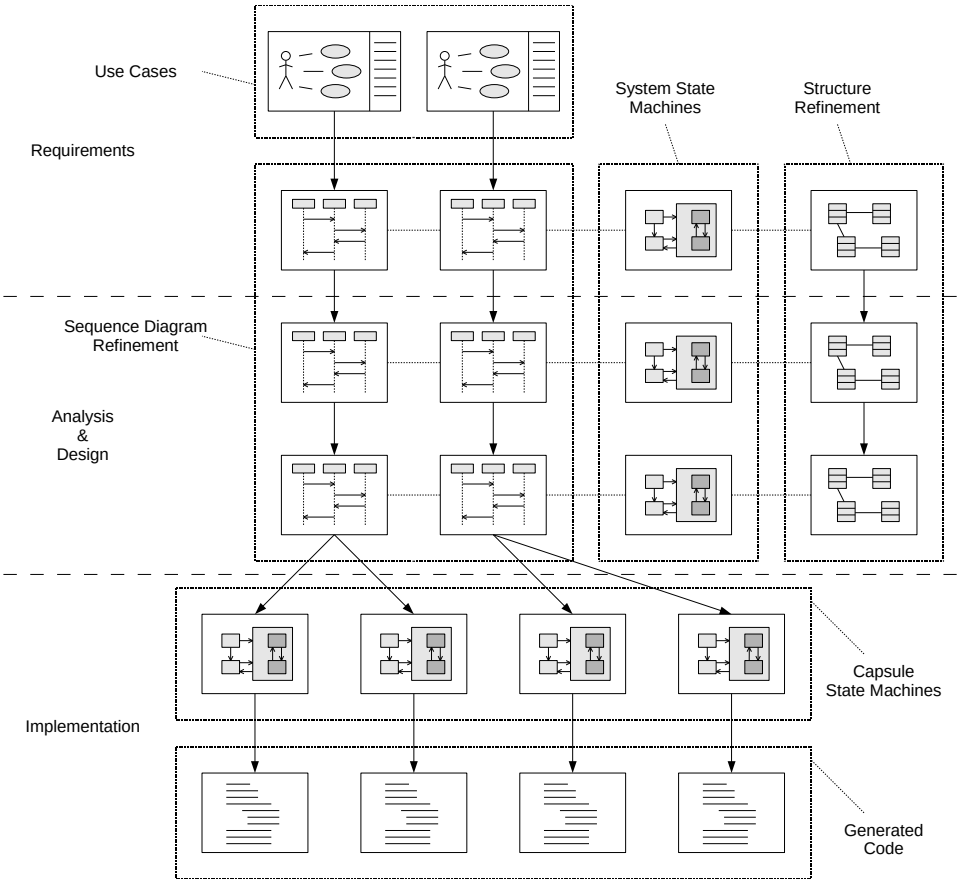


Figure 1.1.: An outline of a single iteration of the Rational Unified Process.

Figure 1.1 shows a rough outline of a single iteration of the development process. The different disciplines introduce several layers, which describe the specified system from different levels of abstraction. Each layer is a more detailed version of its predecessor and contains its own set of capsules, state machines, sequence diagrams, and a so-called *system state machine* [19]. The requirements discipline creates a set of *use cases*, which identify several scenarios and provide informal descriptions of those scenarios. The requirements discipline also derives the first layer that provides a high-level view of the system. The analysis and design disciplines iteratively refine

the requirement view of the system until a sufficient level of detail is reached which serves as a basis for the implementation.

This thesis focuses on the refinement of sequence diagrams. Each sequence diagram describes a scenario of the requirements discipline from the perspective of the particular abstraction layer, *i.e.*, a sequence diagram contains only capsules that are defined within the same layer. There are two possibilities to refine a sequence diagram. *Intra-level* refinement is applied to sequence diagrams that belong to the same level of abstraction. As mentioned above, capsules may be nested, and an intra-level refinement step is used to show the internal communication. *Inter-level* refinement is used to transfer a sequence diagram to the next lower level of abstraction by substituting the instances according to refinement of the structural elements. A detailed description of the refinement concept is provided by Chapter 3.

Sequence diagrams from the lowest level of abstraction serve as patterns for the state machine design so that there is a relationship that ranges from the requirements to the implementation. All known tools, however, seem to neglect the consistent integration of sequence diagrams. An automatic code synthesis considers only structural diagrams and state machine diagrams. Developers still have to exploit the relationships between different sequence diagrams and between sequence diagrams and state machines manually.

Hence, automatic tool support would heavily influence the development process in terms of quality and time and would optimize the product quality relative to the development effort. This is even more apparent when one considers that it is unrealistic to develop a complete system in a single iteration. Requirements are prone to changes due to customer requests, or possibly the design team itself discovers that the requirements need adjustments.

A concept and tool support for synthesis of state machines from sequence diagrams is already provided by Lüdemann [21], which can be used to assure consistency between sequence diagrams and state machines. Furthermore, Lischke [19] deals with different kinds of consistency between sequence diagram. Sequence diagrams that belong to the same abstraction layer are analyzed with a so-called *system state machine* that describes how the global system state changes by the execution of sequence diagrams. The system state machine detects state inconsistencies that lead to unwanted execution traces. Lischke also provides a first refinement concept. This concept, however, is limited to basic sequence diagrams and does not support any kind of parallelism. This limitation leads to the following objectives.

1.2. Objectives

The main objective of this diploma thesis is to extend the already existing refinement concept for basic sequence diagrams [19] to support coregions, implicit parallelism, and, particularly important, the new combined fragments introduced by UML 2.0 [11]. Several secondary objectives were derived and are ordered by importance below.

1. Evaluate whether other refinement approaches that deal with sequence dia-

grams are similar or even applicable to the concept that is already applied in practice.

2. Define a concept that allows efficient decision making whether related sequence diagrams, *i.e.*, sequence diagrams that are views of the same scenario, are consistent or not. The source of inconsistency should be identified as precisely as possible so that inconsistency elimination becomes easier.

3. Implement a plugin into the current modeling tool to evaluate the concept. The implementation should be as tool independent as possible.

1.3. Overview

After the introduction clarified the context and the objectives of this thesis, this section gives an overview of the remaining chapters.

Chapter 2 introduces the syntax and semantics of sequence diagrams and provides a formal representation that is used to define the different relationships in the following chapter. Furthermore, a rather simple example is used to show that reasoning about sequence diagrams at trace level is practically impossible due to the factorial complexity of parallel behavior.

Chapter 3 defines the refinement concept. An instance hierarchy which refers to the refinement of capsules restricts the refinement of sequence diagrams. Several rather strict structural rules allow to reduce the semantical level to partial event orders. Strongly connected components and a Petri net simulation accurately identify misordered messages, if any exist.

Chapter 4 shows how the concept was implemented. A rather small client plugin generates an XML representation of the sequence diagrams that should be checked for consistency. An external checking server receives this XML representation and processes the different rules of the previous chapter.

Chapter 5 presents other approaches that deal with the refinement of sequence diagrams. There are three similar approaches that define refinement relationships in terms of traces, but do not provide efficient methods to verify those relationships. A fourth approach uses a Petri net analysis which is also inefficient. Besides the different refinement concepts several other *ambiguities* of sequence diagrams are mentioned that cannot be avoided by the refinement concept itself. These ambiguities may lead to errors, such as deadlocks, in an implementation.

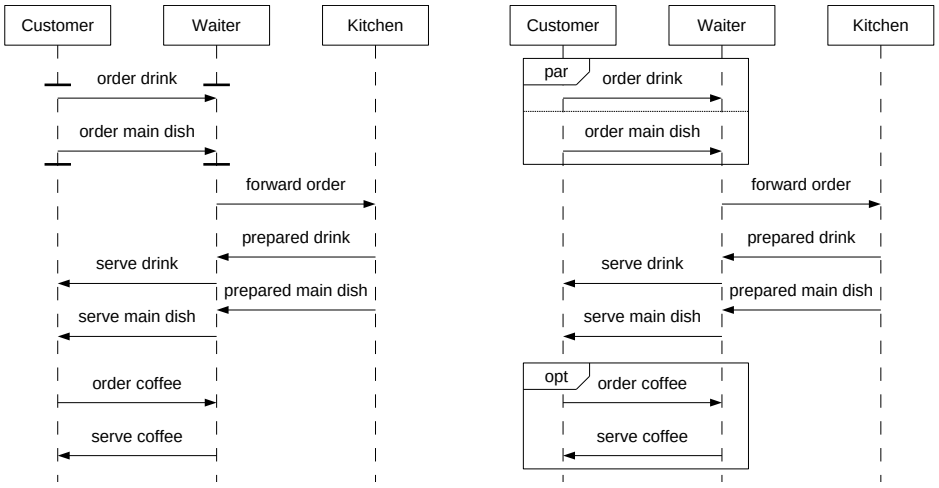
Chapter 6 concludes and discusses possible future extensions. A model/view relationship for sequence diagrams, which is inspired by handling of class diagrams by current modeling tools, could automatically eliminate several sources of inconsistency.

All examples throughout this thesis show sequence diagrams that were taken from the restaurant example which was already used by Lischke [19] and Lüdemann [21]. This rather simple model provides familiar scenarios, which simplifies the evaluation and presentation of new concepts.

2. Sequence Diagrams

The current version of UML sequence diagrams has adopted many features of Message Sequence Charts (*MSC*) [18], which have been standardized by the International Telecommunication Union (*ITU*) [17], and Life Sequence Charts (*LSC*), which have been introduced by Harel *et al.* [7, 42].

Sequence diagrams are used to describe how instances interact in a specific scenario. Such a scenario is usually informally described by a use case. Thus, a sequence diagram provides the first possibility for structural and formal analyses. Distinguishing *basic* sequence diagrams from regular sequence diagrams is a common approach to introduce a formal notation [38, 5]. As opposed to regular sequence diagrams, basic sequence diagrams are *flat*, *i.e.*, they contain only instances, messages and *coregions*. Regular sequence diagrams contain in addition so-called *combined fragments* that introduce a kind of hierarchy.



(a) A *basic* sequence diagram.

(b) A regular sequence diagram.

Figure 2.1.: A basic and a regular sequence diagram.

Figure 2.1 shows a basic and a regular sequence diagram. Instances are drawn as boxes at the top of the diagram. The dashed lines below each instance are *lifelines*, which indicate the time elapsed. Messages are drawn as labeled arrows between those lifelines, thus connecting the sender and the receiver of that message. Furthermore, basic sequence diagrams may contain *coregions*, which are drawn as thick lines and cover an interval of a lifeline. Coregions are used to indicate that several messages

may be sent or received in any order. The two coregions in the sequence diagram in Figure 2.1(a) indicate that the customer can order the drink and the main dish in any order, and that the waiter can receive these messages in any order. Depending on the underlying framework it is even possible that one message overtakes another. This is, however, not allowed in the ROOM [3] framework, and thus not included in this thesis.

Combined fragments are drawn as rectangles that cover several messages. Each combined fragment has a so-called *operator* that is written in the top left corner. Depending on the type of operator, the combined fragment may be separated with a dashed line into several regions, so-called *operands*. The sequence diagram in Figure 2.1(b) describes basically the same scenario as the basic sequence diagram, but due to the opt fragment the coffee order is only optional, *i.e.*, the customer may or may not order a coffee at all, which cannot be expressed with basic sequence diagrams. The following section provides the formalization of sequence diagrams. Section 2.2 explains the *trace semantics* and shows that addressing a consistent refinement concept at the trace level is much too inefficient.

2.1. Formalization

This section provides a formal representation of sequence diagrams and their components. This formalization is used to define the refinement concept in Chapter 3. The formalization of basic sequence diagrams is similar to the notation of Peled [28] or Haugen *et al.* [25, 27].

Messages are used to describe the interfaces and protocols between different instances. The communication is asynchronous. Synchronous messages are not separately considered since they could be replaced with asynchronous request and reply messages. Special *create* or *destroy* messages, which indicate the creation and termination of an instance, respectively, are also presented as simple asynchronous messages. Each message is identified by a pair of *events*: a *send* event and a *receive* event. From the state machine view [21] a receive event refers to a trigger of a transition and send events are a part of the action code of a transition.

Definition (Message):
Let E be a set of send events and let E' be a set of receive events, so that E and E' are disjoint. A *message* m is a pair $(s, r) \in E \times E'$. $!m$ and $?m$ refer to the sending event and receiving event of m, *i.e.*, $!m = s$ and $?m = r$.
A set M of messages is *well-formed* if all events are unique, *i.e.*,

$$\forall m, m' \in M.\ (m \neq m') \Rightarrow (!m \neq !m' \wedge ?m \neq ?m').$$

For a set M of messages, $E(M)$ refers to the set of all events that belong to these messages, *i.e.*,

$$E(M) = \{!m | m \in M\} \cup \{?m | m \in M\}.$$

A basic sequence diagram is characterized by a set of instances and messages and a mapping that assigns an instance to every message. This definition facilitates that messages may appear within several sequence diagrams, which may assign these messages to different instances according to the refinement concept (*cf.* Chapter 3). Other formalizations of sequence diagrams [38, 5] do not exploit the possibility that several sequence diagrams share a common set of elements.

Definition (Basic Sequence Diagram):
A *basic sequence diagram* is a tuple

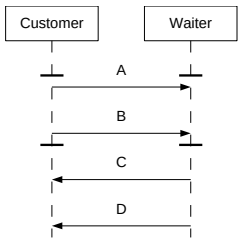
$$b = (I, M, \iota, <)$$

where

- I is a set of instances,
- M is a well-formed set of messages,
- $\iota : E(M) \rightarrow I$ maps every event on an instance,
- $< \subset E(M) \times E(M)$ is a partial event order, and
- every send event occurs before the corresponding receive event, *i.e.*,

$$\{(!m, ?m) | m \in M\} \subseteq < .$$

The partial order does not only relate send and receive events but also events that belong to the same instance and occur one after the other. A full definition of basic sequence diagrams should also include an additional restriction of the partial order, because not every order can be realized by a graphical representation. Assume that events a, b, and c belong to the same instance and, according to the partial order, a occurs before b but there is neither a restriction between a and c nor between b and c. This entails that a, b, and c have to belong to the same coregion, which contradicts the order of a and b and thus, a graphical representation does not exist. This restriction, however, has no influence on the refinement concept, which is defined in the next chapter. Figure 2.2 shows a simplified version of the diagram in Figure 2.1(a) together with its formal representation.



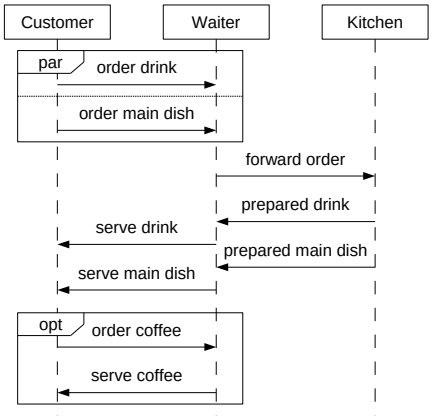
(a) Graphical representation.

$$I = \{\text{Customer}, \text{Waiter}\}$$
$$M = \{A, B, C, D\}$$
$$\iota(e) = \begin{cases} \text{Customer , if } e \in \{!A, !B, ?C, ?D\} \\ \text{Waiter , otherwise} \end{cases}$$
$$< = \{(!A, ?A), (!B, ?B), (!C, ?C), (!D, ?D), (!A, ?C), (!B, ?C), (?A, !C), (?B, !C), (!C, !D), (?C, ?D)\}$$

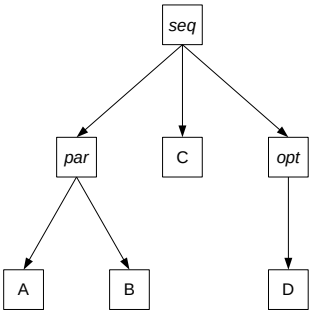
(b) Formal representation.

Figure 2.2.: A simple sequence diagram with its formal representation.

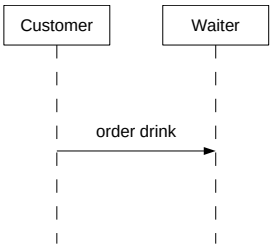
Combined fragments are used to introduce a hierarchy into sequence diagrams. Fragments can be arbitrarily nested but must not overlap, so that regular sequence diagrams can be formalized as a tree, which has ordered nodes. Basic sequence diagrams represent leaf nodes of such a tree and combined fragments cover the inner nodes. The root of such a tree is either taken by the outermost fragment or by the implicit fragment with the weak sequencing operator `seq`, which has no effect on the semantics of the diagram (*cf.* Section 2.2). Figure 2.3 shows the decomposition and the tree view of the sequence diagram in Figure 2.1(b).



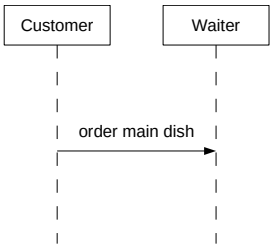
(a) A regular sequence diagram.



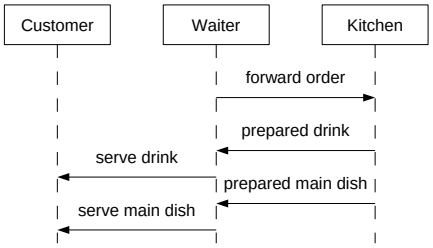
(b) Tree with basic diagrams as leaves and fragments as inner nodes.



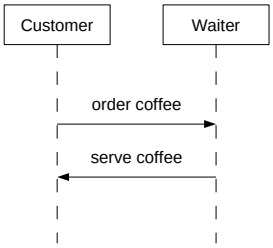
(c) Basic sequence diagram A.



(d) Basic sequence diagram B.



(e) Basic sequence diagram C.



(f) Basic sequence diagram D.

Figure 2.3.: Decomposition of a sequence diagram into basic sequence diagrams and the corresponding tree view.

The formalization of regular sequence diagrams separates the combined fragments from their content, which is similar to the separation from messages and their sending and receiving instances. Thus, a fragment may appear in several diagrams and potentially even covers different sets of operands and messages, which is quite different from other approaches that deal with regular sequence diagrams (*cf.* [38, 5]).

Notation:
In the following, indices are used to distinguish sets, relations, and functions when the context is unclear, *e.g.*, I_b refers to the instances of a sequence diagram b and $<_{b'}$ refers to the event order of sequence diagram b'.

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

$$s = (B, C, f, \tau, <^*)$$

where

- B is a set of basic sequence diagrams without common messages or events, *i.e.*,

$$\forall b, b' \in B.\ \ M_b \cap M_{b'} = \emptyset$$
$$\text{and } \bigcup_{b \in B} M_b \text{ is well-formed,}$$

- C is a set of combined fragments,
- $f \in C$ is the root fragment,
- $\tau \subset C \times (C \cup B)$ relates combined fragments with their operands, which can be combined fragments again or basic sequence diagrams, and
- $<^*$: $(B \cup C) \times (B \cup C)$ is a total order on the basic sequence diagrams and fragments.

τ refers to the tree structure of such a diagram and $<^*$ represents the order of fragments and basic sequence diagrams. $<^*$ is a preorder traversal of the tree induced by τ. The order of the fragments and basic sequence diagrams in Figure 2.3 is:

$$\text{seq} <^* \text{par} <^* A <^* B <^* C <^* \text{opt} <^* D$$

The following notations that are used for basic sequence diagrams can also be used for regular sequence diagrams.

Notation:
Let s be a sequence diagram.

- I_s refers to the set of all instances in s, *i.e.*,

$$I_s = \bigcup_{b \in B_s} I_b.$$

- M_s refers to the set of all messages in s, *i.e.*,

$$M_s = \bigcup_{b \in B_s} M_b.$$

- $\iota_s : E(M_s) \to I_s$ maps every event on an instance,*i.e.*,

$$\iota_s = \bigcup_{b \in B_s} \iota_b.$$

2.2. Trace Semantics

Partial orders are an adequate semantics for basic sequence diagrams, but are not sufficient to describe the semantics of regular sequence diagrams. As mentioned at the beginning of this chapter there is, for example, no possibility to express optional behavior if only partial orders are available. Furthermore, the current UML specification [11] allows to specify alternative and iterative behavior which significantly increases the expressiveness of sequence diagrams.

The introduction of the combined fragment with their twelve different operator kinds (*cf.* Table 2.1) requires a *trace semantics*. A trace is a sequence of events that describes an execution of the scenario. To receive the traces of a basic sequence diagram, one has to enumerate all possible total orders that are completions of the partial order. The sequence diagram in Figure 2.2 leads to twelve valid traces, such as $\langle !A, !B, ?A, ?B, !C, ?C, !D, ?D \rangle$ or $\langle !B, !A, ?A, ?B, !C, !D, ?C, ?D \rangle$. In doing so, each sequence diagram is characterized by its *trace language*.

The semantics of a sequence diagram, however, is not only given by a set of *positive* traces, but also by a set of *negative* traces. Positive traces refer to possible executions and negative traces describe unwanted or forbidden executions. Traces that are neither positive nor negative are usually referred to as *inconclusive* or *incontigent*. Unfortunately, the UML specification itself is rather vague and lacks a precise formal semantics. There are several approaches [37, 38, 5] that try to fill this gap. Especially the possibility to specify negative traces allows many different interpretations [32]. The next section shows that reasoning about sequence diagrams in terms of traces is very inefficient so that the refinement approach, which is defined in Chapter 3,

is based on partial orders. Although partial orders do not suffice to discuss the semantics of regular sequence diagrams, several structural restrictions at least allow to use partial orders to efficiently localize misordered messages.

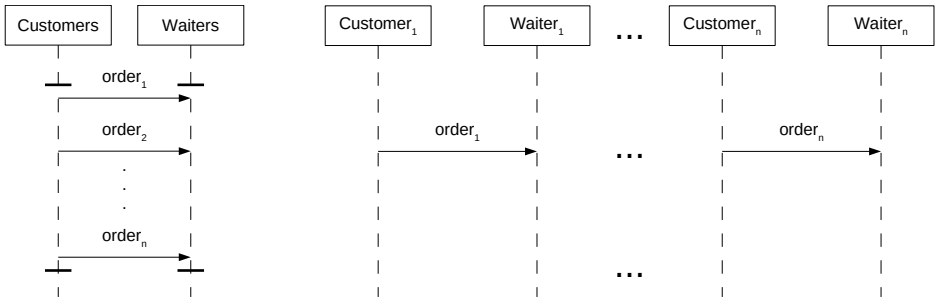
Hence, a complete discussion of the trace semantics is not necessary in this thesis. Table 2.1 contains a short description of the operators that result in positive traces. Negative traces are created with the *neg* and *assert* fragment. As mentioned above, there is no best way to handle negative traces yet. A good overview is provided by Runde *et al.* [32]. The refinement concept presented in the next chapter, however, does not need to distinguish between negative and positive behavior. Complete descriptions of all operators are provided by several papers [37, 38, 5, 32].

operator	description
par	Parallel merge: The traces of the operands are interleaved in any way. The *par* operator is more expressive than a simple coregion, since it preserves the order within each operand.
strict	Strict sequencing: The traces of the operands are concatenated in the order of the operands.
alt	Choice of behavior: The result is the union of the traces of the operands.
opt	Choice of behavior: Similar to an *alt* fragment where one operand is empty, *i.e.*, the empty trace is added to the trace set.
seq	Weak sequencing: The result is a composition of strict sequencing and interleaving. The resulting traces fulfill the following rules: • The event order of each operand is maintained. • Events that belong to different operands but to the same instance are ordered according to the order of the operands. • Events that belong to different operands and different instances may come in any order.
loop	Repeated behavior: The traces of the single operand are concatenated via weak sequencing.
break	Breaking scenario: The traces of the *break* fragment appear instead of the remainder of the enclosing fragment, so that the *break* fragment has no influence on the traces of its operand, but on the traces of its enclosing fragment.
critical	Critical region: Traces of a critical region must not be interleaved with traces of parallel behavior. Like the *break* fragment, *critical* only alters the traces of the enclosing fragment.
ignore	Ignore messages: The *ignore* fragment indicates that several messages, which are not shown within the operand, may appear anywhere within the resulting traces.
consider	Consider messages: As apposed to *ignore*, *consider* designates that only a selection of messages is allowed to appear in the resulting traces.

Table 2.1.: UML operators that result in positive traces.

Trace Complexity

The number of traces grows unmanageably fast if parallelism is involved. There exist several ways to express parallelism within sequence diagrams. Due to the *weak sequencing*, events that do not belong to the same instance and that are not related by a path of messages can occur independently. This is sometimes referred to as *implicit* parallelism. *Explicit* parallelism is introduced with *coregions* and *par* fragments.



(a) Abstract view. (b) Detailed view.

Figure 2.4.: Interleaving of parallel behavior leads to unmanageable trace sets.

Figure 2.4 shows two sequence diagrams that describe a scenario from different levels of abstraction. Each of the n customers has its own personal waiter and may place his or her order whenever he or she wants to. This allows an almost arbitrary execution, where only one restriction remains: every send event has to occur before the corresponding receive event. For n messages this leads to

$$\frac{(2n)!}{2^n} \quad \text{traces.}$$

Proof:
For $n = 1$ this is obviously true, since there is only one trace $\langle !\text{order}_1, ?\text{order}_1 \rangle$ and

$$\frac{2!}{2^1} = 1.$$

Now assume that the statement is true for $n = i$. Every trace has the length $2i$ and if a new message order_{i+1} is interleaved into the existing traces, two cases can be distinguished. If $!\text{order}_{i+1}$ and $?\text{order}_{i+1}$ directly follow each other, they can be inserted at the beginning of a trace or behind each event, *i.e.*, there are

$$2i + 1 \quad \text{possibilities.}$$

If $!\text{order}_{i+1}$ and $?\text{order}_{i+1}$ do not follow each other, two positions out of $2i + 1$ have to be selected so that $!\text{order}_{i+1}$ is inserted before $?\text{order}_{i+1}$. This leads to additional

$$\binom{2i+1}{2} \quad \text{possibilities.}$$

Overall there are

$$\begin{aligned} 2i+1+\binom{2i+1}{2} &= 2i+1+\frac{(2i+1)!}{2!(2i-1)!} \\ &= 2i+1+\frac{(2i+1)(2i)}{2} \\ &= \frac{4i+2+(2i+1)(2i)}{2} \\ &= \frac{4i^2+6i+2}{2} \\ &= \frac{(2i+1)(2i+2)}{2} \quad \text{possibilities.} \end{aligned}$$

Since the new message is interleaved into each trace, this finally leads to

$$\begin{aligned} \frac{(2i)!}{2^i}\cdot\frac{(2i+1)(2i+2)}{2} &= \frac{(2i)!(2i+1)(2i+2)}{2\cdot 2^i} \\ &= \frac{(2i+2)!}{2^{i+1}} \\ &= \frac{(2(i+1))!}{2^{i+1}} \quad \text{traces for } i+1 \text{ messages.} \end{aligned}$$

Thus, for $n = 5$ there are 113400 traces and the number of traces for $n = 20$ already has 42 digits. Hence, reasoning about traces of such sequence diagrams is virtually impossible. The general idea of the concept that is defined in the next chapter is to establish several structural restrictions, which can be efficiently verified. If these structural rules are fulfilled it suffices to compare partial event orders.

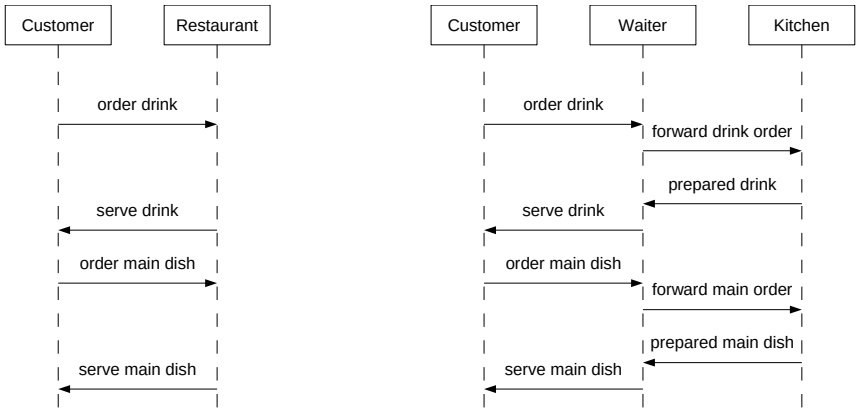
3. Refinement

This chapter uses the formalization and notation from the previous chapter to define consistency rules for the refinement of sequence diagrams. As mentioned in the introduction there are two refinement steps that are applied in practice: *Intra-level* and *inter-level* refinement. The intention of this thesis is to define these refinement relationships so that they can be verified with an efficiency that is not provided by the approaches presented in Chapter 5.

3.1. Examples

This section provides simple examples to informally introduce inter-level and intra-level refinement. The following sections define these concept formally. If a sequence diagram s is inter-level or intra-level refined by another sequence diagram t, s is called t's inter-level or intra-level *master diagram* and t is called s's inter-level or intra-level *refinement diagram*, respectively.

Inter-level Refinement



(a) A customer orders a drink and a main dish in a restaurant.

(b) Waiter and kitchen substitute the restaurant.

Figure 3.1.: A *master* diagram and an inter-level *refinement* diagram.

Inter-level refinement lowers the level of abstraction, *i.e.*, high-level instances are substituted with several more detailed instances. Figure 3.1 shows a sequence diagram with an inter-level refinement diagram. The restaurant in Figure 3.1(a) is

replaced by the waiter and the kitchen in Figure 3.1(b). All messages from the master diagram are transferred to the refinement diagram and a new message may only appear between the replacing instances.

Intra-level Refinement

Intra-level refinement reveals the internal behavior of instances. Figure 3.2 shows an intra-level refinement diagram of the sequence diagram in Figure 3.1(b). Barkeeper and cook are subinstances of kitchen. This example also shows that a refinement step may remove instances if their behavior is not important for this diagram. Thus, the refinement of a sequence diagram can focus on a specific set of instances and the result remains manageable with respect to the visual size.

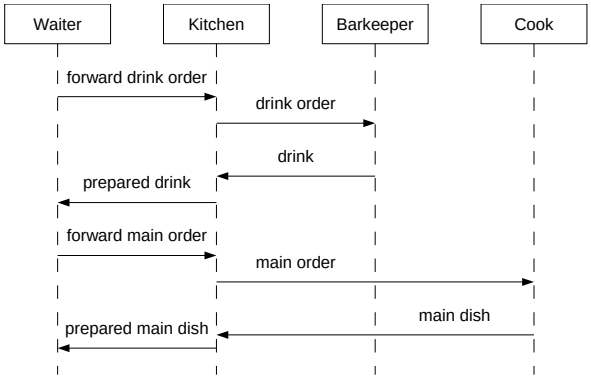


Figure 3.2.: Intra-level refinement of the sequence diagram in Figure 3.1(b).

The previous chapter showed that the common approach to define refinement relationships in terms of *trace languages*, which characterize sequence diagrams, is inefficient. Thus, the general idea of this thesis is to address the refinement at a structural level. Many inconsistencies can be avoided without looking at traces, *i.e.*, the semantical part of sequence diagrams. It is an obvious error, if the sender and the receiver of a message in a sequence diagram and its intra-level refinement diagram differ. In almost the same manner one can prescribe which instances, messages, and combined fragments have to appear within a sequence diagram, and how these elements are nested. The next section introduces an *instance hierarchy* that reflects the structural refinement, *i.e.*, which instances contain or substitute other instances. This instance hierarchy restricts the inter-level and intra-level refinement of sequence diagrams. Section 3.3 introduces several structural rules that establish the basis of the refinement concept. Section 3.4 deals with the one remaining partially semantical problem: *the order of messages*.

3.2. Instance Hierarchy

An instance hierarchy specifies which instances contain or substitute other instances and, thus, forms the backbone of the refinement concept. Any refinement of a sequence diagram must be consistent with the instance hierarchy.

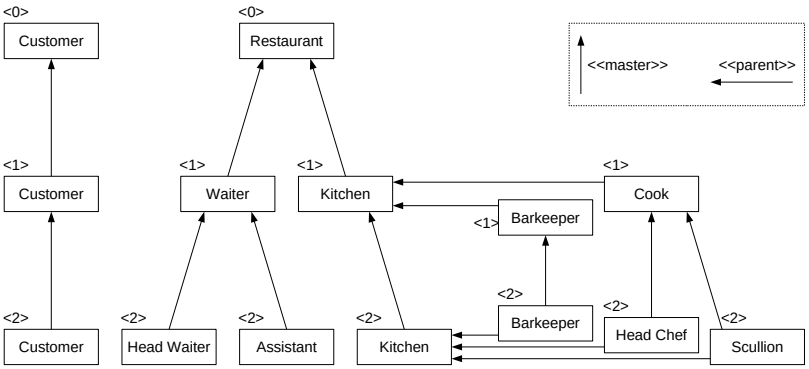


Figure 3.3.: An instance hierarchy with three different levels of abstraction.

Figure 3.3 shows the instance hierarchy that is used throughout the remainder of this chapter. Vertical arrows connect instances with their *master* instances, *i.e.*, the instance which they can replace. Horizontal arrows connect instances with their *parent* instances, *i.e.* the instance which they belong to. The refinement rules of the next sections access the following definition.

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

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

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.*,

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

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

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

and μ and π fulfill the following restrictions:

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

The acyclicity of π prevents mutual inclusion of instances and the latter restriction prescribes that the granularity of parent/child relationships is carried over all abstraction layers. If an instance i has a parent i' and a master i'' at the same time, the master i'' must have a parent i''' itself that is the master of the parent i'.

The instance hierarchy in Figure 3.3 conforms to this definition. The instances barkeeper, head chef, and scullion have parent and master instances at the same time, but all of these depend on the instance kitchen. Henceforth, let h be an instance hierarchy that is referred to in the remainder of this chapter. All sequence diagrams contain only instances that are defined in this instance hierarchy. The following extensions of the definition of sequence diagrams assure that all instances within a single diagram belong to the same abstraction layer and that no so-called *illegal messages* exist.

A message is illegal if it crosses the boundary of at least one instance. As mentioned in the introduction, capsules have only private members so that child instance are only visible to other children of the same parent or to the parent itself. Instances outside of the parent can only communicate with the parent that may forward messages to its children. A message from the waiter to the barkeeper in Figure 3.2 would be illegal. Messages between the barkeeper and the cook are allowed though.

Definition (Illegal Message):
Let s be a sequence diagram. A message $m \in M_s$ 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 $\mathrm{ILL}(s)$ refers to the *illegal messages* of s, *i.e.*,

$$\begin{aligned}\mathrm{ILL}(s) = \{m \in M_s \mid\ & \pi(\iota(!m)) \neq \pi(\iota(?m)) \\ \wedge\ & \pi(\iota(!m)) \neq \iota(?m) \\ \wedge\ & \pi(\iota(?m)) \neq \iota(!m)\}.\end{aligned}$$

Definition (Well-Formed Sequence Diagram):
A sequence diagram s is *well-formed*, if s contains no illegal message and if all instances have the same level of abstraction, *i.e.*,

$$\mathrm{ILL}(s) = \emptyset\ \wedge\ \exists k \in \{0, \ldots, n\}.\ I_s \subseteq I^k.$$

In the following all sequence diagrams are well-formed.

3.3. Structural Consistency

This section provides several structural rules that form the basis of inter-level and intra-level refinement. The first rule prescribes which instances must appear and which instances may appear within sequence diagrams that intra-level refine each other.

Definition (Intra-level Instance Refinement):
Sequence diagram t is an *intra-level instance refinement* of sequence diagram s, if every instance of t that is not in s has a parent instance in t, *i.e.*,

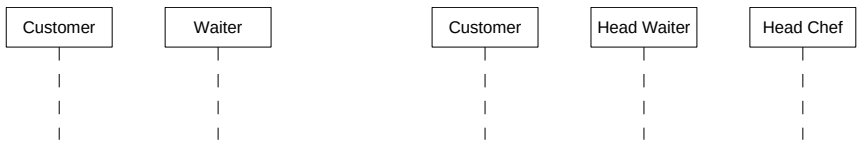
$$\forall i \in I_t \backslash I_s.\ \pi(i) \in I_t.$$

Following the above definition an intra-level refining diagram contains only instances from its master diagram and additional instances have to be children of instances that are already there. A similar rule assures that inter-level refinement diagrams conform with the instance hierarchy. The inter-level refinement relationship also allows the refining diagram to contain child instances.

Definition (Inter-level Instance Refinement):
Sequence diagram t is an *inter-level instance refinement* of sequence diagram s, if s contains all master instances from instances of 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.*,

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

Figure 3.4 shows an incorrect inter-level refinement. The sequence diagram in Figure 3.4(b) must not contain the head chef unless the sequence diagram in Figure 3.4(a) contains the cook.



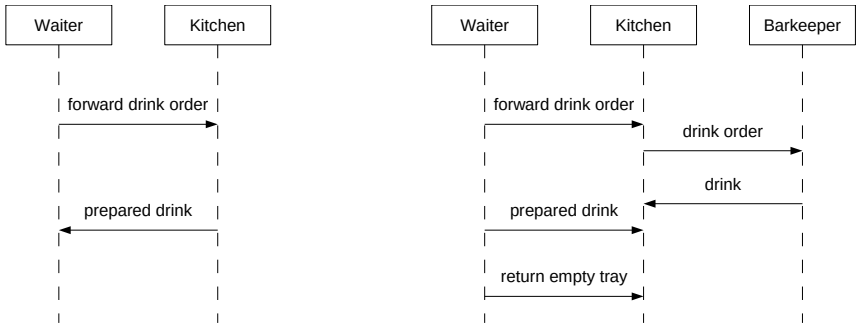
(a) Master diagram. (b) Incorrect inter-level instance refinement.

Figure 3.4.: The inter-level refinement diagram must not contain the head chef.

The following rules prescribe which messages must and must not appear within sequence diagrams that inter-level or intra-level refine each other. For sequence diagrams that intra-level refine each other the common set of messages depends on the common set of instances.

Definition (Intra-level Message Refinement):
Let s and t be sequence diagrams so that t is an intra-level instance refinement of s. t is an *intra-level message refinement* of sequence diagram s if the following holds for all messages in s and t. If the sending and receiving instance of a message in s also belong to t, this message also occurs in t, and vice versa. Furthermore, the sending and receiving instances of messages that belong to both diagrams at the same time are equal in both diagrams, *i.e.*,

$$\begin{aligned} & \forall m \in M_s.\ \{\iota_s(!m), \iota_s(?m)\} \subseteq I_t \Rightarrow m \in M_t \\ \wedge\ & \forall m \in M_t.\ \{\iota_t(!m), \iota_t(?m)\} \subseteq I_s \Rightarrow m \in M_s \\ \wedge\ & \forall m \in M_t \cap M_s.\ (\iota_s(!m) = \iota_t(!m) \wedge \iota_s(?m) = \iota_t(?m)). \end{aligned}$$



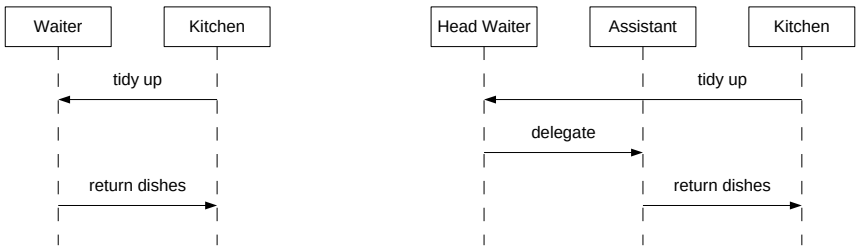
(a) Master diagram. (b) Incorrect intra-level message refinement.

Figure 3.5.: The messages return empty tray and prepared drink are inconsistent.

Figure 3.5 shows an incorrect intra-level message refinement. The message prepared drink has inconsistent sending and receiving instances and the message return empty tray has to appear in the master diagram, too. Message refinement between two sequence diagrams with different levels of abstraction is similar to intra-level refinement.

Definition (Inter-level Message Refinement):
Let s and t be sequence diagrams so that t is an inter-level instance refinement of s. t is an *inter-level message refinement* of sequence diagram s, if the following holds for all messages in s and t. If the sending and receiving instances of a message in s have refinement instances that appear in t, this message also appears in t. If the sending and receiving instances of a message in t are refinements of different instances of s, this message also appears in s. Furthermore, the sending and receiving instances of messages that appear in both diagrams are compatible with the instance hierarchy, *i.e.*,

$$\begin{aligned} & \forall m \in M_s.\ (\exists i, i' \in I_t.\ \mu(i) = \iota_s(!m) \wedge \mu(i') = \iota_s(?m)) \Rightarrow m \in M_t \\ \wedge\ & \forall m \in M_t.\ \bot \neq \mu(\iota_t(!m)) \neq \mu(\iota_t(?m)) \neq \bot \Rightarrow m \in M_s \\ \wedge\ & \forall m \in M_s \cap M_t.\ (\mu(\iota_t(!m)) = \iota_s(!m) \wedge \mu(\iota_t(?m)) = \iota_s(?m)). \end{aligned}$$



(a) Master diagram. (b) Correct inter-level message refinement.

Figure 3.6.: The message delegate may also appear within the master diagram.

Figure 3.6 shows a correct inter-level message refinement. The messages tidy up and return dishes are allocated to different instances, which is allowed since both are refinements of the same master instance. The message delegate is internal from the perspective of the master diagram and, thus, needs not to appear in the master diagram. If the message delegate appears in the master diagram it has to be a self message of the waiter. Both of the following structural rules deal with combined fragments. A fragment may or must appear within a sequence diagram depending on the messages within the sequence diagram.

Notation:
Let s be a sequence diagram and let $c \in C_s$ be a combined fragment in s. $O_s(c)$ refers to the set of *operands* that belong to c in s, *i.e.*,

$$O_s(c) = \{o \in B_s \cup C_s | (c, o) \in \tau_s\}.$$

The shortcut $M_s(.)$ refers to the set of messages that belong to fragments and basic sequence diagrams of s, *i.e.*, for a basic sequence diagram $b \in B_s$ is $M_s(b) = M_b$ and for combined fragments $c \in C_s$ is

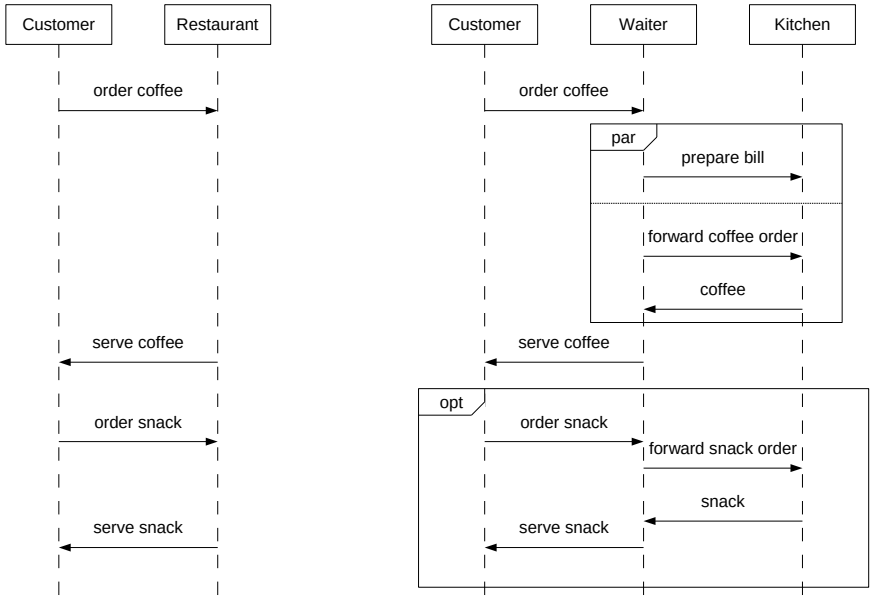
$$M_s(c) = \bigcup_{o \in O_s(c)} M_s(o).$$

Definition (Fragment Complete):
Let s and t be sequence diagrams so that t is an intra-level or inter-level message refinement of s. s and t are *fragment complete* if s contains all combined fragments from t that cover messages in t that also appear in s, and vice versa, *i.e.*,

$$\begin{aligned} &\forall c \in C_t.\ M_t(c) \cap M_s \neq \emptyset \Rightarrow c \in C_s \\ \wedge\ &\forall c \in C_s.\ M_s(c) \cap M_t \neq \emptyset \Rightarrow c \in C_t. \end{aligned}$$

Figure 3.7 shows a correct inter-level message refinement. Waiter and kitchen correctly replace the restaurant, and each diagram contains the necessary messages. Nevertheless, both diagrams are not *fragment complete*. The par fragment covers only messages that do not have to appear within the master diagram so that the fragment

does not have to appear within the master diagram, either. The opt fragment, however, covers two messages that also appear in the master diagram, so that the opt fragment has to appear in that diagram too. Without the opt fragment in the master diagram there is no hint that the snack order is only optional.



(a) Master diagram. (b) Inter-level message refinement.

Figure 3.7.: These diagrams are not *fragment complete*.

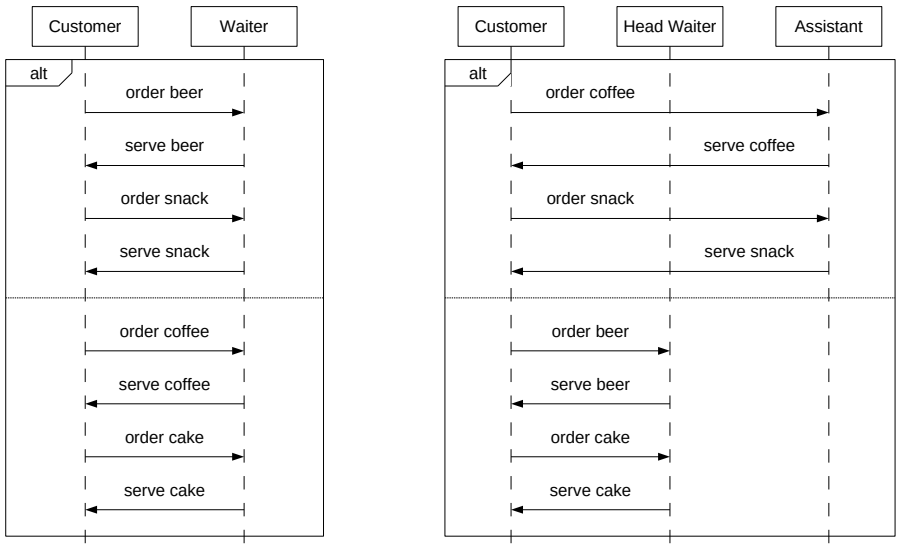
The definition of *fragment complete* does only prescribe which fragments may and must appear within sequence diagrams that refine each other, but there is no restriction on how these fragments are nested and which messages they cover. This is considered by the following definition.

Definition (Fragment consistent):
Let s and t be sequence diagrams so that s and t are *fragment complete*. s and t are *fragment consistent* if their common fragments fulfill the following property: The messages that belong to a single operand of such a fragment in one diagram, reside below a single operand of that fragment within the other diagram, *i.e.*,

$$\forall c \in C_s \cup C_t.\ \ (\forall o \in O_s(c).\ \exists o' \in O_t(c).\ M_s(o) \cap M_t \subseteq M_t(o') \\ \wedge\ \forall o \in O_t(c).\ \exists o' \in O_s(c).\ M_t(o) \cap M_s \subseteq M_s(o')).$$

Figure 3.8(b) shows a correct inter-level message refinement of the sequence diagram in Figure 3.8(a). The *waiter* is replaced by the head waiter and the assistant and all message events are relocated according to the instance hierarchy. The refinement diagram shows that each waiter is responsible for one of the two alternatives. These diagram are also *fragment complete* since the alt fragment appears in both

diagrams. This refinement, however, is not *fragment consistent*, since the messages of one operand in one diagram belong to several operands in the other diagram. The first alternative of the master diagram describes the order of a beer and a snack, and in the refinement diagram it is only possible to order a beer together with a cake, or a snack together with a coffee. Both diagrams would be *fragment consistent* if the snack and cake messages were interchanged.



(a) Master diagram.

(b) Inter-level message refinement.

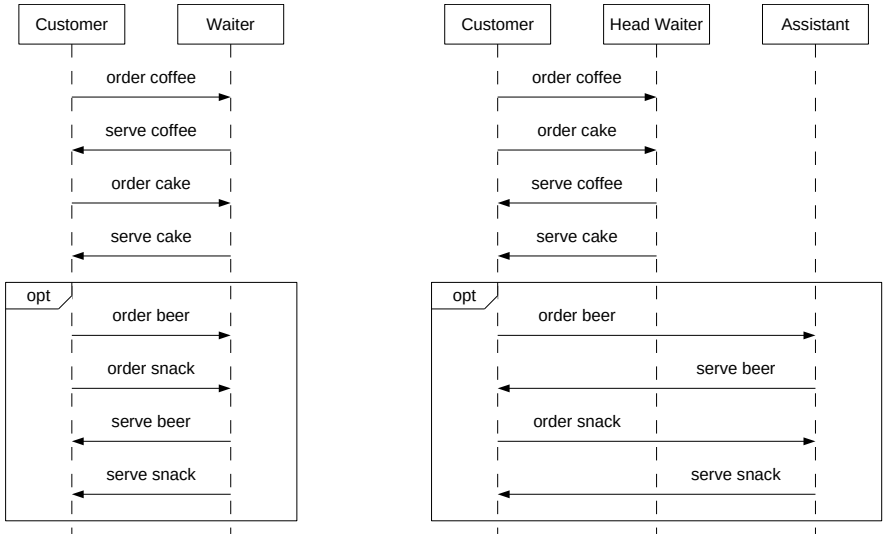
Figure 3.8.: The alt fragment is not *fragment consistent*.

These structural rules, however, are not sufficient for a consistent refinement relationship, since the order of messages is not restricted in any way. Figure 3.9(b) shows a correct inter-level message refinement of the sequence diagram in Figure 3.9(a). Head waiter and assistant replace the single waiter and the messages of the waiter are distributed between these replacing instances. Both diagrams are also *fragment consistent*, because the opt fragment appears in both diagrams and covers the same messages. The order of messages, however, is not consistent, which is crucial for the state machine design [21]. The order of order cake and serve coffee as well as the order of order snack and serve beer is mixed up.

Assume that the sequence diagram in Figure 3.9(a) represents the requirements, and the refinement in Figure 3.9(b) diagram serves as a basis for the implementation. A customer who follows the requirement scenario is not able to interact with the implementation, since he would wait for the coffee to be served before placing the cake order and at the same time, the head waiter waits for the cake order before he serves the coffee. The execution stalls with a deadlock.

A similar problem occurs within the optional beer and snack order, where the assistant may serve the beer even though the customer does not expect the beer until

he also orders the snack. At the same time the assistant does not expect an incoming snack order before serving the beer. Unexpected messages are usually thrown away or at least deferred until they are needed. Although such a defer mechanism would resolve the problem here, such situations should be avoided at all costs because a message order inconsistency in general leads to deadlocks.



(a) Master diagram.

(b) Inter-level message refinement.

Figure 3.9.: These diagrams are *fragment consistent*, but the messages order is inconsistent.

Message order inconsistencies pose the one remaining issue that is usually addressed in terms of traces. Those rather strict structural rules defined above, however, have one major advantage: Partial event orders now suffice to detect message order inconsistencies. As mentioned in the previous chapter, partial orders are an adequate semantics for basic sequence diagrams. The above structural rules preserve any combined fragments that modify traces over messages that appear in the master and in the refining diagram at the same time. Thus, the effect of those fragments becomes unimportant when it comes to the comparison of sequence diagrams that refine each other. The next section describes how to reduce regular sequence diagrams to partial event orders, and how a combination of Petri net execution and the detection of strongly connected components is used to localize misordered messages.

3.4. Message Order Consistency

Although the actual semantics of the twelve UML operators is quite different, they differ only slightly in their influence on the partial event order. Most operators such as *opt*, *loop*, *consider*, *ignore*, *critical*, *break*, *neg*, and *assert* contain only one

operand and have no influence on the partial order at all. Thus, each fragment with one of those operators can be replaced with the fragment or basic sequence diagram that they contain. The *alt* operator that allows an arbitrary number of operands specifies that only one of its operands is executed, so that there are no relationships between events that belong to different operands. Since the latter is true for the *par* operator, every *alt* operator can be replaced with a *par* operator without modifying the partial order. In conclusion, it suffices to separately consider only *par*, *seq*, and *strict* fragments.

Deriving the Partial Order

To derive the partial order from a sequence diagram s, a function $||.||$ is introduced that assigns a partial order to all basic sequence diagrams and combined fragments of s. For a basic sequence diagram $b \in B_s$ that partial order is given by $||b|| = <_b$. For combined fragments this partial order is defined by structural induction, depending on the type of the fragment.

Since the operands of a *par* fragment are completely independent, there is no order between events of different operands. For a *par* fragment $c \in C_s$ the partial order is the union of the partial orders of the fragment's operands, *i.e.*,

$$||c|| \;=\; \bigcup_{o \in O_s(c)} ||o||.$$

The *strict* operator enforces that every event of one operand occurs before any event of a following operand. The event order of each operand is maintained. Since the event order depends on the operand order the following shorthand is used to denote ordered pairs of operands of a fragment c:

$$O_s^*(c) = \{(o, o') | o, o' \in O_s(c) \wedge o <_s^* o'\}.$$

For a *strict* fragment $c \in C_s$ the partial order is given by

$$||c|| \;=\; \bigcup_{o \in O_s(c)} ||o|| \;\cup\; \bigcup_{(o,o') \in O_s^*(c)} E(M_s(o)) \times E(M_s(o')).$$

The *seq* operator leads to a weak sequencing of the operands. As mentioned in the previous chapter, the order of each operand is maintained, and all events that belong to the same instance are ordered according to the order of the operands. For a set E of events and an instance i that appear both in a sequence diagram s, $E|_i$ refers to the subset of E that contains only events of i, *i.e.*,

$$E|_i = \{e \in E | \iota_s(e) = i\}.$$

For a *seq* fragment $c \in C_s$ the partial order is given by

$$||c|| \;=\; \bigcup_{o \in O_s(c)} ||o|| \;\cup\; \bigcup_{i \in I_s,\ (o,o') \in O_s^*(c)} E(M_s(o))|_i \times E(M_s(o'))|_i.$$

The partial orders of sequence diagram now complete the definition of the refinement relationships. Two sequence diagrams are consistent with respect to their partial orders, if these orders are compatible with each other, *i.e.* the orders do not contain any contradicting relations between events, which belong to both diagrams.

Definition (Intra-level/Inter-level Refinement):
Sequence diagram t is an intra-level/inter-level refinement of sequence diagram s if t is an intra-level/inter-level message refinement of s, t and s are fragment consistent, and $||t|| \cup ||s||$ is acyclic.

Following the definition of inter-level and intra-level refinement it seems straightforward to search for cyclic dependencies to detect misordered messages. These dependency cycles, however, usually involve more messages than just the misordered ones, so that further manual investigations are necessary. The remainder of this chapter introduces Petri nets and shows how a combination of the detection of strongly connected components, which cover the dependency cycles, and a Petri net execution leads to a very accurate set of misordered messages.

Petri Nets

According to Murata [22], Petri nets are a *graphical and mathematical modeling tool [...] for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic.* Thus, Petri nets qualify for the analysis of sequence diagrams in the context of this diploma thesis. Basically, a Petri net is a bipartite directed graph with two different kinds of nodes: *places* and *transitions*. The following definition provides the mathematical basis.

Definition (Petri Net):
A *Petri net* is a tuple

$$n = (P, T, F)$$

where

- P is a set of places,
- T is a set of transitions, and
- $F \subset (P \times T) \cup (T \times P)$ is a flow relation.

Figure 3.10 shows a graphical representation of a Petri net. Places are usually drawn as circles and transitions as rectangles. Arrows connect places and transitions according to the flow relation, so that there are no arrows that connect places to places or transitions to transitions.

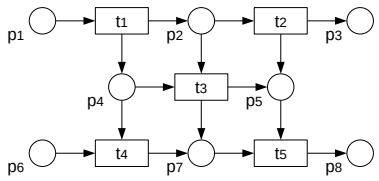


Figure 3.10.: A simple Petri net.

Definition (Input/Output Places):
Let n be a Petri net and let $t \in T$ be a transition of n. $\bullet t$ and $\bullet n$ refer to the *input places* of t and n, *i.e.*,

$$\bullet t = \{p \in P \mid (p,t) \in F\} \text{ and}$$
$$\bullet n = \{p \in P \mid \forall t \in T.\ (t,p) \notin F\}.$$

$t\bullet$ and $n\bullet$ refer to the *output places* of t and n, *i.e.*,

$$t\bullet = \{p \in P \mid (t,p) \in F\} \text{ and}$$
$$n\bullet = \{p \in P \mid \forall t \in T.\ (p,t) \notin F\}.$$

p1 and p6 are the input places and p3 and p8 are the output places of the net in Figure 3.10. p2 and p4 are the input places of transition t3 and p5 and p7 are the output places of that transition. Figure 3.11 shows a *marked* Petri net, where places p1 and p6 contain so-called *tokens*.

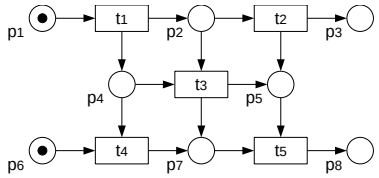


Figure 3.11.: A marked Petri net.

The *marking* of a Petri net indicates the current state of the net. A single place is in general allowed to contain an arbitrary number of tokens. Since this is not necessary for this diploma thesis, the following definition is based on sets instead of multisets [22].

Definition (Marking):
Let n be a Petri net. A *marking* x of n is a subset of the places, *i.e.*, $x \subseteq P$. The marking that contains only the input places of n is called the *initial marking* and the marking that contains only the ouput places of n is called the *final marking* of n.

The marking of a Petri net may change if a transition is *fired*. The transition t1 in Figure 3.11 may fire since all of its input places are marked. The transition t4 may not fire because there is no token in p4. If transition t1 is fired p1 is removed from the current marking while p2 and p4 become marked.

Definition (Transition Firing Rule):
Let n be a Petri net, $t \in T$ a transition of n, and let x be the current marking of n. t is said to be *enabled* if all of its input places are marked, *i.e.*,

$$\bullet t \subseteq x.$$

An enabled transition may fire and change the current marking to a next marking x' of n, which is written as

$$x \xrightarrow[n]{t} x'.$$

The new marking x' contains all places from the previous marking x except for the input places of t plus the output places of t, *i.e.*,

$$x' = (x \setminus \bullet t) \cup t \bullet .$$

The previous definitions are simplifications of definitions provided by Murata [22]. The following definitions are based on the approach of van der Aalst *et al.* [2, 41]. The usage of these definition, however, is quite different in this thesis (*cf.* Chapter 5).

Definition (Petri Net Trace):
Let n be a Petri net and let $t_1, \ldots, t_k \in T$ be transitions of n. $\langle t_1, \ldots, t_k \rangle$ is a *trace* of n if there are markings $x_1, \ldots, x_{k+1} \subset P$ of n, so that x_1 is the initial marking of n and transition t_i is enabled in marking x_i and firing this transition leads to marking x_{i+1}, *i.e.*,

$$\forall i \in \{1, \ldots, k\}.\ x_i \xrightarrow[n]{t_i} x_{i+1}.$$

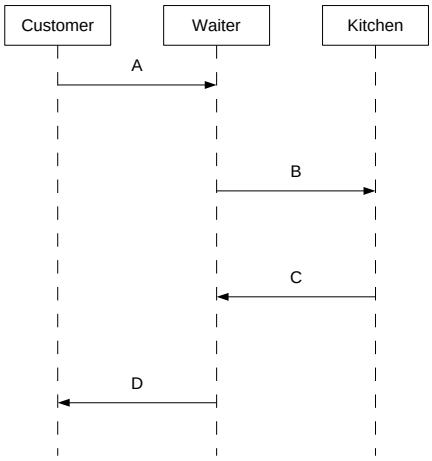
A trace is said to be *proper* if it contains each transition of the net exactly once and if it reaches the final marking of the net. A trace is said to be *improper* if it is neither proper nor extendable to a proper trace. A Petri net is said to be *well-formed* if none of its traces is improper. The simple example that is shown in Figure 3.11 is not proper. The trace $\langle t_1, t_3, t_5 \rangle$ leads to the marking $\{p_6, p_8\}$ which is not the final marking. Removing the transition t_3 from the net leads to a proper net.

Locating Misordered Messages

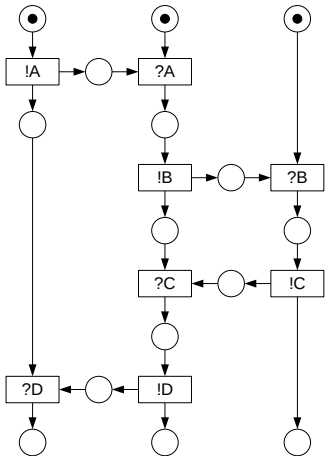
Kluge [24] and Graubmann *et al.* [10] suggest Petri nets as a semantic model for basic sequence diagrams. Each transition of a Petri net refers to an event of a sequence diagram and a place between two transitions exists if the corresponding events are ordered. In doing so a transition is only eligible to fire if all transitions that refer to events that have to occur before also fire before, so that the sequence diagram and the Petri net are equivalent.

Since this thesis reduces regular sequence diagrams to partial orders, the Petri net transformation is applicable, too. The transformation of *par*, *strict*, and *seq* fragments is analogous to the definition of the partial order function $||.||$. Figure 3.12 shows two sequence diagrams and the Petri nets that represent their partial event orders. Without the need to consider repetitive, alternative, or optional behavior,

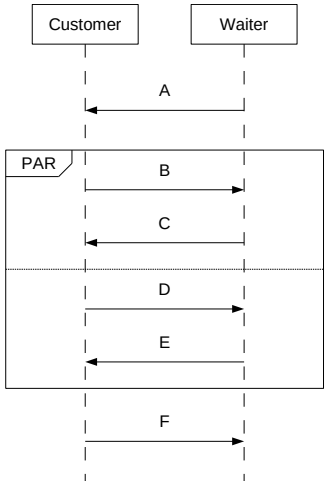
the Petri net representation remains straightforward. The traces of such a Petri net are in general not equivalent to traces of the original sequence diagram, since, for example, there is a difference between *par* and *alt*. This equality is of no importance though, because only the order information is relevant.



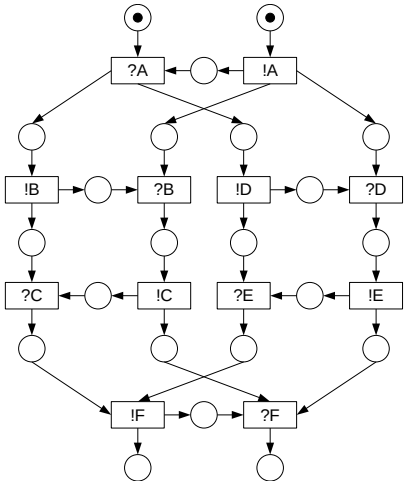
(a) A basic sequence diagram.



(b) A straightforward Petri net representation.



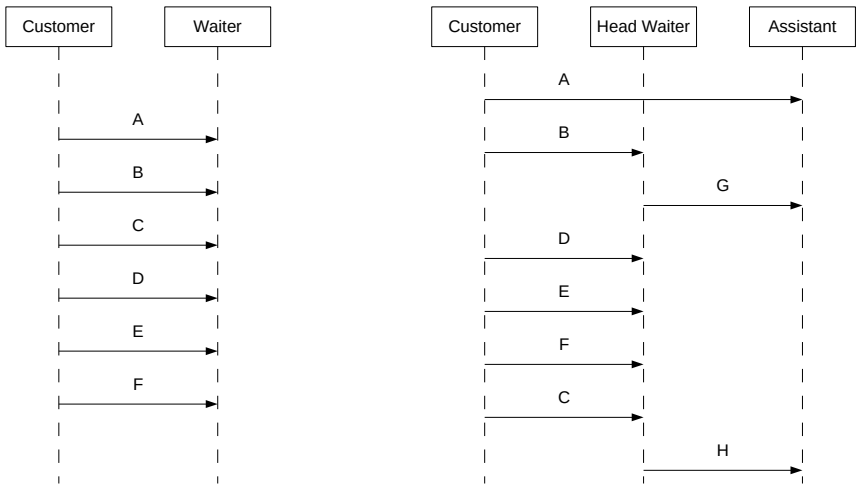
(c) A sequence diagram with a *par* fragment.



(d) Parallel nets are indepedent.

Figure 3.12.: Sequence diagrams and their Petri net representations.

All Petri nets that are created from sequence diagrams belong to the subclass of so-called *Marked Graphs* [22]. Places of such marked graphs have at most one incoming and one outgoing arc, so that transitions do not share common places. The most important property, however, is that all nets that are based upon partial orders are proper, since an execution of such a net is equivalent to a construction of a total order that extends its underlying partial order.



(a) The master diagram.

(b) The inter-level message refinement diagram.

Figure 3.13.: Correct inter-level message refinement with inconsistent message order.

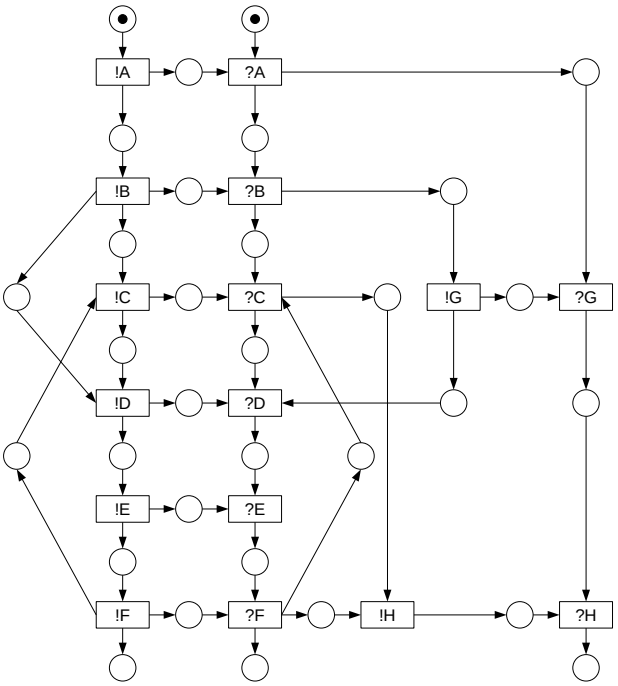


Figure 3.14.: The Petri net representing the union of the partial event orders.

The remainder of this chapter uses the example in Figure 3.13 to show how misordered messages are detected. Figure 3.13(a) shows the master diagram and Figure 3.13(b) shows a correct inter-level message refinement, *i.e.*, all structural rules

are fulfilled. The message order, however, is inconsistent, since the message C obviously occurs at different locations in the master and the refining diagram. To detect this order inconsistency both diagrams are reduced to a Petri net that represents their partial event order. These Petri nets are joined so that a single net represents the union of both orders. The resulting net is shown in Figure 3.14. The next step is to evaluate whether the union of the partial event orders remains acyclic. Tarjans algorithm [39] is used to compute strongly connected components. A cycle exists if there is a component with more than one element. The result is shown in Figure 3.15. There are two cyclic dependencies that contain four transitions each. The first cycle contains the transitions *!C*, *!D*, *!E*, and *!F* and the second cycle contains the transitions *?C*, *?D*, *?E*, and *?F*.

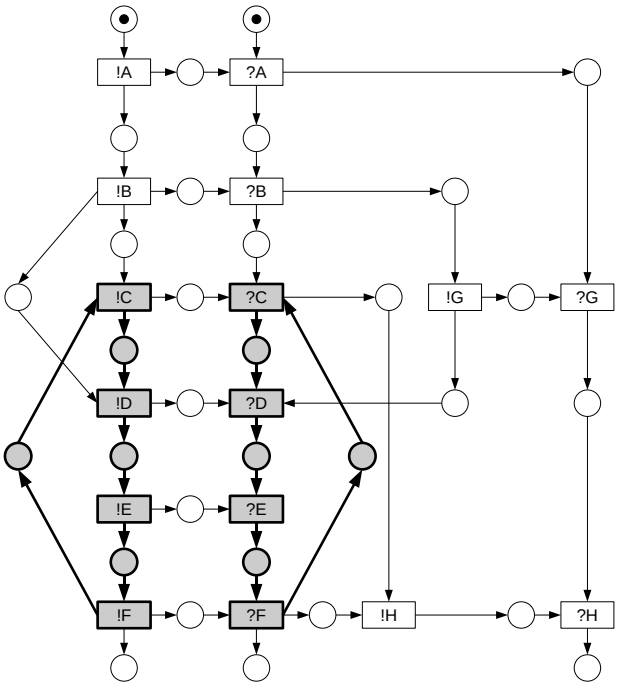


Figure 3.15.: The Petri net contains two strongly connected components containing four transitions each.

The computation of cyclic dependencies does not only reveal that the refinement in this example is incorrect but also isolates the messages that are involved. The message C, D, E, and F need to be checked. The result, however, can still be improved,*i.e.* made more precise, by executing the Petri net. As mentioned above Petri nets based on partial orders always arrive at the final marking. In this example, however, there is a cyclic dependency, *i.e.*, the event relation is not a partial order, and the execution will stall with a marking different from the final marking. If at least two transitions depend on each other neither of them can fire since they require the other transitions to fire first. The idea is to execute the net until there is no enabled

transition left. If the current marking is not the final marking, all transitions that have at least one marked input place indicate a possible inconsistency. Figure 3.16 shows that the execution stops before the final marking is reached. A trace that leads to this marking is, for example, $\langle !A, ?A, !B, ?B, !G, ?G \rangle$ and the messages that are involved in this inconsistency are C, D, and H.

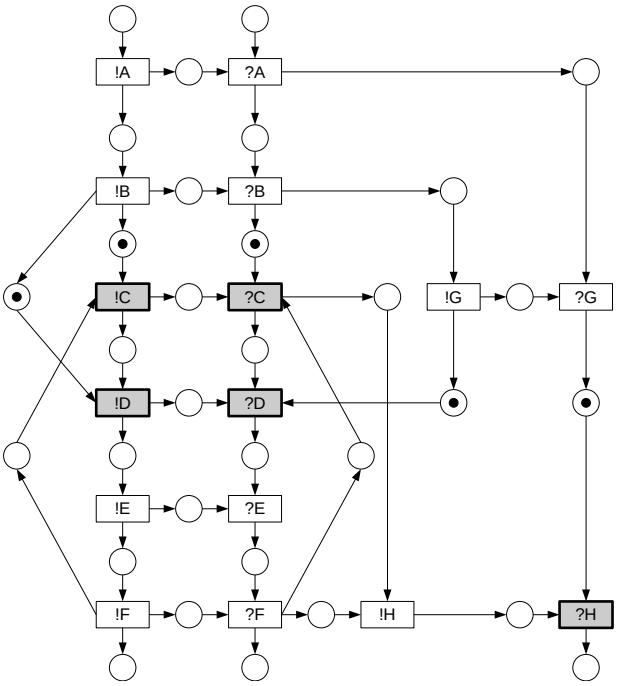


Figure 3.16.: Cyclic dependencies impede a proper execution of the Petri net.

The Petri net execution and the computation of strongly connected components yield different results. These results, however, will always have a common core, since the cyclic dependency prevents the net from beeing executed properly. In this example only the messages C and D are present in both results.

This order inconsistency, however, must be solved manually, since it is unclear which of the concerned sequence diagrams is actually wrong. The approach described above localizes the first essential difference between both diagrams and, thus, provides a good starting point for further investigations.

4. Implementation

This chapter describes how the theoretical concepts of the previous chapters are realized and evaluated. As mentioned in the introduction, the software team that supervises this diploma thesis develops embedded real-time systems. Their modeling tool is based on the ROOM framework [3, 4], where active classes, so called *capsules*, are first-class citizens. This modeling tool provides an API that allows to implement customized plugins. These plugins range from straightforward macros that speed up recurring development steps to complex validating and consistency checking algorithms.

To realize the refinement concept as platform-independent as possible, it is not directly integrated into the modeling tool. Instead, an external server application covers the consistency checking and a rather thin client is implemented as a plugin for the modeling tool (*cf.* Figure 4.1). This client (Section 4.1) converts a subset of the current model into an XML representation (Section 4.2), which is then processed by the server application (Section 4.3).

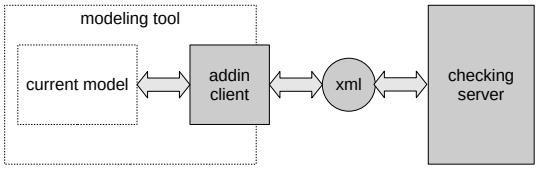
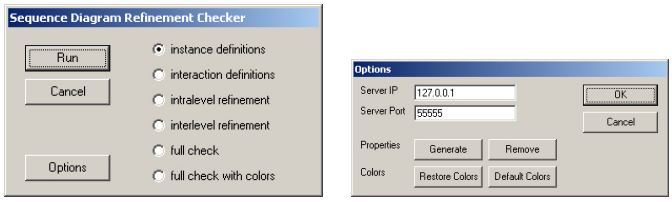


Figure 4.1.: A client communicates with a checking server.

4.1. Client

The advantage of having a thin client is that only minor modifications are necessary to apply the refinement concept to different modeling tools. The main task of the client is to generate and export an XML representation of the sequence diagrams, which the server should check. This XML representation is straightforward (*cf.* Section 4.2) and is generated by a recursive traversal of the model. Furthermore, the client allows the developer to specify which sequence diagrams refine each other and which of those should be checked. The modeling tool itself does not support this, but allows to add meta properties to every model element, so that plugins have a possibility to store this necessary information.



(a) The client main dialog. (b) The options dialog.

Figure 4.2.: The user interface of the client application.

Figure 4.2(a) shows the main dialog of the client application. The user can specify a *checking depth* that determines which definitions presented in the previous chapter are verified. There are five different checking levels, where each level includes the checks of its predecessor:

instance definitions This check assures that the hierarchy conforms with the definition of the instance hierarchy in Chapter 3. This is always the first check, since the correctness of the structural rules depends on a well-formed instance hierarchy.

interaction definitions This check assures that each sequence diagram is well-formed, *i.e.*, that all instances of a sequence diagram belong to the same abstraction layer and that no illegal messages exist.

intralevel refinement As mentioned above developers can specify pairs of sequence diagrams that intra-level refine each other. This check assures for each of those pairs that the definition of intra-level message refinement is fulfilled, and that both sequence diagrams are *fragment complete* and *fragment consistent*. This check, however, does not consider any message order.

interlevel refinement Similar to the intra-level check, this check verifies pairs of inter-level refining sequence diagrams.

full check (with colors) This check finally evaluates for every pair of refining sequence diagram whether the message order is consistent or not. This check optionally uses colors to highlight erroneous elements.

Besides the checking depth the graphical user interface provides an options dialog (Figure 4.2(b)) that allows the user to specify the location where the server application is running. Furthermore, this options dialog can also be used to generate and delete the meta properties that are used to store the refinement relationships. The client displays any discrepancies in the error log window of the modeling tool. Since many inconsistencies, such as an illegal message, involve only a single model element the client is able to focus the modeling on that erroneous element, if the user selects one of those log entries. The modeling tool, however, is not able to focus on several elements at the same time, so that errors that involve a set of elements need

manual handling. In the case of a set of misordered messages the client optionally highlights those messages in red. In order to avoid interferences with other plugins that depend on colors, this color mechanism is only optional and the options dialog (Figure 4.2(b)) provides two functions to restore the previous colors or the default color of changed elements.

4.2. XML

This section introduces the XML model representation, which is exchanged between the client and the server application. The XML data contains the instance hierarchy and a set of sequence diagrams to be validated.

```
<instances>
    <instance label="Customer" level="0"/>
    <instance label="Restaurant" level="0/>
    <!-- ... -->
    <instance label="Kitchen" level="2" master="Kitchen"/>
    <instance label="Barkeeper" level="2" master="Barkeeper" parent="Kitchen"/>
</instances>
```

Figure 4.3.: XML representation of an instance hierarchy.

Figure 4.3 shows a part of the XML representation of the instance hierarchy in Figure 3.3. There is an `instance` element for each instance. The `label` attribute holds the instance's name, the `level` attribute refers to the abstraction level and the `master` and `parent` attributes refer to the optional master and parent instance, respectively.

```
<sd label="sequence diagram 2.1(b)" instances="Customer Waiter Kitchen">
    <fragment type="seq">
        <fragment type="par">
            <fragment type="basic">
                <fragment type="lifeline" label="Customer">
                    <fragment type="send" label="order drink"/>
                </fragment>
                <fragment type="lifeline" label="Waiter">
                    <fragment type="receive" label="order drink"/>
                </fragment>
            </fragment>
        <!-- ... -->
    </fragment>
</sd>
```

Figure 4.4.: XML representation of a sequence diagram.

The XML representation of sequence diagrams is straightforward due to their hierarchical character. Figure 4.4 shows a part of the XML representation of the sequence diagram in Figure 2.1(b). The most important element is the `fragment` element, which is used to represent combined fragments, basic sequence diagrams, lifelines, coregions, and events. Depending on its type, a fragment may contain an arbitrary

number of fragments. The `par` fragment in the above example may contain any other combined fragment or basic sequence diagram. A `basic` fragment, which refers to a basic sequence diagram, contains `lifeline` fragments and a `lifeline` fragment contains `send`, `receive`, and `coregion` fragments. The order of a sequence of fragments refers to the visual order of the operands or events. Complete versions of these two examples can be found in the appendix. Furthermore, the appendix contains an XML schema definition for the model representation.

4.3. Server

The server checks whether the given sequence diagrams and their refinement relationships conform to the definitions of the previous chapter. The client/server communication is based on the Java Socket API. The server listens on a specific port and waits for a client to connect. A new `ConnectionHandler` is created for every incoming connection, so that the server is able to handle several clients at the same time. The remainder of this section gives a short overview of the components of the server application.

Server and Checker Classes

Server The Server class itself is a rather simple. Its only purpose is to create a ConnectionHandler for every connected client.

ConnectionHandler As presented in Figure 4.5 the ConnectionHandler class is the central component of the server application. A ConnectionHandler waits until the corresponding client transmits a complete model and then runs the desired checks. Any error message that is generated by one of the checking components is forwarded to the client. The ConnectionHandler closes the connection and terminates itself when the check is completely evaluated. The different checks are evaluated in the order of the checking depth and if a sequence diagram fails a check, this diagram is not considered by subsequent checks to suppress subsequent errors.

StructureChecker StructureChecker is an abstract basic class for four different structure checkers, which refer to the first four levels of the checking depth mentioned in Section 4.1. The different structure checkers use the XML representation to generate the model representation whilst checking if the different structural definitions are fulfilled (*cf.* Chapter 3).

InstanceDefinitions The InstanceDefinitions class checks whether the given instance relationships conform to the definition of an instance hierarchy that is given in Chapter 3.

InteractionDefinitions The InteractionDefinitions class assures that all sequence diagrams are well-formed, *i.e.*, all instances of a single diagram belong to the same level of abstraction and if no illegal messages exist.

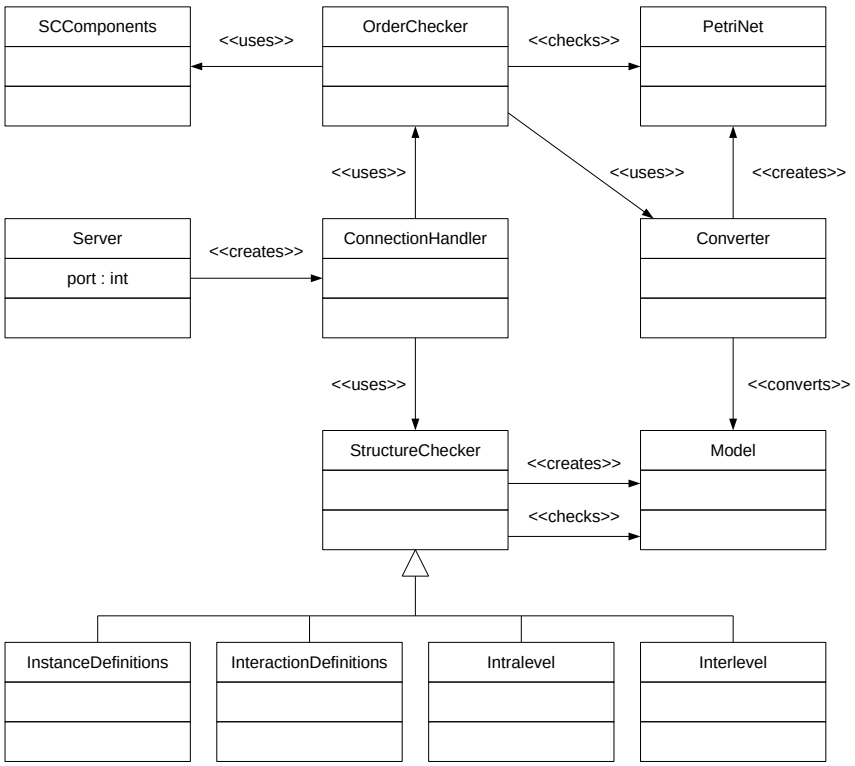


Figure 4.5.: Overview of the server and checker classes.

IntraLevel The IntraLevel class validates pairs of master and intra-level refinement diagrams. This check verifies the definitions of *intra-level message refinement*, *fragment consistency*, and *fragment completeness*.

InterLevel The InterLevel class checks master and inter-level refinement diagrams in almost the same manner as IntraLevel validates intra-level refinement diagrams.

OrderChecker The OrderChecker verifies the message order for all refining diagrams by combining a Petri net simulation with a computation of strongly connected components.

Converter The Converter class is used by the OrderChecker to create a Petri net representation of the partial order of sequence diagrams.

PetriNet The PetriNet class together with several subclasses are used to create a Petri net representation of a sequence diagram. These subclasses are explained later.

SCComponents The SCComponents class is used by the OrderChecker to compute strongly connected components within a Petri Net. The intersection of the re-

sult of the Petri net simulation and the strongly connected components localize any misordered messages very well.

Model Element Classes

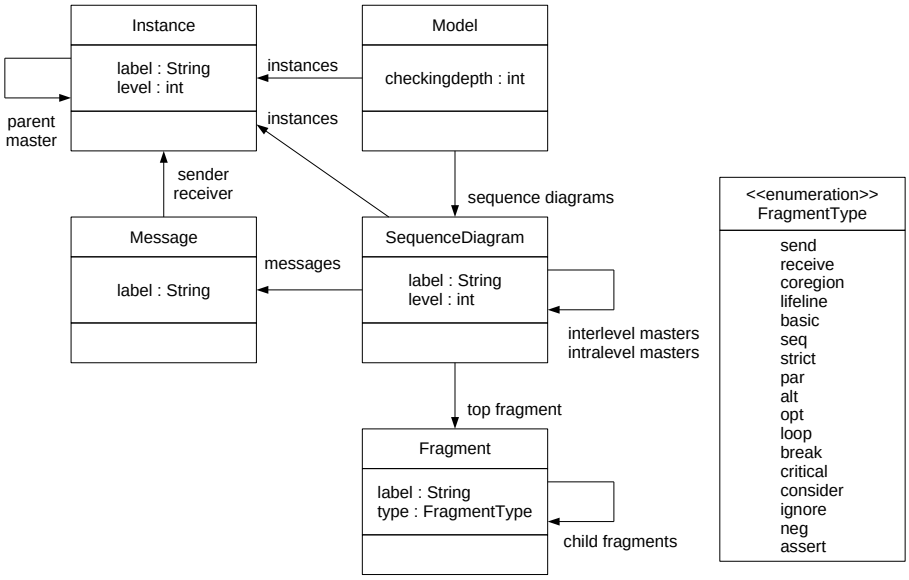


Figure 4.6.: Overview of the model element classes.

Model The Model is recreated by deserializing the XML representation that is exported by the client application. A model consists of a set of instances and a set of sequence diagrams. The checking depth specifies which checks should be executed.

Instance An Instance has a label and a level that refers to the abstraction layer. Each instance also has optional references to its parent and master instance. These references implement the instance hierarchy (*cf.* Chapter 3).

SequenceDiagram A SequenceDiagram has a label that contains the name of the diagram. The level attribute refers to the level of the instances within the sequence diagram. Each sequence diagram has optional references to its intra-level and inter-level master diagrams. A root fragment represents the structure of the sequence diagram.

Message A Message is characterized by its label. According to the refinement concept a single message may belong to several sequence diagrams that possibly relate the message to different sending and receiving instances.

Fragment A Fragment represents a part of the structure of a sequence diagram. Fragments are identified by their label and, similar to messages, a single fragment may appear within several sequence diagram. Each fragment has a type and represents either a send event, a receive event, a coregion, a lifeline, a basic sequence diagram, or one of the twelve UML 2.0 operator kinds.

Petri Net Classes

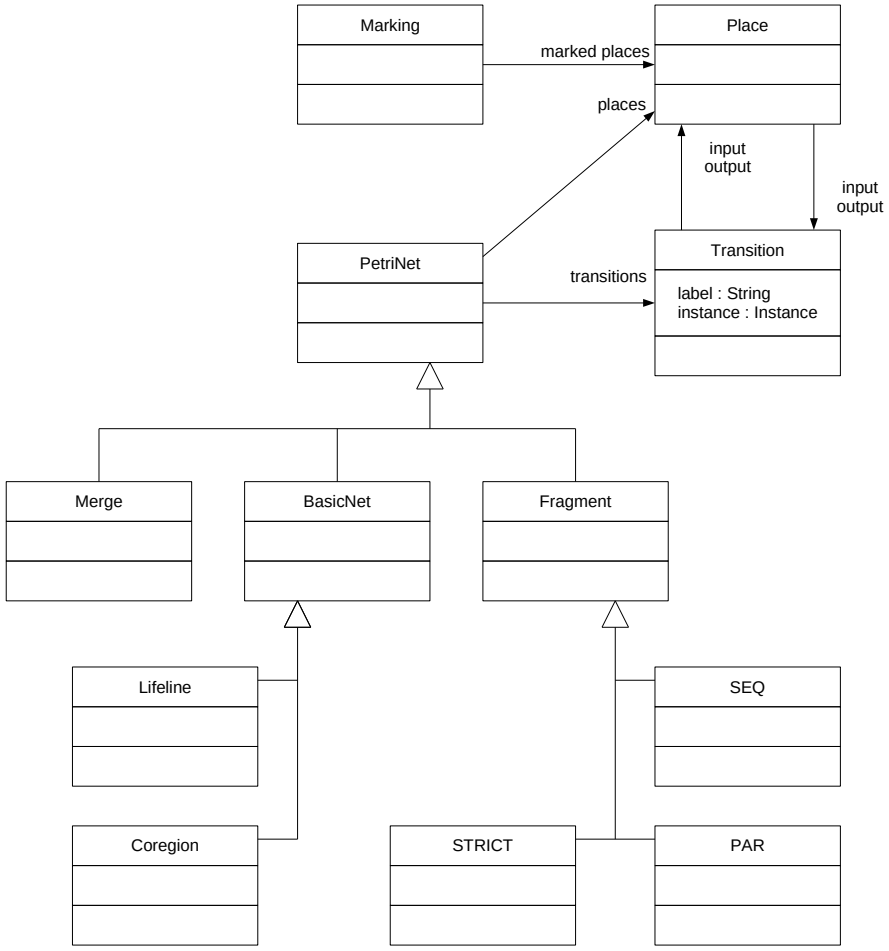


Figure 4.7.: Overview of the Petri net classes.

Figure 4.7 shows the different Petri net classes that are used to derive and represent the partial order of sequence diagrams. Petri nets are constructed from sequence diagrams by a recursive traversal of the sequence diagram structure. Thereby each subclass of PetriNet is used to convert a specific element of a sequence diagram.

PetriNet PetriNet is an abstract class that serves as a basis for other classes. As

introduced in the previous chapter, a Petri net consists of a set of places and transitions that compose a bipartite directed graph. This class provides basic methods to add, remove, or connect places and transitions.

Transition Transition objects represent send or receive events of a sequence diagram. Each transition has a reference to the instance to which the event belongs. A transition may fire if all of its input places belong to the current marking of the net.

Place Place objects are used to connect at least two transitions and thus represent the partial event order.

Marking Marking represents the marking of a net, *i.e.* a marking contains a set of places that are considered to be marked. Markings are used during the Petri net simulation.

BasicNet A BasicNet is created from a basic sequence diagram that only contains lifelines, coregions, and send and receive events.

Lifeline Lifeline represents a single lifeline of a basic net. Lifeline objects are created by BasicNet classes during the conversion routine.

Coregion A Coregion contains a set of transitions that belong to the same instance but are not ordered at all. Coregion objects are created from Lifeline objects if the corresponding lifeline contains a coregion.

Fragment A Fragment is created from a list of Petri nets, which represent the operands of that fragment. Fragment is the abstract basic class for the three subclasses SEQ, STRICT, and PAR.

PAR The PAR class represents the parallel operator. Since parallel operands are completely independent the resulting Petri net is a simple union of the nets representing the different operands.

STRICT The STRICT class implements the strict sequencing operator. All sinks of a Petri net are connected via places with the sources of the following net. A transition is called sink or source if it has no output or input places, respectively. In doing so all transitions of the first net have to be fired before any event of the second net. This reflects exactly the semantics of the strict operator.

SEQ The SEQ class provides the weak sequencing mechanism. The construction is similar to the construction of the strict sequencing operator with the only difference that sinks and sources of different nets are only connected if they belong to the same instance. Thus, events that belong to the same lifeline can only occur in the operand order, and events of different lifelines may occur in any order.

Merge A Merge Petri net is created from several sequence diagrams that refine each other. Each of these sequence diagram is converted into a Petri net and the Merge net is the result of a straightforward union of these Petri nets. A cyclic dependency between the event order of at least two sequence diagrams leads to a cyclic dependency between several transitions, which impedes a proper execution of the resulting net.

4.4. Complexity

All checking algorithms are derived from the formal definitions of the corresponding structural rules of Chapter 3. Since many rules require to check the containedness of different objects in different sets, hash tables are used as basic data structures to provide a look-up in constant time.

Illegal Message Check Let s be a sequence diagram. Each message of s has to fulfill five conditions that are verified in $O(1)$, so that the complete check is evaluated in $O(|M_s|)$.

Intra-level Instance Refinement Check Let t be an intra-level instance refinement of s. This check requires to evaluate for each instance of t whether it also belongs to s or if its parent instance belongs to t. Since each instance is verified in $O(1)$, the complete check runs in $O(|I_t|)$.

Inter-level Instance Refinement Check Let t be an inter-level instance refinement of s. This check assures that each instance of t has either a corresponding master instance in s or a parent instance in t. Similar to the intra-level instance refinement check each instance is checked in $O(1)$ and the complete check is computed in $O(|I_t|)$.

Intra-level Message Refinement Check Let t be an intra-level message refinement of s. Each message of s that has sending and receiving instances that also belong to t must also belong to t and vice versa. Each message in s and t is checked in $O(1)$. Furthermore, the sending and receiving instances of messages that appear in both diagrams have to be equal in both diagrams. This is evaluated in $O(1)$ for each of those messages. The intra-level message refinement check runs in $O(|M_s| + |M_t| + |M_s \cap M_t|)$.

Inter-level Message Refinement Check Let t be an inter-level message refinement of s. Each message of s that has sending and receiving instances with refining instances in t must belong to t. Furthermore, each message of t that has sending and receiving instances that refine different instances in s must also belong to s. Finally, sending and receiving instances of messages that appear in both diagrams must conform to the instance hierarchy. The inter-level message refinement check runs in $O(|M_s| + |M_t| + |M_s \cap M_t|)$.

Fragment Completeness Check Let s and t be two *fragment complete* sequence diagrams. Each fragment in one of these diagrams that covers messages that appear in both diagrams has to appear in the other diagram, too. Let m_{max} be the maximum number of messages within a fragment in s and t. A check of a single fragment runs in $O(m_{max})$ in the worst case, so that the complete check is evaluated in $O((|C_s| + |C_t|) \cdot m_{max})$.

Fragment Consistency Check Let s and t be two *fragment consistent* sequence diagrams. The check requires to iterate through the operands of each fragment that appears in both diagrams. Each operand that belongs to such a fragment in one diagram is compared with each operand that belongs to the same fragment in the other diagram. Comparing two operands means to check whether they contain the same messages with respect to the messages that appear in both diagrams. Let m_{max} and o_{max} be the maximum number of messages and operands within a fragment, respectively. Let c^* be the number of fragments that appear in both diagrams. The fragment consistency check runs in $O(c^* \cdot o_{max}^2 \cdot m_{max})$.

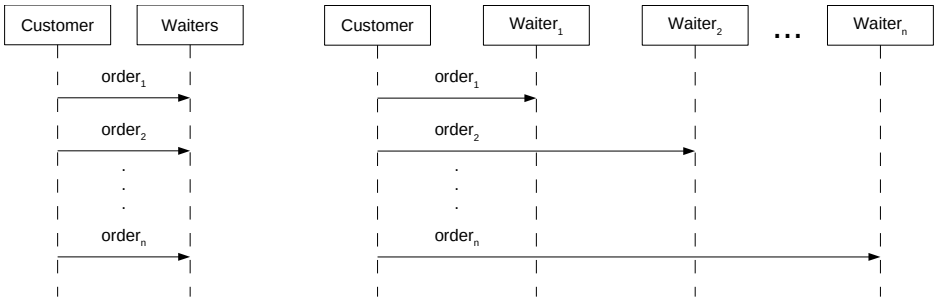
Partial Order / Petri Net Construction Chapter 3 gives an inductive definition of the $||.||$ function that is used to derive the partial order of sequence diagrams. Petri net are used to represent these partial orders. The worst case during the construction occurs when a *strict* fragment is involved, because each event of one operand has to be connected to each event of the following operands. The partial order is computed $O(m^2)$, where m is the number of messages. The actual implementation, however, is more efficient, since the transitivity of a partial order is inherent in the Petri net representation (*cf.* [24, 10]). During the conversion of a *strict* fragment, for example, it suffices to connect the sinks of an operand with the sources of the directly following operand.

Petri Net Execution Let s and t be two sequence diagrams that refine each other. The Petri net that represents the union of the partial orders contains $n = 2 \cdot |M_s \cup M_t|$ transitions. Each transition fires exactly once during the execution. Let p_{max} be the maximum number of input and output places of a transition. p_{max} is usually very small compared to the number of transitions. Checking and firing a transition is computed in $O(p_{max})$. Searching for the next firing transition requires to iterate through all transitions in the worst case. The Petri net execution is computed in $O(n^2 \cdot p_{max})$.

4.5. Benchmarks

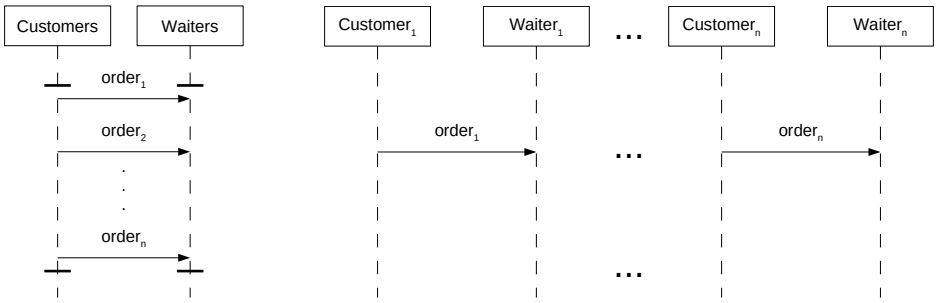
This section contains some benchmarks to demonstrate the efficiency of the structural approach. Two examples, shown in Figure 4.8 and Figure 4.9, are used to compare the duration of the structural approach with a complete trace enumeration that is required by other refinement approaches as described in the next chapter.

Furthermore, the structural approach is compared with the *inheritance checker* of the WOFLAN tool by van der Aalst *et al.* [13, 12, 2, 41]. The inheritance checker is based on labeled transition systems and branching bisimulation (*cf.* Chapter 5).



(a) Abstract view. (b) Detailed view.

Figure 4.8.: Several waiters replace the single waiter of the first example.



(a) Abstract view. (b) Detailed view.

Figure 4.9.: Coregions in the second example allow the messages to appear in any order.

Both examples show a correct inter-level refinement, so that no inconsistency aborts the evaluation early without measuring the complete time. In the first example the single waiter is replaced by n different waiters and each waiter receives the corresponding order message. In the second example the customer is also replaced by n different customers. Furthermore, the second example contains two coregions that allow the messages to appear in any order. The second example is also used in Chapter 2 to show that trace sets are of factorial complexity.

Both examples are evaluated for increasing values of n, *i.e.*, increasing number of messages and instances. Table 4.1 shows the results for the first example. The value of n and the number of messages is given by the first column. The second column contains the number of instances. The third column shows how long the structural approach takes for a complete evaluation. This includes the verification of all structural rules that cover inter-level refinement and the construction and

simulation of the Petri net. The fourth column shows the number of valid traces for the given value for n. The time in parenthesis behind the number of traces states how long it takes to enumerate those traces.

WOFLAN's results are shown in the last column. These results were taken manually, since the built-in *inheritance checker* does not provide any duration details. All benchmarks were run on an AMD Athlon™ 64 3200+ (2GHz) with 1GByte of main memory. Some table entries are empty because the duration was either to short (in the case of WOFLAN) or to long to be measured.

n / # messages	# instances	duration	# traces (duration)	WOFLAN
6	7	8 ms	132 (14 ms)	-
7	8	8 ms	429 (37 ms)	-
8	9	8 ms	1430 (119 ms)	-
9	10	9 ms	4862 (421 ms)	-
10	11	9 ms	16796 (1507 ms)	-
11	12	9 ms	58786 (5563 ms)	-
12	13	9 ms	208012 (22718 ms)	$<$ 500 ms
20	21	19 ms	- (-)	1000 ms
30	31	31 ms	- (-)	4500 ms
40	41	43 ms	- (-)	27000 ms
50	51	57 ms	- (-)	88000 ms
60	61	76 ms	- (-)	$>$ 360000 ms
100	101	183 ms	- (-)	-

Table 4.1.: Verifying the sequence diagrams in Figure 4.8.

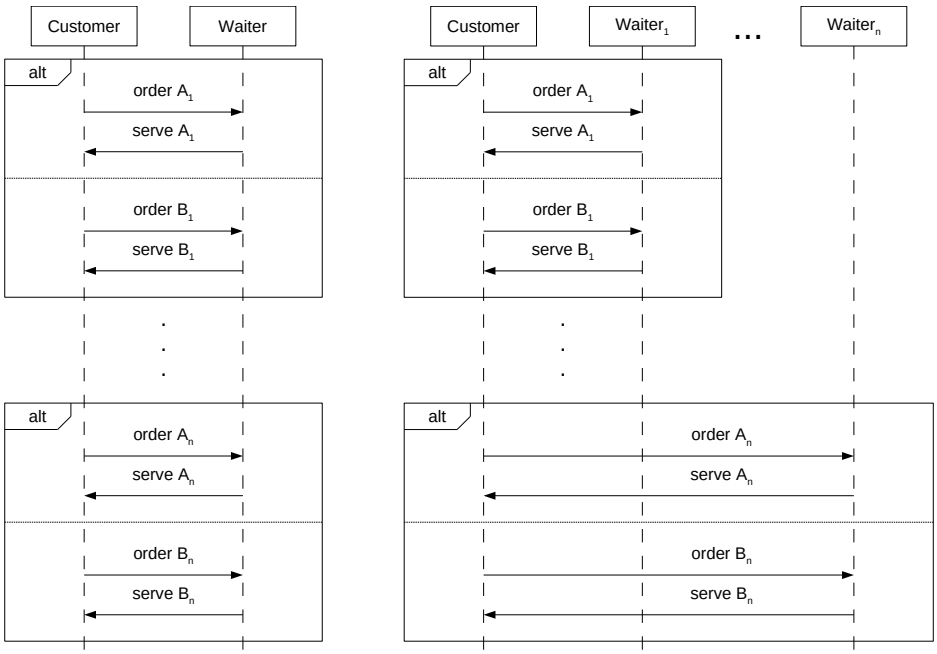
The structural approach presented in this thesis is able to validate the first example without a remarkable delay even if it contains 100 messages. The inheritance checker and trace based approaches can handle only small examples. The results of the second example (Table 4.2) show that the structural approach is even insensitive to parallelism. Compared to the first example there is a only minor difference that is caused by the fact that the second example contains almost the double number of instances. The validation of an example without parallelism but with the same number of instances would take just as long.

n / # messages	# instances	duration	# traces (duration)	WOFLAN
3	6	6 ms	90 (47 ms)	-
4	8	6 ms	2520 (250 ms)	-
5	10	7 ms	113400 (11594 ms)	$<$ 500 ms
6	12	7 ms	7484400 (672297 ms)	1500 ms
7	14	8 ms	681080400 (-)	42000 ms
50	100	96 ms	- (-)	-
100	200	333 ms	- (-)	-

Table 4.2.: Verifying the sequence diagrams in Figure 4.9.

Figure 4.10 shows the last example of this chapter. Since the first two examples

do not contain any combined fragments, the most complex *fragment consistency* rule is not applied. The third example contains n *alt* fragments with four message each. The results in Table 4.3 show that the complex fragment consistency rule has a minor impact on the duration. For $n = 25$ the diagram contains 100 messages and the evaluation takes 190ms. This is even faster than the evaluation of the previous example with 100 messages that does not contain any fragments at all. Table 4.3 contains no entries for WOFLAN's inheritance checker, since this tool does not support combined fragments, yet.



(a) Abstract view. (b) Detailed view.

Figure 4.10.: Consecutive *alt* fragments.

n / # fragments	# messages	# instances	duration
20	80	21	130 ms
25	100	26	190 ms
40	160	41	400 ms
60	240	61	840 ms
80	320	81	1440 ms
100	400	101	2250 ms

Table 4.3.: Verifying the sequence diagrams in Figure 4.10.

5. Related Work

This chapter provides an overview of the existing approaches that deal with the refinement of sequence diagrams. As mentioned in the introduction, current projects consist of several layers with consecutive levels of abstraction, so that according to Elaasar *et al.* [8] inter-level refinement assures *inter-model* consistency. Furthermore, intra-level refinement assures *intra-model* consistency that refers to the consistency between different diagrams within a single model. Another example for intra-model consistency is the consistency between sequence diagrams and state machine diagrams [21].

Three of the four concepts presented in the following section are basically very similar since they address the refinement at trace level, *i.e.*, the relationship between sequence diagrams is expressed in relationships between their positive and negative trace sets. The fourth approach is quite different because it uses a Petri net analysis to decide refinement in an operational way.

Several problems of working with sequence diagrams that exist besides the refinement concept are mentioned in Section 5.2. These problems are likely to cause errors, such as deadlocks, in an implementation, because the refinement concept cannot avoid these problems by itself.

5.1. Refinement

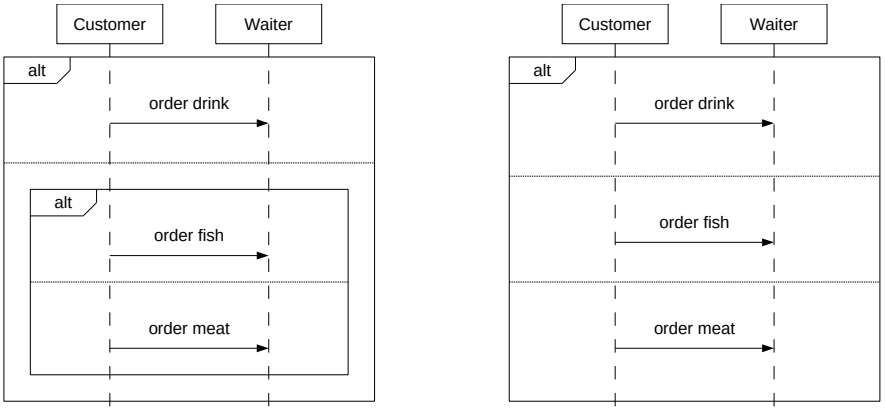
Cengarle *et al.* [5] have introduced a relation between processes and sequence diagrams. A process I is an implementation of a sequence diagram S if at least one linearization of I is a valid trace of S and if no linearization is a negative trace. In addition a sequence diagram T is a refinement of a sequence diagram S, if every implementation of T is also an implementation of S. This definition is very abstract and a practical implementation cannot be derived. Furthermore, the concept does not provide a possibility to decide if sequence diagrams refine each other without accessing an implementation, but a set of sequence diagrams should already be consistent before an implementation is actually generated. Expressing this concept in terms of traces reveals that every refinement step leads to a sequence diagram with less or equal positive traces, so that it differs from the refinement concept presented in Chapter 3.

Another refinement approach is presented by Störrle [37, 38]. Störrle refers directly to traces and characterizes refinement as *enrichment* and *restriction*. An enrichment step increases the number of positive traces, where restriction increases the number of negative traces. Furthermore, Störrle distinguishes three classes of design steps.

Detailing removes uncertainty, *i.e.*, contingent traces become either positive or negative. Since the previous positive and negative traces are maintained, a detailing step is only able to introduce additional alternative or optional behavior. *Adapting* allows to change the design completely and thus cannot be expressed or analyzed formally. Finally, *Realizing* refers to the final implementation step where every trace becomes either valid or invalid.

The STAIRS project by Haugen *et al.* [25, 26, 27] follows a similar approach. STAIRS distinguishes three basic kinds of incremental design steps. *Supplementing* corresponds to Störrle's detailing concept, *i.e.*, inconclusive traces become either positive or negative. *Narrowing* is similar to the approach by Cengarle and Knapp. Narrowing is used to remove underspecification, *i.e.*, positive traces become negative and negative traces remain negative, which can only be achieved by inserting additional *neg* fragments. Finally, *detailing* is characterized as *introducing a more detailed description without significantly altering the externally observable behavior* [25]. Detailing can be viewed as a combination of *lifeline substitution* and *black-box refinement* and is very similar to the approach presented in this thesis.

All three approaches mentioned so far have in common that they require to compute the positve and negative trace sets that are then checked for equivalence or inclusion to verify a refinement relationship. They do not provide other suggestions for efficient implemenations yet, so that practical applications become nearly impossible due to the factorial complexity of interleaving traces (*cf.* Sections 2.2 and 4.5). The possibility to create infinite traces with infinite loops poses another problem: In general, it is not even possible to characterize trace languages as ω-regular languages (*cf.* [7, 28]).



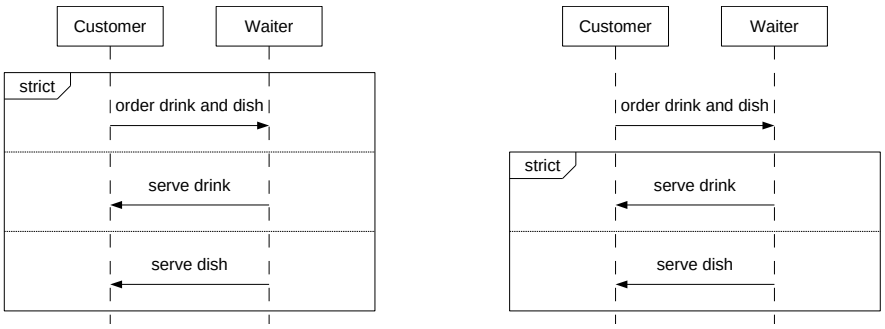
(a) Nested alt fragments.

(b) A single alt fragment with three alternatives.

Figure 5.1.: Equivalent traces although an alt fragment is missing.

The structural approach presented in this thesis can be computed more efficiently than the trace-based approaches presented above. However, a price to pay for this

efficiency is that the structural approach is more restrictive than the trace-based approaches. Figure 5.1(a) and 5.1(b) show two sequence diagrams with identical trace sets. In each diagram the customer orders either a drink, fish, or meat. Nevertheless, such a situation is rejected by two different structural rules (*cf. operand consistency* and *fragment completeness* in Chapter 3). Since both diagrams contain the same messages, they automatically have to contain the same combined fragments. Furthermore, the outer alt fragment is not consistent, because the messages order fish and order meat in Figure 5.1(a) have to appear in the same operand in Figure 5.1(b), but they do not.

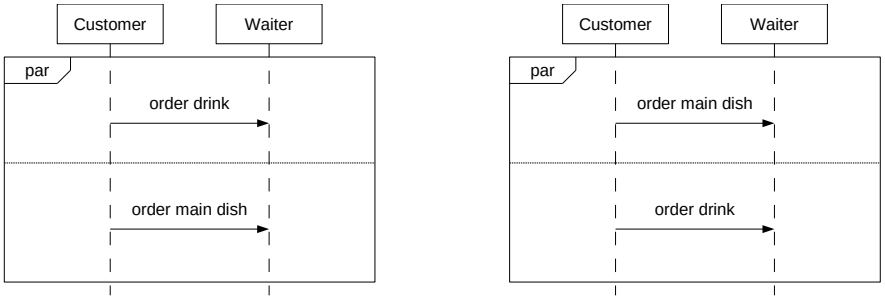


(a) A *strict* fragment covering all messages.

(b) One message is not covered by the *strict* fragment.

Figure 5.2.: Equivalent traces although the strict fragment covers different messages.

The trace sets of the sequence diagrams in Figure 5.2 are equal. The strict fragment, however, is not *operand consistent* so that such a situation is rejected too.



(a) Parallel messages

(b) Changed order of operands.

Figure 5.3.: The order of operands of a *par* fragment does not influence the traces.

At first this might seem too restrictive, but as sequence diagram are often larger than a single screen, such a situation becomes very misleading and confusing. Although there is no additional restriction on the sequence diagram layout yet, the strict approach at least helps to preserve a *mental map* [29] of a scenario. A minor

modification of the *operand consistency* rule (*cf.* Chapter 3) makes the concept even stricter by not allowing to change the order of operands within a par or alt fragment. Figure 5.3 shows two sequence diagrams with equal trace sets that could be rejected too.

Petri Nets

The approach presented by van der Aalst *et al.* [2, 41] is quite different from the previous three. Initially, van der Aalst *et al.* focused on the analysis of workflows providing a tool called *WOFLAN* [13, 12]. Workflows are represented as Petri nets [22] and are analyzed for a *soundness* property that assures a proper execution. Further investigations of van der Aalst *et al.* lead to an inheritance concept on workflows and Petri nets. They introduced the notions *protocol*, *projection*, and *life cycle* inheritance. These concepts can also be applied to UML behavior diagrams, such as state machine diagrams or sequence diagrams, if these diagrams are given as an equivalent Petri net [40]. The transformation from basic sequence diagrams to Petri nets is rather straightforward [10, 24]. The inheritance concept is similar to the refinement concept, since it allows to add instances and messages, if the primal structure is maintained. Replacing or removing instances, however, is not supported.

Although the Petri net inheritance is very interesting, it is decided by a *branching bisimulation* that leads to an exponential-size search space. Furthermore, at the moment the Petri net approach considers only basic sequence diagrams, *i.e.*, sequence diagrams without combined fragments, and it is still an open question whether the semantic of all operator kinds can be represented by equivalent Petri nets. The *loop* fragment, for example, allows to specify unbounded iterations that can lead to unbounded Petri nets. Unbounded Petri nets require places that contain an arbitrary number of tokens, which is not supported by WOFLAN.

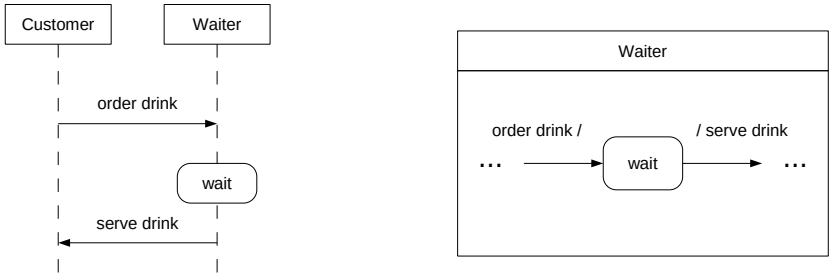
Nevertheless, Petri nets are very suitable candidates to discuss the semantics of sequence diagrams, since they provide an executable mechanism instead of the abstract trace notation. That is why the present thesis uses Petri nets to detect message order inconsistencies (*cf.* Chapter 4).

5.2. Soundness

Even if a set of sequence diagrams is consistent with respect to one of the refinement concepts some problems still remain, namely *unmotivated events*, *race conditions*, *process divergence*, and *non-local choices*. These problems might lead to deadlocks, buffer overflows, and to an unspecified or unwanted behavior in an implementation [34, 16]. Since these problems are induced by a wrong or too weak message order and since one important aspect of a refinement concept is to generally maintain the order of messages, these errors are preserved as well. Developers even need to pay attention that they do not accidentally use a refinement concept to propagate an error across several diagrams. Although there is little tool support, many approaches exist that help to detect and to avoid these problems.

Unmotivated Events

As mentioned before, a common approach is to use sequence diagrams as patterns to design interdependent and communicating state machines. Such state machines are usually event driven, *i.e.*, spontaneous actions are unwanted or even impossible, depending on the underlying framework. Besides presenting a refinement concept, Lischke [19] also provides a sequence diagram consistency checker that discovers so-called *unmotivated events*, *i.e.*, events within a sequence diagram that do not depend on a previously received trigger event. The send event of the message serve drink in Figure 5.4(a) is unmotivated, because there is no incoming message after the local state. Figure 5.4(b) shows the state machine fragment that corresponds to the unmotivated event. Such a state machine must change its state without an incoming trigger, which is not allowed by the ROOM framework [3].



(a) Sequence diagram with an *unmotivated event*.

(b) Incorrect state machine fragment caused by unmotivated events.

Figure 5.4.: Unmotivated events lead to incorrect state machines.

Race Conditions

The classic notion of race condition refers to the situation where a result unintentionally depends on the relative timing of at least two independent processes. A similar problem exists for sequence diagrams that arises from a possibly too weak message order. Alur *et al.* [33] define the problem as follows: *a race condition exists when two events appear in (visual) order, but can be shown to occur in the opposite order during an actual system execution.*

Figure 5.5 shows a simple sequence diagram that contains a race condition according to the definition just mentioned. The customer assumes that the drink and the meal arrive in exactly the given order, but since both waiters are completely independent a naïve implementation cannot assure this. Chen *et al.* [6] introduce a rather strict *race free* property that can be checked easily. Furthermore, they provide an algorithm that eliminates race conditions by inserting additional *silent* messages that strengthen the causal ordering.

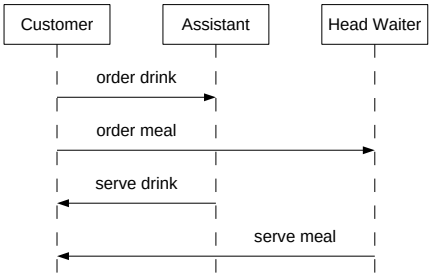


Figure 5.5.: Although the customer awaits the meal after the drink, a naïve implementation cannot assure this.

Process Divergence

The loop fragment, which was explained in Chapter 2, allows to specify an unbounded iterative behavior. A problem that may arise due to the asynchronous communication and the basic weak sequencing mechanism is *process divergence*. Ben-Abdallah *et al.* [14] define process divergence as *a system execution where one process sends a message an unbounded number of times ahead of the receiving process*. Figure 5.6 shows an unbounded loop where the customer has not to wait for the waiter, which allows him to flood the waiter with messages. This leads to buffer overflows or lost messages within the implementation. A syntactically verifiable property that impedes process divergence is *com-boundedness* [7]. Com-boundedness is also a sufficient condition for the regularity of the induced trace language.

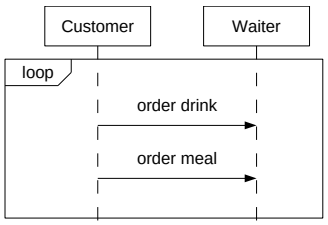


Figure 5.6.: With infinite loops the customer can flood the waiter with messages.

Non-local Choices

Another well-studied problem is the *non-local choice* problem [1, 20] of message sequence charts. This problem now also occurs in sequence diagrams since the alt fragment allows to specify alternative behavior. A non-local choice refers to a situation in which several instances can independently select different alternatives. As Muccini [16] shows, this may lead to *implied scenarios* [36, 34]. An implied scenario is characterized as an execution that is not covered by the specification. Figure 5.7 shows a sequence diagram with a non-local choice. Either waiter should serve the main dish. The diagram itself, however, does not explain how the waiters agree

on who serves it. They both may assume that it is their turn since there is no communication between them and the customer will finally receive two main dishes.

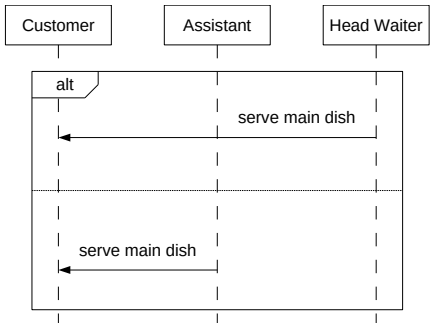


Figure 5.7.: Non-local choices might lead to unspecified behavior.

It is, however, still possible that an implementation does not contain an error even if a sequence diagram contains one of the problems mentioned above. In general, a sequence diagram does not provide a complete description of a scenario, since it focuses on the inter-object behavior. There is most likely additional information that cannot be expressed if only messages are available and which nevertheless has influence on the execution of an implementation. This information can be, for example, the execution time of an internal computation or even a property of the environment of the system.

In the last scenario, for example, there could be a board with the pending orders that is hanging in the kitchen. If each waiter obliterates the dish he wants to serve next, there is no possibility that a customer receives two dishes, unless he was very hungry and ordered two.

6. Conclusion and Further Work

Sequence diagrams are a convenient way to capture requirements. Due to their rather lean syntax and semantics and their focus on inter-object behavior, they are a suitable communication medium between different stakeholders, such as developers, testers, customers, and end-users. Furthermore, sequence diagrams can be used at nearly all development stages. During the implementation, for example, they serve as a pattern for the state machine design [21]. A good technique to obtain an implementation that fulfills the requirements is to incrementally refine the sequence diagrams from the requirement level until an adequate level of detail is reached. In doing so consistency management becomes mandatory [8].

This thesis defines a refinement concept that reflects the informal concept that is already applied in practice. A first approach, presented by Lischke [19] has introduced the concept for basic sequence diagrams without coregions or implicit parallelism. This approach is extended here to support parallelism, and moreover the new UML 2.0 combined fragments, which significantly increase the expressiveness. The concept presented in Chapter 3 separates structural from behavioral information.

Several strict rules prescribe which messages and combined fragments have to appear in which sequence diagrams without considering any semantic domain. If all these rather simple structural rules are fulfilled, it remains to verify whether the message order is consistent or not. As opposed to other approaches that deal with the refinement of sequence diagrams (*cf.* Chapter 5), this approach does not require to compute the positive and negative trace sets, which can be of factorial complexity. Instead, it suffices to reduce sequence diagrams to Petri nets that maintain the partial event orders. These Petri nets are used to validate different diagrams against each other. The construction and the evaluation of the Petri nets are efficient and the combination of a simulation and a computation of strongly connected components precisely identifies misordered messages.

The plugin that is described in Chapter 4 allows to evaluate the concept. The developers specify which sequence diagrams refine each other and can then run the consistency check. As the experimental results confirm, this approach scales very well to large diagrams. The sources of inconsistencies, if any exist, are located precisely. If, for example, some messages are wrongly ordered, these messages are highlighted in all diagrams in which they appear. It is, however, not possible to repair inconsistencies automatically, since in the case of a contradicting message order it is unclear which diagram is right.

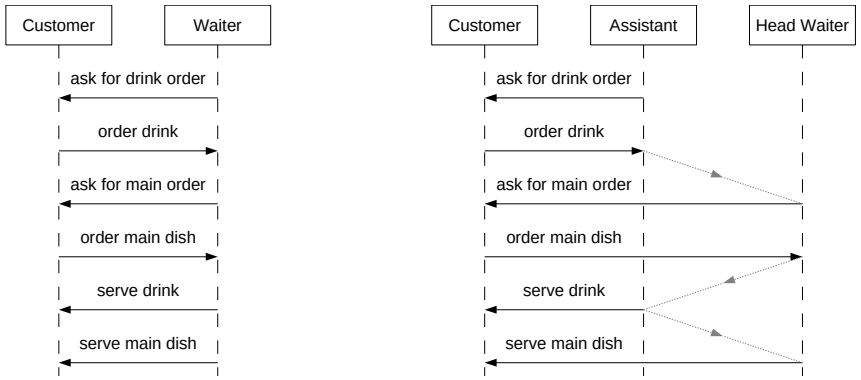
Although the results are already satisfying, since all objectives that were mentioned in the introduction are fulfilled, there are several potential extensions, which are outlined on the following.

6.1. Concept Extensions

The developed refinement concept focuses on instances, messages, and combined fragments. Sequence diagrams, however, can contain several other elements, such as *general orderings*, *local states*, or *local actions*.

General Orderings

General orderings are drawn as dotted lines between two independent event occurrences. An arrowhead in the middle of the line indicates the direction. General orderings strengthen the partial order so that they are basically used to introduce dependencies between event occurrences that otherwise would not be ordered at all. The refinement concept could use general orderings to maintain relations between event occurrences that were lost due to a refinement step.



(a) A customer ordering a drink and a main dish.

(b) Drinks and meals are served by different waiters.

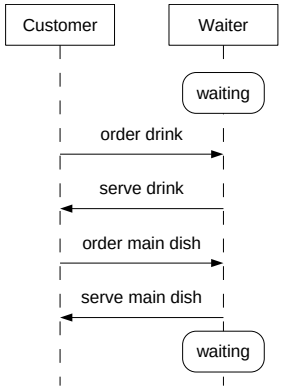
Figure 6.1.: General orderings replace lost information.

Figure 6.1(a) shows a scenario of a customer ordering a drink and a main dish from a waiter. This simple scenario is consistently refined in Figure 6.1(b) by replacing the single waiter with a head waiter and an assistant. Without general orderings the information that, for example, the message ask for drink order is sent before the message ask for main order would be lost.

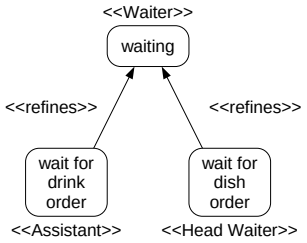
Local Actions / States

Local actions and local states are usually used to denote internal behavior and state changes, respectively. They should also be included in the refinement concept because their order is just as important as the order of messages. Especially local states are important for the state machine design (*cf.* [21]). There is no best way to relate actions between different abstraction levels, but for local states one approach could be to introduce a state hierarchy similar to the hierarchy of instances (*cf.* Chapter

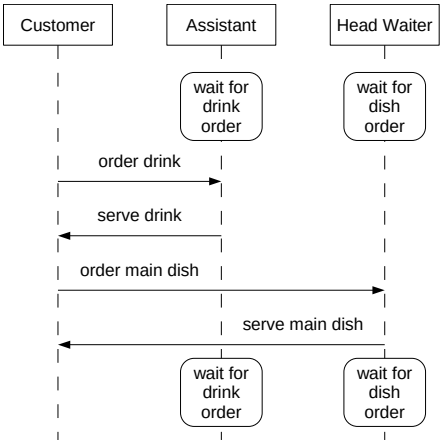
3). If an instance is replaced with its refining instances, the states of this instance must be replaced by the given states.



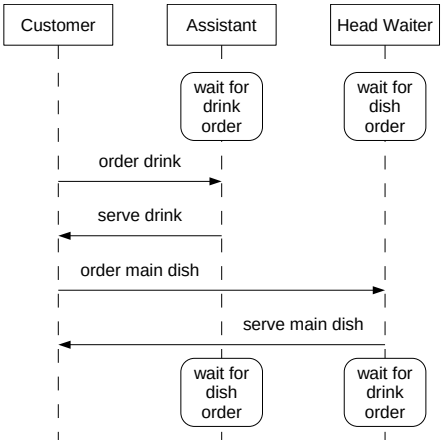
(a) Local states denote internal behavior.



(b) A state hierarchy restricts the state refinement.



(c) The state *waiting* is correct refined.



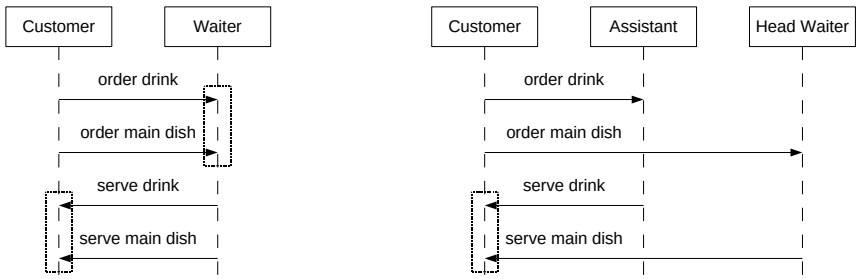
(d) A wrong refinement of the state *waiting*.

Figure 6.2.: A first approach to regard state refinement.

Figure 6.2(a) shows a sequence diagram with local states and Figure 6.2(b) shows the corresponding state hierarchy that restricts how local states are replaced in refinement steps. Every refinement step that replaces the instance waiter with the instances assistant and/or head waiter must also replace the local states according to the given hierarchy. The refining sequence diagram in Figure 6.2(c) is consistent with the state hierarchy, but Figure 6.2(d) is inconsistent because the states wait for drink order and wait for dish order are interchanged.

6.2. Additional Checks

Besides a refinement concept, there are several other approaches to improve modeling with sequence diagrams (*cf.* Chapter 5). It is obviously dangerous if a set of sequence diagrams is only consistent because they all contain the same error. Race conditions, for example, arise due to an event order that is too weak. Since the main principle of the refinement concept is to maintain the event order, a refining sequence diagram will contain a possible race condition too.



(a) ROOM does not allow message overtaking.

(b) Run-to-completion semantics and shared message queues avoid race conditions too.

Figure 6.3.: Not all race conditions according to Chen *et al.* arise in the ROOM framework.

Chen *et al.* [6] present a concept to detect and repair race conditions (*cf.* Chapter 5). This concept, however, may be too strict to be applied to all frameworks. According to Chen *et al.* Figure 6.3(a) contains two race conditions that are marked by dotted rectangles. If the underlying framework, however, does not allow *message overtaking*, *i.e.*, channels between different components preserve the order of messages, these race condition cannot occur. Another example is shown in Figure 6.3(b). In the ROOM framework [3], for example, it is possible that all instances share a single message queue. Assume that the customer initiates the scenario by sending the message order drink. Due to the run-to-completion semantics [3] the customer also sends the message order main dish before he has to wait for the message serve drink to arrive. Now the first message, which is order drink, is taken from the queue and forwarded to the assistant, which then appends the message serve drink at the end of the queue. Finally, the head waiter is able to receive his message and send the next message and the queue contains only messages for the customer exactly in the expected order. An extension of the race condition concept is needed that considers whether message overtaking is possible or not. Furthermore, the concept must support the possibility that several instances share a single message queue.

Other problems of sequence diagrams, such as non-local choices and process divergence mentioned in Chapter 5, can also be detected syntactically according to Ben-Abdallah *et al.* [14]. To detect non-local choices a sequence diagram is given as a so-called *message flow graph* that is similar to the Petri net representation used

in this thesis. For each choice node the set of events that might directly follow is computed. There is no non-local choice if all successors of a choice node belong to the same instance.

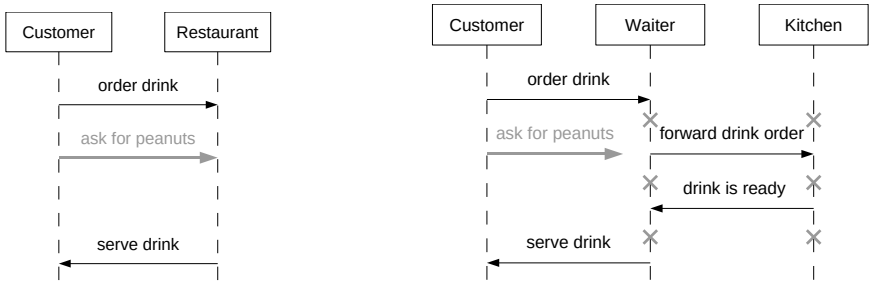
A so-called *coordination graph* is derived from the message flow graph to detect process divergence. The nodes of this graph are instances and directed arcs between two instances exist if one instance sends a message to the other. There is no process divergence if the transitive closure of the coordination graph is symmetric. Both algorithms run in linear time with the number of events within the sequence diagram and, with minor modifications, can be applied to the Petri net representation that already exists.

6.3. A Constructive Approach

Many modeling tools support consistency between different structural views. It is, for example, self-evident that a new class attribute immediately appears in all class diagrams that contain the corresponding class. These modeling tools usually also provide consistency between structural and behavioral views. It is possible to drag and drop classes from class diagrams into sequence diagrams. Messages are then simply created by selecting the corresponding operations. The modeling tools are able to automatically preserve consistency by separating the graphical representation from the actual model elements. This separation is similar to the Model-View-Controller (*MVC*) architectural design pattern [9] that enforces a strict separation from data (*model*), information display (*view*) and user input handling (*controller*). Views contain only references that point to the model elements so that there is no redundancy, which is one of the main reasons for inconsistency. Any change to the model elements automatically changes the information that is shown by the depending views. Such a mechanism would automatically eliminate several sources of inconsistencies during the refinement of sequence diagrams.

As shown in Chapter 4, the refinement concept is evaluated using an existing modeling tool. All information necessary is exported and checked by an external server application. Although this approach encourages reusability and maintainability, the development of a new tool that would provides a model view separation for sequence diagrams exhibits several major advantages. First of all, messages, instances, and combined fragments could be separated from their graphical representation and thus could be shared by all refining sequence diagrams. Any change to the label of a message or instance would automatically change the label in all dependent diagrams. If a message is deleted it would be deleted from the model and simultaneous from all views that contain this message. Other inconsistencies could be avoided by simply not allowing them to be made. For example, it could be forbidden to insert instances into a sequence diagram that are not consistent with the instance hierarchy. Furthermore, according to the structural rules in Chapter 3 it would always evident which elements are still missing. If a new message is inserted into a sequence diagram all sequence diagrams that describe the same scenario could be immediately notified

if they have to contain this message too. However, it is not possible to insert new messages or combined fragments automatically because in general there are several consistent possibilities. Figure 6.4(a) contains a basic diagram with a new message and Figure 6.4(b) shows the refinement diagram. It is obvious where to append the sending message end but there exist six consistent locations for the receiving end. If those situations occur, it would be convenient if the developer only has to choose one of the consistent possibilities.



(a) A new message was added to the basic diagram.

(b) Several possibilities to locate the receiving end of the new message exist in the refining diagram.

Figure 6.4.: Inserting new messages with a constructive approach.

The optimal solution would be to include the consistency between sequence diagrams and state machines [21]. The relationship between these diagrams, however, is even more intricate than the refinement of sequence diagrams. A modification of one sequence diagram may lead to inconsistencies between several state machines and vice versa. Since sequence diagrams and state machines focus on different domains, *i.e.*, inter-object and intra-object behavior, respectively, additional syntactical and semantical restrictions are most likely necessary and it still has to be evaluated whether this is practicable or not.

A. Bibliography

[1] Arjan J. Mooij and Nicolae Goga and Judi Romijn. Non-local Choice and Beyond: Intricacies of MSC Choice Nodes. In *Fundamental Approaches to Software Engineering (FASE '05)*, pages 273–288, 2005.

[2] Twan Basten and Wil M. P. van der Aalst. Inheritance of Behavior. In *Computing Science Report*, volume 17, Eindhoven University of Technology, Eindhoven, 1999.

[3] Bran Selic and Garth Gullekson and Paul T. Ward. *Real-Time Object-Oriented Modeling (ROOM)*. John Wiley & Sons, 1994.

[4] Branislav Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems. http://www-128.ibm.com/developerworks/rational/library/, 1998. IBM, White Paper.

[5] María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In *Proceedings of the 3rd Intl. Workshop on Critical Systems Development with UML (CSDUML '04)*, pages 85–99. Technische Universität München, 2004.

[6] Chien-An Chen, Sara Kalvala, and Jane Sinclair. Race Conditions in Message Sequence Charts. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS '05)*, pages 195–211, 2005.

[7] David Harel and P. S. Thiagarajan. Message Sequence Charts. *UML for Real: Design of Embedded Real-Time Systems*, pages 77–105, 2003. Kluwer Academic Publishers.

[8] M. Elaasar and L. Briand. An Overview of UML Consistency Management. Technical Report SCE-04-18, Carleton University, August 2004.

[9] Frank Buschmann and Regine Meunier and Hans Rohnert and Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[10] Peter Graubmann, Ekkart Rudolph, and Jens Grabowski. Towards a Petri Net Based Semantics Definition for Message Sequence Charts. In *Specification and Description Language (SDL) '93 - Using Objects*, October 1993.

[11] OMG Object Modelling Group. Unified Modelling Language: Superstructure. http://www.omg.org/technology/documents/formal/uml.htm, July 2004.

[12] H. M. W. (Eric) Verbeek and Twan Basten and Wil M. P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[13] H. M. W. (Eric) Verbeek and Wil M. P. van der Aalst. Woflan 2.0: A Petri-Net-Based Workflow Diagnosis Tool. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets (ICATPN) 2000*, pages 475–484, 2000.

[14] Hanêne Ben-Abdallah and Stefan Leue. Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts. In *Proceeding of 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1997.

[15] 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.

[16] Henry Muccini. Detecting Implied Scenarios Analyzing Non-local Branching Choices. In *Fundamental Approaches to Software Engineering (FASE '05)*, pages 372–386, 2003.

[17] International Telecommunication Union (ITU). ITU Homepage. http://www.itu.int/home/, 2006.

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

[19] Andrea Lischke. Consistency Check of Sequence Diagrams, 2005. Diploma Thesis, TU Cottbus.

[20] Loïc Hélouët. Some Pathological Message Sequence Charts, and how to detect them. In *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 348–364, London, UK, 2001. Springer-Verlag.

[21] Björn Lüdemann. Synthesis of human-readable Statecharts from Sequence Diagrams in the ROOM Environment, 2005. Diploma Thesis, Christian-Albrechts-Universität zu Kiel.

[22] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989.

[23] Object Managment Group (OMG). Unified Modeling Language (UML). http://www.uml.org/, 2006.

[24] 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.

[25] Øystein Haugen and Ketil Stølen. STAIRS - Steps to Analyze Interactions with Refinement Sematics. In *Proceedings of the Sixth International Conference on UML (UML'2003)*, pages 388–402, 2003.

[26] Øystein Haugen and Knut Eilif Husa and Ragnhild Kobro Runde and Ketil Stølen. Why Timed Sequence Diagrams Require Three-Event Semantics. In *Scenarios: Models, Transformations and Tools*, pages 1–25. Springer, 2003.

[27] Øystein Haugen and Knut Eilif Husa and Ragnhild Kobro Runde and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software and System Modeling*, 4(4):355–357, 2005.

[28] Doron Peled. Specification and Verification using Message Sequence Charts. *Electronic Notes in Theoretical Computer Science*, 65(7), 2002.

[29] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[30] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison Wesley, 2003.

[31] Philips Medical Systems (PMS). PMS Homepage. http://www.medical.philips.com/de/, 2006.

[32] 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.

[33] Rajeev Alur and Gerard J. Holzmann and Doron Peled. An Analyzer for Message Sequence Charts. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS),*, pages 35–48, 1996.

[34] Rajeev Alur and Kousha Etessami and Mihalis Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, July 2003.

[35] Rational Software. The Rational Unified Process: Best Practices for Software Development Teams. http://www-128.ibm.com/developerworks/rational/library/253.html, 1998.

[36] Sebastian Uchitel and Jeff Kramer and Jeff Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. In *9th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2001.

[37] Harald Störrle. Assert, Negate and Refinement in UML 2 Interactions. In *Proceedings of the International Workshop on Critical Systems Development with UML 2003*, pages 79–94, October 2003.

[38] Harald Störrle. Trace Semantics of Interactions in UML 2.0. Technical Report 09, University of Munich, February 2004.

[39] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[40] Wil M. P. van der Aalst. Inheritance of Dynamic Behavior in UML. In *Proceedings of the Second Workshop on Modelling of Objects, Components and Agents (MOCA 2002)*, volume 561, pages 105–120, August 2002.

[41] H. M. W. (Eric) Verbeek and Twan Basten. Deciding life-cycle inheritance on petri nets. In *Proceedings of the 24th International Conference on Application and Theory of Petri Nets (ICATPN) 2003*, pages 44–63, 2003.

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

B. XML

This appendix contains the full XML representation of the instance hierarchy that is shown in Figure 4.3 and the sequence diagram that is shown in Figure 4.4. Furthermore, this appendix contains the XML schema definitions that describe the structure of the information that is exchanged between the client and the server application.

instancehierarchy.xml

```
<!-- XML representation of Figure 3.3 -->

<instances>

    <!-- abstraction layer 0 -->

    <instance label="Customer" level="0"/>
    <instance label="Restaurant" level="0"/>

    <!-- abstraction layer 1 -->

    <instance label="Customer" level="1" master="Customer"/>
    <instance label="Waiter" level="1" master="Restaurant"/>
    <instance label="Kitchen" level="1" master="Restaurant"/>
    <instance label="Barkeeper" level="1" parent="Kitchen"/>
    <instance label="Cook" level="1" parent="Kitchen"/>

    <!-- abstraction layer 2 -->

    <instance label="Customer" level="2"/>
    <instance label="Head Waiter" level="2" master="Waiter"/>
    <instance label="Assistant" level="2" master="Waiter"/>
    <instance label="Kitchen" level="2" master="Kitchen"/>
    <instance label="Barkeeper" level="2" master="Barkeeper" parent="Kitchen"/>
    <instance label="Head Chef" level="2" master="Cook" parent="Kitchen"/>
    <instance label="Scullion" level="2" master="Cook" parent="Kitchen"/>

</instances>
```

sequencediagram.xml

```
<!-- XML representation of Figure 2.1(b) -->

<sd label="sequence diagram 2.1(b)" instances="Customer Waiter Kitchen">

    <fragment type="seq">

        <fragment type="par">

            <fragment type="basic">

                <fragment type="lifeline" label="Customer">
                    <fragment type="send" label="order drink"/>
                </fragment>

                <fragment type="lifeline" label="Waiter">
                    <fragment type="receive" label="order drink"/>
                </fragment>

            </fragment>

            <fragment type="basic">

                <fragment type="lifeline" label="Customer">
                    <fragment type="send" label="order main dish"/>
                </fragment>

                <fragment type="lifeline" label="Waiter">
                    <fragment type="receive" label="order main dish"/>
                </fragment>

            </fragment>

        </fragment>

        <fragment type="basic">

           <fragment type="lifeline" label="Customer">
               <fragment type="receive" label="serve drink"/>
               <fragment type="receive" label="serve main dish"/>
           </fragment>

           <fragment type="lifeline" label="Waiter">
               <fragment type="send" label="forward order"/>
               <fragment type="receive" label="prepared drink"/>
               <fragment type="send" label="serve drink"/>
               <fragment type="receive" label="prepared main dish"/>
               <fragment type="send" label="serve main dish"/>
           </fragment>

           <fragment type="lifeline" label="Kitchen">
               <fragment type="receive" label="forward order"/>
               <fragment type="send" label="prepared drink"/>
               <fragment type="send" label="prepared main dish"/>
           </fragment>

        </fragment>

        <fragment type="opt">

            <fragment type="basic">

                <fragment type="lifeline" label="Customer">
                    <fragment type="send" label="order coffee"/>
                    <fragment type="receive" label="serve coffee"/>
                </fragment>

                <fragment type="lifeline" label="Waiter">
                    <fragment type="receive" label="order coffee"/>
                    <fragment type="send" label="serve coffee"/>
                </fragment>

            </fragment>

        </fragment>

    </fragment>

</sd>
```

model.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="model" type="modelType"/>

<xs:complexType name="modelType">
    <xs:sequence>
        <xs:element name="instances" type="listOfInstances"/>
        <xs:element name="sds" type="listOfSDs"/>
    </xs:sequence>
    <xs:attribute name="checklevel" type="xs:integer" use="required"/>
</xs:complexType>

<xs:complexType name="listOfInstances">
    <xs:sequence>
        <xs:element name="instance" type="instanceType" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="instanceType">
    <xs:attribute name="label" type="xs:string" use="required"/>
    <xs:attribute name="parent" type="xs:string" use="optional"/>
    <xs:attribute name="master" type="xs:string" use="optional"/>
</xs:complexType>

<xs:complexType name="listOfSDs">
   <xs:sequence minOccurs="1" maxOccurs="unbounded">
       <xs:element name="sd" type="sdType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="sdType">
    <xs:sequence>
        <xs:element name="fragment" type="fragmentType"/>
    </xs:sequence>
    <xs:attribute name="label" type="xs:string" use="required"/>
    <xs:attribute name="instances" type="listOfLabels" use="required"/>
    <xs:attribute name="intralevel" type="listOfLabels" use="required"/>
    <xs:attribute name="interlevel" type="listOfLabels" use="required"/>
</xs:complexType>

<xs:complexType name="fragmentType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="fragment" type="fragmentType"/>
    </xs:sequence>
    <xs:attribute name="label" type="xs:string" use="optional"/>
    <xs:attribute name="type" type="fragmentSimpleType" use="required"/>
</xs:complexType>

<xs:simpleType name="fragmentSimpleType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="basic"/>
        <xs:enumeration value="par"/>
        <xs:enumeration value="seq"/>
        <xs:enumeration value="strict"/>
        <xs:enumeration value="loop"/>
        <xs:enumeration value="alt"/>
        <xs:enumeration value="opt"/>
        <xs:enumeration value="break"/>
        <xs:enumeration value="critical"/>
        <xs:enumeration value="consider"/>
        <xs:enumeration value="ignore"/>
        <xs:enumeration value="neg"/>
        <xs:enumeration value="assert"/>
        <xs:enumeration value="lifeline"/>
        <xs:enumeration value="send"/>
        <xs:enumeration value="receive"/>
        <xs:enumeration value="coregion"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="listOfLabels">
    <xs:list itemType="xs:string"/>
</xs:simpleType>

</xs:schema>
```