PEDECIBA Informática

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

UML 2.0 Interactions with OCL/RT Constraints

Daniel Calegari García

Trabajo de tesis para la obtención del grado de Magister en Informática de la Universidad de la República en el programa de Maestría del área Informática del Pedeciba

Supervisor:	Dr. Alberto Pardo Instituto de Computación Universidad de la República
Orientador:	Dra. María Victoria Cengarle Institut für Informatik Technische Universität München
	Dra. Nora Szasz Facultad de Ingeniería Universidad ORT Uruguay
Presentación:	29 de octubre de 2007

UML 2.0 Interactions
with OCL/RT Constraints

Daniel Calegari García

ISSN 0797–6410
Tesis de Maestría en Informática
Reporte Técnico RT 07-17
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, Octubre de 2007

Resumen

El uso de métodos formales en etapas tempranas del desarrollo de software contribuye a la confiabilidad y robustez de los sistemas a construir. En este contexto, el desarrollo de sistemas de tiempo real se beneficia de la construcción de modelos de comportamiento a los efectos de verificar la correcta satisfacción de sus restricciones de tiempo. El Unified Modeling Language (UML) es un lenguaje de especificación de software ampliamente utilizado por la industria y la academia. Sin embargo, su versión 2.0 carece de una semántica formal para el desarrollo de modelos cuya corrección pueda ser verificada. Además, su lenguaje de especificación de restricciones, el Object Constraint Language (OCL), posee limitaciones en cuanto a su uso en modelos de comportamiento de sistemas de tiempo real. Este trabajo se centra en la especificación del comportamiento inter-componente de sistemas de tiempo real. Dicho comportamiento es descrito utilizando el lenguaje de Interacciones de UML 2.0, extendido para la inclusión de restricciones de tiempo expresadas en el lenguaje OCL for Real Time (OCL/RT). El problema central abordado en este trabajo es la definición de una semántica formal para la fusión de ambos lenguajes. La semántica permite reconocer comportamiento válido e inválido de un sistema con restricciones de tiempo. Se realiza un análisis de las propiedades derivadas de la semántica, destinadas a la verificación formal. En particular, se exploran las nociones de refinamiento de interacciones y refinamiento de restricciones. Finalmente, se compara la propuesta con trabajos relacionados y se estudia su aplicación práctica a los efectos de analizar sus debilidades y beneficios. Este trabajo contribuye a la formalización de conceptos ampliamente utilizados en la práctica y, en consecuencia, a su inclusión en herramientas de modelado y razonamiento formal. Además, se aumenta la expresividad del lenguaje de Interacciones de UML 2.0 para soportar restricciones de tiempo complejas, no expresables hasta este momento.

Abstract

The use of formal methods at early stages of software development contributes to the reliability and robustness of the system to be constructed. In this context, real-time system development benefits from the construction of behavioral models in order to verify the correct satisfaction of time constraints. The Unified Modeling Language (UML) is a software specification language widely used by the industry and the academia. Nevertheless, its version 2.0 lacks a formal semantics for the development of provably-correct models. In addition, its constraint specification language, Object Constraint Language (OCL), has limitations for its use in behavioral models of real-time systems. This work concerns the inter-component behavioral specification of real-time systems. Such behavior is described using the UML 2.0 Interactions language extended for the inclusion of time constraints using the OCL for Real Time (OCL/RT) language. The main problem addressed in this work is the definition of a formal semantics for the fusion of both languages. The semantics allows recognizing valid and invalid behaviors of a system with time constraints. Intended for formal verification, an analysis of the properties derived from the semantics is also done. In particular, the notions of refinement of interactions and refinement of constraints are explored. Finally, the proposal is compared with related works and its practical application is studied in order to analyze its benefits and weaknesses. This work contributes to the formalization of concepts widely used in practice and, in consequence, to its inclusion in modeling and formal reasoning tools. Moreover, the expressivity of the UML 2.0 Interactions language is augmented in order to support complex real-time constraints, not expressable until this moment.

danksagung

me he preguntado qué tan cruel puede ser un agradecimiento para apresarlo con sus gramaticales cadenas y confinarlo a un trozo de papel. no lo es, pero aún así lo dejamos a merced del tiempo diluyéndose en olvido. solo trascendiendo los límites de la razón, solo donde la memoria claudica impotente al pie de nuestros sentimientos, solo allí es libre. por eso no agradezco grabando aquí tu nombre, sino que estimo tu solidaridad cuando lo que hago te es indiferente; envidio la pasión con la que profesas tus ideales e intento compartir tu espacio; aprecio la respetuosa distancia que creas cuando estás cerca y la íntima cercanía cuando te alejas; siento que olvido y distancia son lo mismo pero que nunca estuvimos más cerca; comparto tus horas como tu lo haces con las mías; sufro por ti ante la agobiante tarea de corregirme e intento crecer con tus comentarios; ignoro si nuestra amistad nació por causalidad o casualidad y celebro que no nos importe; temo de que la vida no me alcance para amarte a ti que abnegadamente me sostienes y me das más de lo que necesito; y finalmente, te dedico esta pequeña parte de nuestra historia con el más sincero y profundo silencio.

daniel, setiembre de 2007

Defer no time,
delays have dangerous ends.

William Shakespeare

Contents

1 Introduction **1**

2 Background and Motivation **5**

2.1 Model-Driven Engineering 5

2.2 UML 2.0 7

2.3 UML 2.0 Interactions 9

2.4 Real-Time and Reactive Systems 13

2.5 Object Constraint Language 14

2.6 OCL for Real Time 16

3 Syntax and Semantics of Constrained Interactions **21**

3.1 Abstract Syntax 21

3.2 Semantics 25

3.2.1 Semantic Domains 25

3.2.2 The Positive Fragment 28

3.2.3 The Negative Fragment 30

3.2.4 Summary of Constraints Satisfaction 32

3.3 Discussion: Basic vs Event 33

3.4 Related Work 34

3.4.1 Related Semantics 35

3.4.2 Other Semantics 38

4 Properties of Constrained Interactions **41**

4.1 Basic Properties 41

4.2 Constraint Properties 43

4.3 Semantical Concretization 44

5 Implementation and Refinement **49**

5.1 Definitions 49

5.2 Constraint Refinement 57

5.3 Related Work 59

6 In Practice **63**

6.1 OCL/RT in use 63

6.2 Examples 65

6.2.1 Basic Constraints 66

6.2.2 Satisfying System Runs 69

6.2.3 Hot and Cold Constraints 73

6.2.4 Refinement at Work 76

6.3 Discussions 78

6.3.1 Guards on Operands 78

6.3.2 State Invariants 80

6.4 Related Work 82

7 Conclusions and Future Work **85**

7.1 Summary and Conclusions 85

7.2 Future Work 88

References **91**

Chapter 1

Introduction

The subject of this thesis is the behavioral specification of real-time systems. In particular, we are concerned with the inclusion of real-time constraints in inter-component specifications using the Unified Modeling Language (UML,[OMG05b]).

UML enables the specification of a wide variety of aspects of a system from static structure and dynamic behavior, including non-structural constraints with the language OCL [OMG05a]. However, the last version of UML, 2.0, is far from the ideal specification language since the underlying theory is defined using natural language instead of being described in a formal way.

The inherent complexity of some kind of models, as behavioral models, increases in some specific contexts, for example with real-time, embedded and reactive systems [Dou04]. One of the main aspects involved in these kind of systems is timing, since a real-time system have to ensure not only the correct satisfaction of functional requirements but also the fulfillment of the time constraints. This implies that real-time constraint specification is an important aspect in behavioral specification of such systems.

Behavioral specifications describe how system elements interact over time. Inter-component behavioral specifications of real-time systems can be done using UML 2.0 Interactions. However, a real-time constraints specification language is needed. Since OCL is part of the UML, it is a natural candidate to be that language. The OCL has limitations for its use on the specification of real-time and reactive systems since it neither can feature time or signal handling constructs, nor is capable of expressing general liveness properties or performance aspects of systems conveniently. However, there is an extension named OCL for Real Time (OCL/RT) [CK02] for these purposes.

Although OCL/RT is adequate to real-time constraints specification, it must

be related with UML 2.0 interactions. Nowadays, there is no direct relationship between them and in this thesis we make them "work together". The main problem addressed in this sense is the statement of a formal semantics for the fusion of both languages. By achieving this, the expressivity of UML 2.0 Interactions can be augmented considering the meaning of time for behavioral specification of real-time systems. A common semantics will allow the incorporation of the resulting language in model-driven tools and its future relation with other languages, e.g., intra-component communications or design by contracts.

This intention is not capricious since software modeling and formal verification are strongly related in growing engineering approaches nowadays. Model-driven engineering [Ken02, Sch06] is a software engineering approach based on the high-level model-based specification of a system as the primary development activity. The construction of a software system is driven by model transformation techniques and automatic code generation. This approach allows a tighter control of the processes improving efficiency and making the resulting products more reliable. It also ensures software quality attributes by verification of the generated models at early development stages.

Related Work

UML 2.0 Interactions are not the only description language for inter-component communications. Their potential can be compared with High-Level Messages Sequence Charts (HMSC, [IT96]) and Live Sequence Charts (LSC, [DH01]) as studied in [Hau04, HT03], but they have differences. The choice of UML 2.0 addresses the intention of getting closer to an implicit standard.

There are some languages like UML/RT [Dou04] and ROOM [SGW94] which extend UML for real-time specifications. However, these works only take into account interactions with basic timing annotations, not complex constraints as we do.

There are also some works which define the semantics of UML 2.0 Interaction. Störrle [Stö03a, Stö03b, Stö04c], Haugen and Stølen [HKHS05, HS03], and Cengarle and Knapp [CK04b, CK05] define a denotational semantics intended to verify when an execution is valid or invalid, according to a given interaction. The works from Grosu and Smolka [GS05], Hammal [Ham06], and Cavarra and Küster-Filipe [CKF04] present an automata-theoretic semantics for UML 2.0 Interactions. None of them addresses the inclusion of complex real-time constraints, some of them include just basic time constraints. The semantics considered in this work will be the one proposed by

Cengarle and Knapp in [CK04b].

Finally, there are several works which analyze how to specify real-time constraints, from basic real-time constraints in High-Level Message Sequence Charts [LL00, BAL97], to the meaning of allocation and scheduling [KS01]. Some of them analyze different OCL extensions, mainly to specify temporal logic constraints in state machines [Fla03, FM04], and also in interactions [CKF05] to specify liveness properties. Only the work by Cavarra and Küster-Filipe [CKF05] analyzes an OCL extension to be used with interactions. In our work we consider OCL/RT, an extension of OCL for specifying real-time constraints proposed by Cengarle and Knapp [CK02].

Contributions of this Work

This work contributes to the clarification on the use of constraints in the inter-component behavioral specification of real-time systems. We consider that this is an authentic contribution since UML does not take into account interactions with time playing a main role. Our approach will go further on considering general constraints, not only real-time ones. It also contributes to the inclusion of concepts widely used in practice in modeling and formal reasoning tools.

The primary goal of this thesis is the definition of a formal semantics for the fusion of UML 2.0 Interactions and OCL/RT languages. This provides a solid formal background that tends to bridge the gap between different model-driven engineering needs.

We also study some properties derived from the semantics like refinement of interactions and refinement of constraints. Besides, a comparison with other approaches is done in order to analyze benefits and weaknesses of the proposal.

Finally, we are concerned with how these results work from a practical point of view. In particular, we analyze detailed consequences of this application and also study the practical applicability of this approach.

Structure of the Thesis

This thesis is structured as follows. Chapter 2 includes background information on model-driven engineering and real-time and reactive systems. Behavioral specifications with UML 2.0 Interactions and real-time constraints specification with OCL/RT are further motivated, pointing previous works strongly related to this thesis. Chapter 3 introduces a denotational semantics for an enhanced UML 2.0 Interactions language with OCL/RT constraints support. Chapter 4 presents some properties of interest derived from the semantics. In Chapter 5 we analyze the notions of implementation and refinement of an interaction, and refinement of constraints (both notions intended for formal verification). Chapter 6 shows the practical application of the concepts developed in previous sections and discusses the use of OCL/RT constraints in UML 2.0 Interactions. Finally, Chapter 7 presents a short summary with concluding remarks and an outline of future work.

Chapter 2

Background and Motivation

In this chapter we present the basic concepts of dynamic behavior specification of real-time systems, the general context in which the subject of our study is placed. We also introduce Model-Driven Engineering as a software engineering methodology capable of reducing development costs. It is based on the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. The use of UML 2.0 is motivated as a modeling language for this purpose. We also present main features of real-time and reactive systems, pointing out the importance of real-time constraints in behavioral specifications. Finally, the importance of a formal background and the main aspects of our study are motivated.

This chapter is structured as follows. In Section 2.1 we present an overview of Model-Driven Engineering. Section 2.2 introduces the use of UML 2.0 as a wide specification language. Section 2.3 is focused on UML 2.0 Interactions for behavioral specification and a specific formal semantics for it. In Section 2.4 an introduction to real-time and reactive systems is presented, and finally in Section 2.5 we motivate the use of the Object Constraint Language for real-time constraints specification.

2.1 Model-Driven Engineering

Nowadays software requirements lead to hard development decisions as they involve a wide variety of architectural and technical problems. In the last ten years a new development approach has been rising: the Model-Driven Engineering (MDE) [Ken02, Sch06]. MDE is a software engineering methodology based on models of the system to be constructed and the evolution of those models in order to perform an incremental development. The methodology involves the iterative transformation of abstract models into detailed ones

ending with the automatic code generation as the main construction technique. This methodology guides from business domains to software domains, enhancing industrial development productivity and software quality.

In this context, each model must be based on a formalism, to precisely define its syntax and its semantics. Syntax is made up of abstract and concrete syntax (which specifies the readable representation of the abstract notational elements). Semantics consists of static semantics (well-formedness rules that restrict the set of valid models) and dynamic semantics (assigning meaning to the abstract syntax). Figure 2.1 graphically presents the relationship among the system, the models and the formalisms. It is also very useful to count with a graphical and flexible modeling language in order to simplify the communication between different people involved in the project, and to be able to extend the language with domain-specific models.

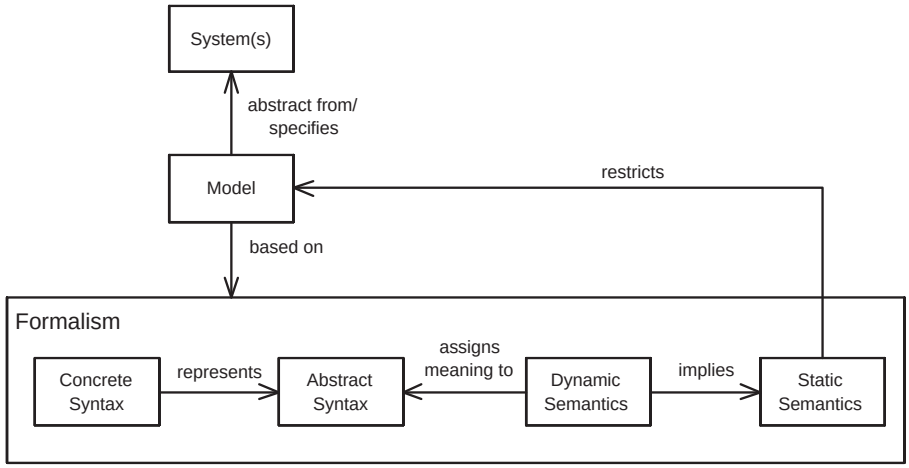


Figure 2.1: System, model and formalism

Model-Driven Architecture (MDA, [OMG03]) is a specific implementation of the Model-Driven Engineering (MDE) approach proposed by the Object Management Group (OMG). Another OMG acronym is Model-Driven Development (MDD) [MCF03, Sel03]. MDA introduces a set of layers that describe the levels of abstraction of the model. These range from models that are independent from any idea of computation to models that are tightly coupled to a specific implementation language. MDA also describes transformations driving business requirements models to implementations to provide a conceptual framework, as shown in Figure 2.2.

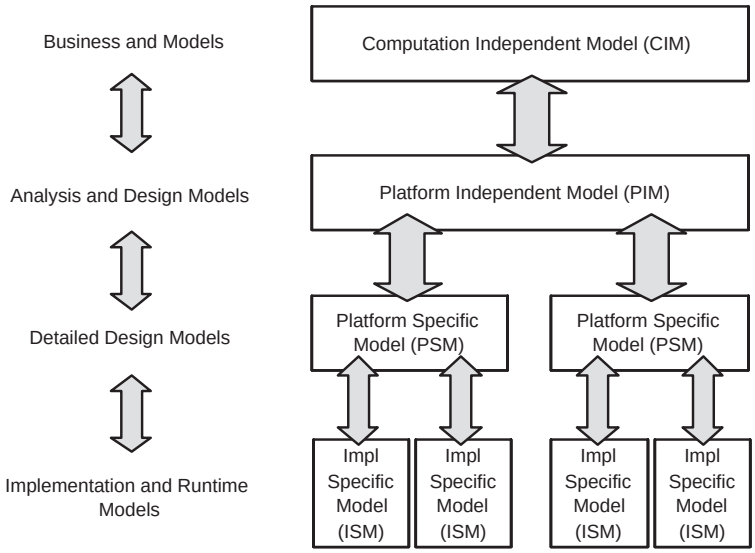


Figure 2.2: The layers and transformations of MDA

MDA uses the Unified Modeling Language (UML, [OMG05b]) as its specification language. This language has risen as the industry and academic standard for system specifications. Nowadays there is a wide variety of tools that partially support UML, varying from graphical modeling tools [Mic03] to development assistants, including partial code generators [Tig99, OMO06, KC04, IR05, Bor05]

2.2 UML 2.0

UML is a set of graphical languages within the object oriented paradigm (Figure 2.3). These languages allow specifying a wide variety of aspects of a system, from static structure to dynamic behavior. Structure can be described with static model elements such as classes, relationships, nodes, and components. Behavior describes how the elements within the structure interact over time. Besides, any UML language can be extended by its own extension mechanisms in order to define domain-specific models. UML also provides the Object Constraint Language (OCL, [OMG05a]) to specify non-structural constraints in a given object model with a textual notation.

UML 2.0 presents four different languages to specify dynamic behavior: Use

Cases, State Machines, Activities, and Interactions. Use Cases show the required usages of a system (functional requirements). State Machines are used to describe the behavior of a class, a subsystem or an entire system, focussing on how their state changes over time (intra-component behavior). Activities are used to describe "how things are done" and the work that is performed (system behavior). Finally, interactions are used to describe how a set of objects interact (by message passing) with each other in a specific scenario (inter-component behavior). The inherent complexity of behavioral models grows up in some specific contexts, for example with real-time, embedded and reactive systems. In this case not only UML models are necessary but also a careful specification of the real-time constraints. OCL can be used in conjunction with some dynamic specification language but there are no standard guidelines on how to do that.

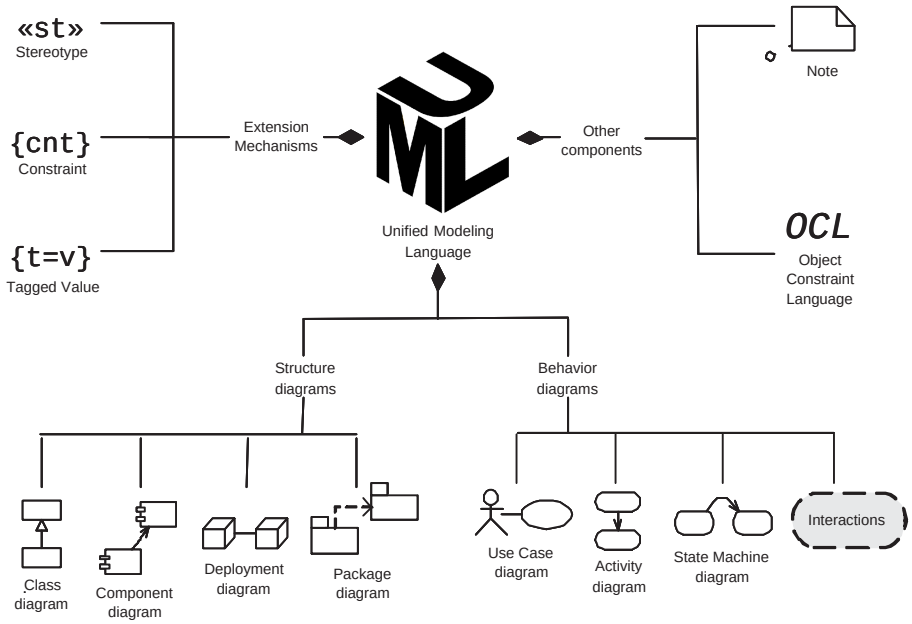


Figure 2.3: UML components

Although UML is simple and flexible, its last version 2.0 is far away from the ideal MDA specification language since the theory behind it is defined using natural language instead of being described in a formal way. The lack of a formal specification can lead to ambiguous, imprecise, contradictive and error-prone specifications. These drawbacks are recurrent from older versions of UML, promoting intensive academic work in order to solve them

[vdB02, BCR00, CK04a, CK05, GZK03, HKB04, HvdZ03, JEJ04, Jür02a, Jür02b, KFdB+04, KMR02, Krü00, Kwo00, RCA01, Rod00, Ste02, Stö04b, Stö04a, IMU05, Var02, Vig04]. Very often these kind of works are done without any relationship between them. This is a drawback in order to achieve a complete specification framework with unified semantics.

This work is focused on the inter-component behavior of real-time systems described by UML 2.0 Interactions. In particular, we are concerned about interactions with OCL real-time constraints and a unified formal semantics from the fusion of both languages.

2.3 UML 2.0 Interactions

UML 2.0 interactions [OMG05b] describe possible message exchanges between system instances, and provide means for specifying sequential, alternative, parallel, iterative and negative behavior (i.e. behavior forbidden in system implementations), among others.

A *message* is a communication between two instances which causes an operation to be invoked, a signal to be raised, or an instance to be created or destroyed. Every message is defined by two events representing the sending and the receiving of that message. For each system instance there is a *lifeline* representing the instance (typically object) execution over time, that is, messages sent or received with the corresponding object activation.

An interaction can be displayed in several different types of diagrams providing slightly different capabilities. Sequence Diagrams focus on message interchange (communication) among lifelines. Communication Diagrams show interactions where the communications among lifelines are decorated with descriptions of the passed messages and their sequencing. Interaction Overview Diagrams are a variant of Activity Diagrams that define interactions in a way that promote overview of the control flow. Finally, Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. Figure 2.4 shows examples of these diagrams.

The most common interaction diagram is the Sequence Diagram. A Sequence Diagram is enclosed by a rectangular frame. The name of the diagram is shown in a pentagram at the upper-left corner, prefixed by the keyword *sd*. The instances involved in the sequence are on the horizontal axis, each one is represented by an object rectangle with the object or class name. The rectangle along with the vertical dashed line represents the lifeline. Communications between instances are represented as horizontal message lines between the instance lifelines. Sequence Diagrams are read from top to bottom to view the exchange of messages taking place as time

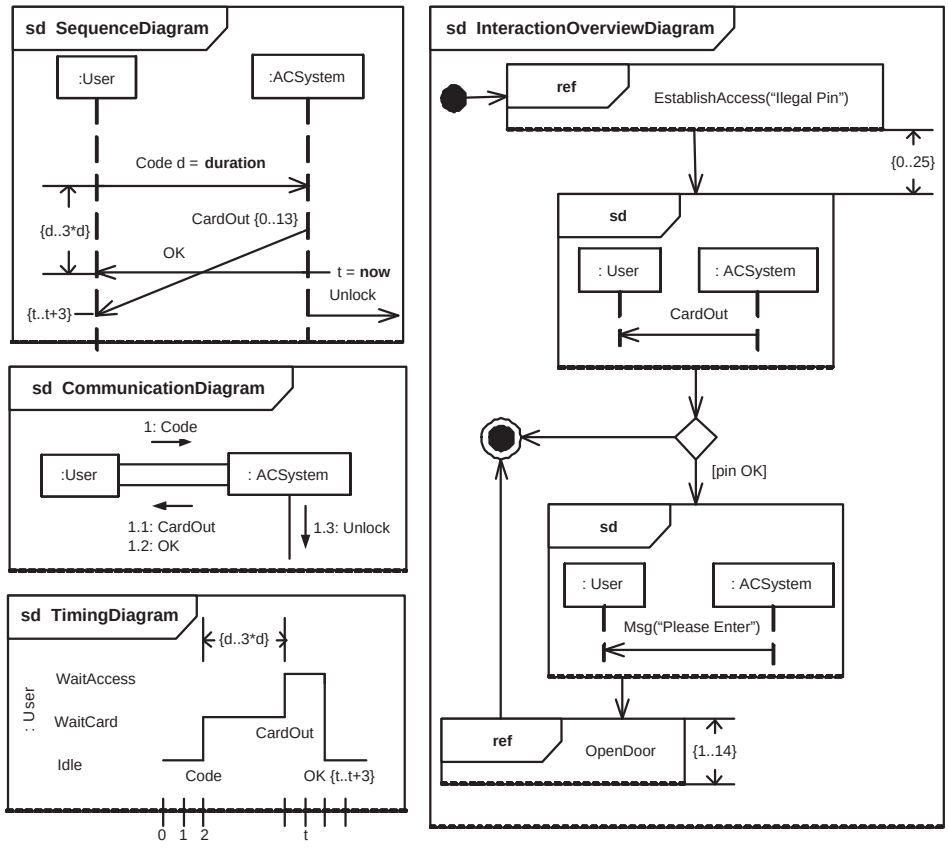
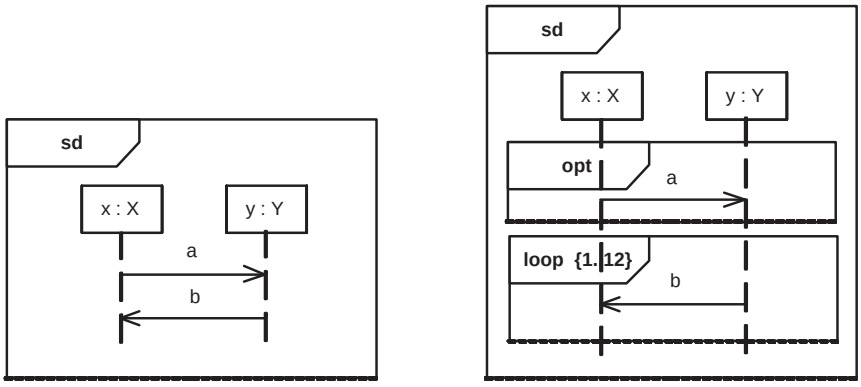


Figure 2.4: UML interaction diagrams

passes. Figure 2.5(a) shows an example of an interaction between two objects x and y that interchange messages a and b.

Each interaction may contain sub-interactions called *interaction fragments*, which can be structured and combined using *interaction operators*. The resulting *combined fragments* enclose messages within a rectangular frame with the name of the operator shown in a pentagram at the upper-left corner. These define special behavior like sequential, parallel, and iterative composition of interactions, alternative, optional and invalid interactions, and interactions considering and ignoring a set of messages, among others. An example is shown in Figure 2.5(b) where message a is optional and message b is sent up to 12 times.



(a) Basic interaction (b) Interaction with combined fragments

Figure 2.5: Basic interactions and combined fragments

Usual operators are:

`alt` represents a choice of behavior constrained by a guard expression

`par` represents a parallel merge between the behaviors of the operands

`seq` represents a weak sequencing between the behavior of the operands

`strict` represents a strict sequencing between the behaviors of the operands

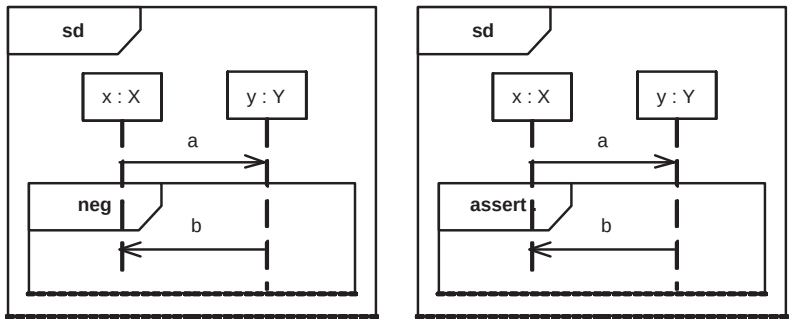
`loop` represents an iterative behavior

`ignore` represents a behavior where some messages are ignored

Some other operators, such as `break` and `critical`, are left apart. Weak sequencing between operands implies that the ordering of event occurrences within each of the operands is maintained whereas event occurrences on different lifelines from different operands may come in any order. It also implies that event occurrences on the same lifeline from different operands are ordered in such a way that an event occurrence of the first operand comes before any other event of the second operand.

An interaction can also specify negative behavior (behavior forbidden in system implementations). Negative behavior is introduced by two specific unary operators: `neg` and `assert`. While `neg` states that the sequence of messages within the fragment are supposed to be invalid, `assert` indicates that the sequence of messages within the fragment is an assertion (the only

valid execution of the system). The interaction shown in Figure 2.6(a) specifies that message b cannot be sent after message a, while the one of Figure 2.6(b) specifies that message b is the only possible message after message a (any other behavior is prohibited).



(a) Negative fragment (b) Assertion

Figure 2.6: Interactions with negative behavior

Syntax & Semantics

The UML 2.0 specification introduces some guidelines for the formal understanding of interactions, which is mandatory for the use of the language as a modeling tool. However, the specification is rather vague and obscure in some aspects since it is specified in natural language. As an example, it is not clear whether it should be possible to have several guarded expressions in an alternative specification (`alt` operator) evaluating to true.

There are some works which define the semantics of UML 2.0 Interactions: Störrle [Stö03a, Stö03b, Stö04c], Haugen and Stølen [HS03], Cengarle and Knapp [CK04b, CK05], and Cavarra and Küster-Filipe [CKF05]. None of these works considers complex constraints to be verified over an interaction. In this work we consider the semantics proposed by Cengarle and Knapp in [CK04b], which will be introduced in this section. The semantics decides when a *trace* (sequence of messages) is positive (a valid execution) or negative (an invalid execution) for some interaction. Since our work will be strongly based in this semantics, we will briefly introduce it.

The abstract syntax of the language is specified by a context-free grammar which does not consider all the features introduced by the UML 2.0 specification. Some of them, such as the operators `break` and `critical`, are left for future study.

Two satisfaction relations between a trace and an interaction are defined: one that does not contain occurrences of the negation and assertion operators (positive fragment), and the other that contains them (negative fragment). The positive (negative) satisfaction relation between traces and interactions, denoted by $t \models_p (\models_n) S$ and read t positively (negatively) satisfies S, where t is a trace and S an interaction of the positive (negative) fragment, is inductively defined on the structure of S. The relations, plainly worded, state that a trace satisfies the interaction if it completely runs trough a possible path of the interaction.

Semantically, an interaction specifies valid (also called *positive*) traces of event occurrences and invalid ones (also called *negative*). A trace t is positive for some interaction S if it positively satisfies the interaction ($t \models_p S$). A trace t is negative for some interaction S if it negatively satisfies the interaction ($t \models_n S$). Some traces cannot be neither positive nor negative for some interaction, those traces are called *inconclusive* for that interaction. Moreover, in some cases, a trace can be both positive and negative, in this case the interaction is called *overspecified*.

2.4 Real-Time and Reactive Systems

Real-time and reactive systems are special software systems concerned with performance, scheduling and timing. Due to these aspects, real-time systems development is one of the most complex software engineering problems [SGW94, Dou04, SR98, MCM04].

One of the main aspects involved in these systems is timing, since a real-time system must not only ensure the correct satisfaction of functional requirements but also the time when the results are produced. A real-time system can be classified as either hard or soft. The distinction is somewhat fuzzy. On one end there are non-real-time systems, where no important deadlines exist (meaning all deadlines can be missed), while on the other end there are hard real-time systems, where no deadlines can be missed.

Somewhere in the real-time spectrum, real-time constraints start being the critical aspect in the system behavioral specification. There are tools [IL05, Sof05b, IR04, Sof05a] which allow the specification of some real-time system aspects in UML 2.0, generate code and test the application. But some aspects are left apart, e.g., integration with tools for testing safety and liveness properties and specification of complex real-time constraints.

Inter-component behavioral specification of real-time systems can be done by using UML 2.0 Interactions. However, a constraint specification language is needed. Since OCL is part of the UML, it is a natural candidate.

2.5 Object Constraint Language

The Object Constraint Language (OCL, [OMG05a]) is a semi-formal language for the axiomatic specification of consistent system states using invariants and the transformation of system states by means of pre-/post-conditions for operations. OCL also allows the definition of variables and operations that can be used within expressions and action clauses (Kleppe y Warmer [KW00]), indicating that actions will be taken when some condition becomes satisfied.

We define the main elements of OCL with the following example (extracted from [CK02]). In Figure 2.7 there is a class diagram representing the static structure of a (over-simplified) model of several automatic teller machines (ATMs) connected to a single bank, showing an association with according multiplicities between the classes *ATM* and *Bank*. An *ATM* has a depot attribute, holding the current amount of money it can spend; the identification number of the card in use, with *cardId* set to, zero if it holds no card; and a state indicating whether an error has occurred during processing. An *ATM* may spend an amount of money when the operation *spend* is called on it. The *Bank* has two operations: *credit* withdraws an amount of money from the card holder's account if this amount is covered; *requestRefill* registers *ATM*s whose depots are running low.

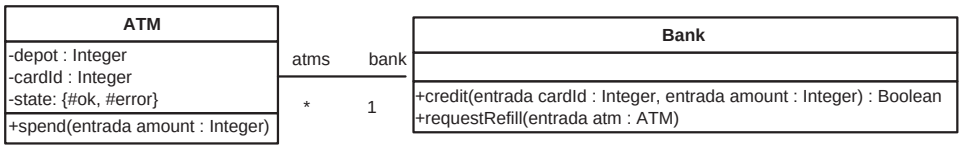


Figure 2.7: UML class diagram for ATMs

• **Invariants** An invariant is a boolean expression that must be true for every instance of some type at any time. In the example, if there is not an error then the ATM has enough money to spend.

```
context ATM
inv: (self.state = #ok) implies (self.depot >= 100)
```

OCL uses the dot-notation for navigation to attributes and through associations (as well as for operation calls). The OCL expression `self` denotes

the instance on which the constraint is evaluated, and may be omitted if the navigation reference remains unambiguous.

• **Pre- and Post- Conditions** A pre-condition is a condition that must always be true just prior to the execution of an operation in order to get the expected effect. Every time *spend* is invoked, the ATM must not be on an error state, it must have a card, the amount of money to be withdrawn must be positive and the depot must cover the withdrawal.

```
context ATM::spend(amount : Integer)
pre:  (state = #ok) and (cardId <> 0) and
      (amount > 0) and (depot >= amount+100)
```

A post-condition is a condition that must always be true just after the execution of an operation. After *spend* has been executed, the right amount of money must have been spent or some error has occurred.

```
context ATM::spend(amount : Integer)
post: (depot = depot@pre-amount) or (state = #error)
```

Post-conditions make use of the `@pre` operator that yields the value of an expression at pre-condition time.

• **Definitions** A definition is the specification of a helper variable or operation to be reused in multiple OCL expressions. The bank may define an operation calculating the sum of the depots in its ATMs.

```
context Bank
def: depotSum() : Integer =
        self.atms->iterate(i : ATM;
                           sum : Integer = 0 | sum+i.depot)
```

The `iterate` operator iterates through a given collection and accumulates the results of evaluating an expression with an iterator variable bound to the current element and an accumulator variable bound to the previous result. The execution of an operation is made by the arrow notation $->$.

• **Actions** An *action clause for classes* requires that whenever a condition becomes satisfied, an operation has to be called. In the example, if the depot is less than 1000, the bank has to refill the ATM.

```
context ATM
action: depot < 1000 ==> bank.requestRefill(self)
```

An *action clause for operations* specifies that, when some condition is specified at post-condition time, certain other operation calls must have happened while executing the operation. In the example, during the execution of *spend* the bank must verify the credit in the client's card.

```
context ATM::spend(amount : Integer)
action: true ==> bank.credit(cardId, amount)
```

The OCL has limitations for its use on the specification of real-time and reactive systems as the language does not feature time or signal handling constructs, nor is capable of expressing general liveness properties or performance aspects of systems conveniently. However there are some works that analyze different OCL extensions varying from the specification of temporal logic constraints in state machines [Fla03, FM04] to real-time constraints in every UML model [CK02]. There are also some works that explore how to specify real-time constraints in interactions [HKHS05, HS03, Kna99], but these works do not support complex constraints specification or they do not consider it within interactions semantics.

2.6 OCL for Real Time

OCL/RT (OCL for Real Time, [CK02]) is an extension of OCL 1.4 for real-time and reactive systems constraints specification. The language is based on the notion of traces of events with timestamps which allows specifying the temporal behavior of a system. The constraints are evaluated over sequences of system states (system execution) instead of just on a given state as OCL does. In this context, new operators are defined extending the expressiveness of the original language.

In order to consider timed-events the meta-class `Event` (UML 1.5) was extended, as shown in Figure 2.8. Each event shows the time at which it occurred by a link to the new primitive datatype `Time` that represents the global system time. It is assumed that time comes with a total ordering relation $\leq$ for comparing time values, an associative and commutative binary operation $+$ for adding time values, and a class attribute `now` that always yields the current system time. Events are associated to instances (of classifiers), such that an instance is linked to all its current events. Also, an event must carry a list of actual parameters.

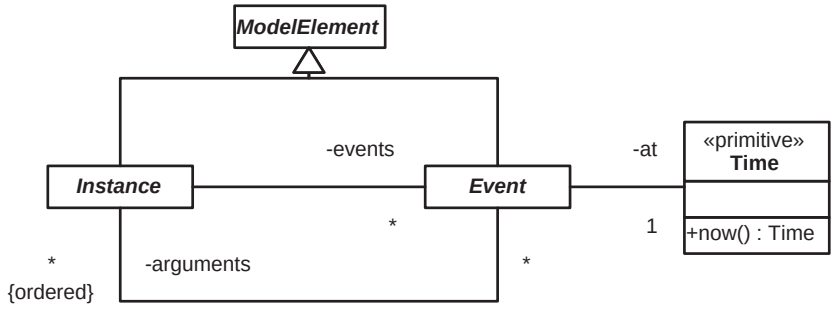


Figure 2.8: Event meta-class

For the definition of OCL/RT constraints, that are evaluated over a sequence of system states, a new clause is introduced

```
context C
constr: c
```

where `C` is a classifier and `c` an OCL/RT constraint.

OCL terms are captured by a the context-free grammar in [CK04a]. This grammar is extended with particular OCL/RT features as shown in Figure 2.9. The modality `always c` satisfies the constraint `c` over a system-run when the constraint evaluates to true in every system state. OCL/RT expressions also include a satisfaction operator $@\eta$ that yields the value of an expression when the event η occurs (the expression *Expr* in *Event* must have type `Event` to be well-formed).

The original OCL operators (inv, pre, post and action) are reinterpreted in OCL/RT using the operator `constr`, as is the case of invariant (*inv*) operator showed next.

```
-- The invariant must be checked whenever an operation
-- is called (StartEvent) on the instance from outside
context C
def: publicCalls() : Event
       events->select(e |
         e.isTypeOf(StartEvent) and
         e.call.operation.visibility = #public)
constr: always (publicCalls()->forAll(s | inv@s))
```

Term ::= *Spec* | *Constr*
Spec ::= ...
Def ::= ...
Constr ::= `context` *Type* `constr:` *Expr*
.....
Expr ::= ...
| `always` *Expr*
| *Expr* @ *Event*
Event ::= *Expr*

Figure 2.9: Abstract syntax of OCL/RT

In order to do such reinterpretation, it is necessary to extend the UML events hierarchy, e.g., with call, start and termination events for the execution of an operation as shown in Figure 2.10. Depending on what is necessary to model, the event hierarchy must be extended, as in the case of modeling signals or attribute value changes.

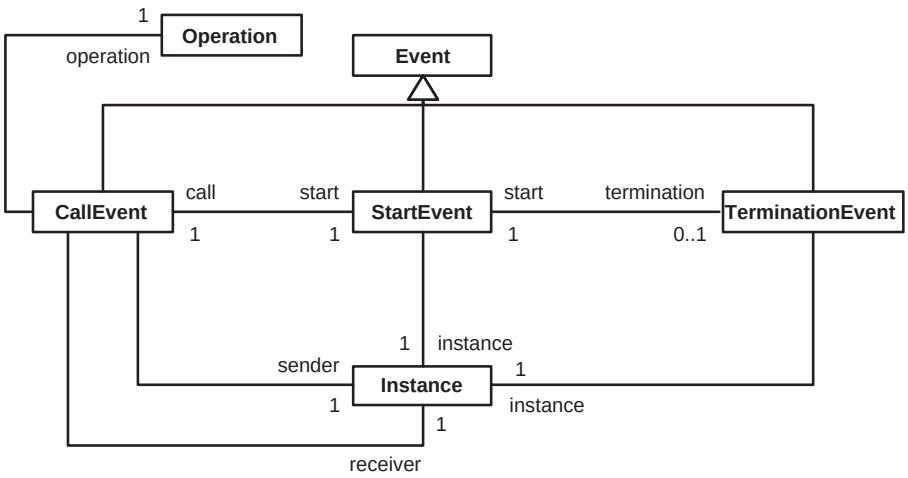


Figure 2.10: Events for operations

With this extension, deadlines for operations and reaction to signals and timeouts, among others, can be specified. As an example, if the operation *spend* (shown in Figure 2.7) must finish before T units of time (deadline), the post-condition of *spend* is rewritten as follows:

```
context ATM::spend(amount : Integer)}
pre:  ...
post: (depot = depot@pre-amount and
       Time.now <= Time.now@pre+T) or (state = #error)
```

Although OCL/RT is adequate to real-time constraints specification, it must be related with UML 2.0 interactions. Nowadays, there is no direct relationship between both languages. A common semantics allows the incorporation of the resulting language in model-driven tools and the future relation of it with other languages, e.g., for intra-component communications or design by contract.

Semantics

System states are formalized by *dynamic bases*. A *dynamic basis* comprises an implementation of the predefined OCL types and their operations, as well as the set of current instances of classes together with their attribute valuations, connections to other instances, and implementations of operations. Moreover a *dynamic basis* can be extended by implementations of auxiliary, user-defined operations.

Invariants, pre-/post-conditions, and action clauses restrict system-runs. A *system-run* ρ is a finite or infinite sequence of pairs of dynamic bases and finite sets of events. The dynamic basis ω_0 defines the initial system state; ω_n is transformed into ω_{n+1} by a single system step where the events in H_i occur.

Cengarle and Knapp define an operational semantics [CK02] that derives judgments of the form

$$(\rho,i);\gamma \vdash c \downarrow v$$

where ρ is a system-run, i an index in the system-run, γ a variable environment, c an OCL/RT constraint, and v a value. Such a judgement conveys the fact that c evaluates to v at the ith system state in the system-run ρ using the variable environment γ.

The operational rules comprise all rules of OCL as defined in [CK01, CK04b] and also new rules for the new language features as the instance attribute `events`, the @η operator, the modality `always` and the evaluation of `constr`.

Summary

In this chapter background information on dynamic behavior specification of real-time systems was given. The language of UML 2.0 Interactions is a good starting point to specify inter-component communications, as well as OCL/RT is a good option in order to specify temporal constraints. In addition, both languages have a formal semantics, but there are not related.

The motivation of this work is the need of a language to specify inter-component behavior of real-time systems with constraints support. We achieve this by defining the language of UML 2.0 Interactions with OCL/RT constraints. This means, the fusion of both languages by the definition of common semantics from those already defined by Cengarle and Knapp in [CK02, CK04b]. A formal semantics allows the incorporation of the language in model-driven tools and the future relation of it with other languages, e.g., for intra-component communications or design by contract.

Chapter 3

Syntax and Semantics of Constrained Interactions

We extend the language of UML 2.0 Interactions with OCL/RT constraints which must be satisfied on a system-run in order to consider the system-run a valid execution of the system being modeled. This increases the potential of the interactions for inter-component behavioral specifications. However, the main problem addressed is the statement of a formal semantics for the language, since without it the meaning of a specification could be ambiguous, imprecise, contradictive and in consequence error-prone. Our proposal is based on the definition of a trace-based formal semantics to recognize valid and invalid system-runs for some given interaction with constraints satisfaction.

This chapter is structured as follows. Section 3.1 presents the abstract syntax of the UML 2.0 Interactions language extended with OCL/RT constraints. Section 3.2 introduces the trace-based formal semantics for the resulting language. Section 3.3 presents a discussion about the possibility of simplifying the semantics already defined. Finally, in Section 3.4 a comparison with other approaches is given and related work is considered.

3.1 Abstract Syntax

The abstract syntax of the fragment of the language of UML 2.0 Interactions already introduced in Section 2.3 is extended in order to consider OCL/RT time constraints as shown in Figure 3.1.

The production rule `constraint`(*Term*,*Interaction*) considers an OCL/RT term and the interaction over which the term is evaluated. For our purposes,

we must ensure that every *Term* must be a boolean expression since other kind of expressions are not constraints.

$$
\begin{aligned}
\mathit{Interaction} ::= {} & \mathit{Basic} \\
| {} & \mathit{CombinedFragment} \\
\mathit{CombinedFragment} ::= {} & \texttt{strict}(\mathit{Interaction},\mathit{Interaction}) \\
| {} & \texttt{seq}(\mathit{Interaction},\mathit{Interaction}) \\
| {} & \texttt{par}(\mathit{Interaction},\mathit{Interaction}) \\
| {} & \texttt{loop}(\mathit{Nat},(\mathit{Nat}|\infty),\mathit{Interaction}) \\
| {} & \texttt{ignore}(\mathit{Messages},\mathit{Interaction}) \\
| {} & \texttt{alt}(\mathit{Interaction},\mathit{Interaction}) \\
| {} & \texttt{neg}(\mathit{Interaction}) \\
| {} & \texttt{assert}(\mathit{Interaction}) \\
| {} & \texttt{constraint}(\mathit{Term},\ \mathit{Interaction})
\end{aligned}
$$

Figure 3.1: Abstract syntax of interactions with OCL/RT constraints

As has been described in Section 2.6, OCL/RT is based on a modification and extension of the original UML 1.5 abstract meta-class `Event`. We adapt that extension to the UML 2.0 metamodel since the classes `Instance` and `ModelElement` do not longer exist.

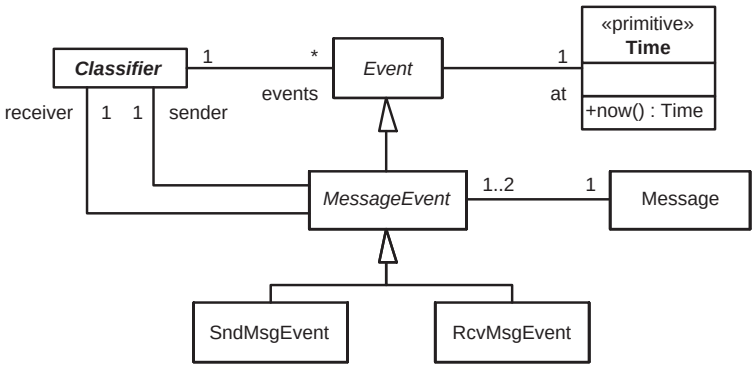


Figure 3.2: Event model for messages

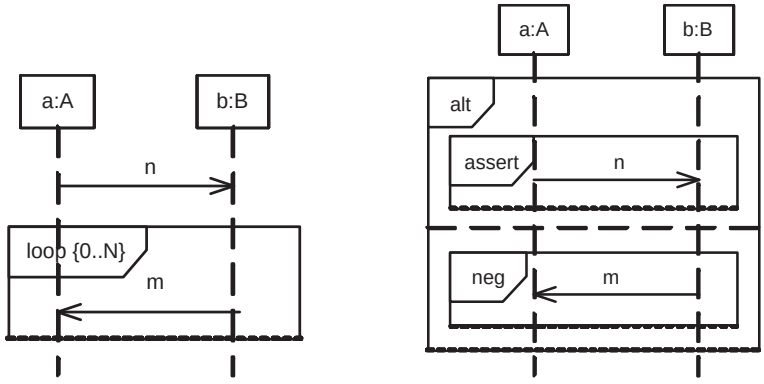
Each event shows the time at which it occurred by a link to the primitive data type `Time` that represents the global system time, as in the original definition. It is assumed that `Time` comes with a total ordering relation $\leq$ for comparing time values, an associative and commutative binary operation

+ for adding time values, and a class attribute `now` that always yields the current system time.

In order to model a system based on message passing through instances we extend the hierarchy as shown in Figure 3.2. Every message is composed of at most two events (partial messages) representing the sending of that message by an instance (of a classifier) and the reception of it by another instance (possibly the same). An instance of a classifier is linked to all its current events, message events among them. There is also one well-formedness rule on system-runs: send and receive events of the same message must occur in this order.

Example (Abstract Syntax)

During this chapter we will consider two simple examples as a proof of concepts. Figure 3.3 shows two simple interactions.



(a) Interaction S_1 (b) Interaction S_2

Figure 3.3: Simple interactions

In the interaction S_1 an instance a of A sends the message n to an instance b of B and b responds sending up to N times the message m. There is also a constraint (φ_1), not shown graphically, involving events which says that every message m must be sent within 10 time units since the reception of n, we can express it as follws.

```
context B
def: sndM : Set(Event) =
       events->select(e | e.oclIsTypeOf(SndMsgEvent)
                          and e.message.name = "m")
def: rcvN : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent)
                          and e.message.name = "n")->any()
constr: always(sndM->forAll(e | e.at < rcvN.at + 10))
```

In order to construct the abstract syntax we need to represent message n as a basic interaction B_1 and message m as a basic interaction B_2. Joining the interaction and the constraint we get the following abstract syntax.

$$\texttt{constraint}(\varphi_1, \texttt{seq}(B_1, \texttt{loop}(0,N,B_2)))$$

Notice that there exists an implicit weak sequence between B_1 and the loop. There is also an implicit `constraint` fragment which must be placed before the weak sequence since it involves both fragments of the sequence.

In the interaction S_2 there are two alternatives: sending the message n to instance b or not receiving the message m from instance b. The operator `neg` introduces negative behavior. There is also a constraint (φ_2), not shown graphically, involving events and the state of the system which restricts the sending of message n only if the boolean attribute ok of instance a is true. This constraint is expressed as follows.

```
context A
def: sndN : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent)
                          and e.message.name = "n")->any()
constr: always((self.ok)@sndN)
```

In this case the abstract syntax is expressed as follows, where the constraint φ_2 only restricts the basic interaction B_1 (basic interactions B_1 and B_2 are the same as in the first example).

$$\texttt{alt}(\texttt{constraint}(\varphi_2, \texttt{assert}(B_1)), \texttt{neg}(B_2))$$

3.2 Semantics

We propose a formal semantics to recognize valid and invalid behavior for some given interaction with constraints. The semantics is divided into two fragments: one that does not contain occurrences of the `neg` nor `assert` operators (positive fragment), and the other that contains them (negative fragment).

There are several semantic paradigms which must be taken into consideration when dealing with concurrency, as discussed in [Stö04c]. The semantics introduced in this work can be described as a *true concurrency linear-time trace-based semantics of complete traces with real time*. The terms involved in this classification are explained below.

- ***true concurrency***. There is a partial order over events of an interaction, two events can occur concurrently (not an interleaving semantics)
- ***linear-time***. There is only one global time, no notion of concurrent clocks
- ***trace-based***. The semantics is based on the notion of trace (system-run) over which two satisfaction relations on interactions are defined
- ***complete traces***. A system-run is valid only if it is completely consumed (except for the negation), i.e., system-run prefixes are not considered by the positive satisfaction relation.
- ***real time***. Real-time is considered by allowing OCL/RT constraints

The semantical unification was based on the idea of the denotational semantics of interactions [CK04b] and using the notion of system-run [CK02]. In what follows we first introduce the unified semantic domains of the solution and then we present the positive and negative fragments of the denotational semantics.

3.2.1 Semantic Domains

We assume four primitive domains for *events* $\mathbb{E}$, *messages* $\mathbb{M}$, *abstract time points* $\mathbb{T}$ and natural numbers $\mathbb{N}$.

Events

An event $\eta \in \mathbb{E}$ is either of the form $\texttt{snd}(s,r,m)$ or of the form $\texttt{rcv}(s,r,m)$, representing the dispatch and the arrival of *message* m from *sender* instance

s to *receiver* instance r, respectively. Events are unequivocally identified, i.e., two arbitrary events, even if they are both a send (or receive) event of the same message from the same sender to the same receiver, can be distinguished.

Certain events are made anonymous and/or unobservable by replacing them by a predefined *silent event* (in the sense of process algebra [Mil80]) denoted by τ. We define the domain $\mathbb{E}_\tau$ of events and the silent event τ as $\mathbb{E} \cup \{\tau\}$. A silent event represents an event that is not of interest for some given interaction, e.g. an attribute value change, but its occurrence is considered for a state change. System runs contain every event occurred during the system execution while interactions can specify partial behavior of the system.

We say that the instance s is active for $\texttt{snd}(s,r,m)$ and, similarly, that the instance r is active for $\texttt{rcv}(s,r,m)$. We define a binary, symmetric conflict relation $\asymp \subseteq \mathbb{E} \times \mathbb{E}$ on events: If an instance is active for both events η_1 and η_2 then $\eta_1 \asymp \eta_2$. Trivially, $\tau \not\asymp \eta$ for any $\eta \in \mathbb{E}_\tau$.

Given an event $\eta \in \mathbb{E}$, the occurrence time of the event can be retrieved by the map at: $\mathbb{E} \to \mathbb{T}$. $\mathbb{T}$ is equipped with a total ordering relation $\leq$ for comparing time values, and binary operations $+$ and $-$ for adding and subtracting time values respectively.

We write $lin_{at}(E)$ for all possible linearizations of the set of events E given by the function at, i.e., $l \in lin_{at}(E)$ if, and only if l is the isomorphism class $[(X, \leq_X, \lambda_X)]$ of the totally ordered, labeled sets $(X,\leq_X, \lambda_X)$ with λ_X: $X \to E$ bijective and such that $\forall x_1, x_2 \in X.\ at(\lambda_X(x_1)) < at(\lambda_X(x_2)) \implies x_1 \leq_X x_2$ and $\forall x_1, x_2 \in X.\ x_1 \leq_X x_2$ or $x_2 \leq_X x_1$.

Basic Interactions

Basic interactions are modeled using event-labeled pomsets. A partially ordered, labelled multiset, or *pomset* [Pra86], is the isomorphism class $[(X,\leq_X,\lambda_X)]$ of a labeled partial order $(X,\leq_X,\lambda_X)$ w.r.t. monotone, label-preserving maps. An event-labeled pomset is a pomset with an injective labeling function λ_X: $X \to \mathbb{E}$ (or simply λ) from elements in the pomset to events. The order $x_1 \leq_X x_2$ is interpreted as "the occurrence of event $\lambda(x_1)$ precedes the occurrence of event $\lambda(x_2)$".

We write *lin(p)* for all possible linearizations of a pomset p, i.e., all traces that extend the ordering of p: $[(X', \leq_{X'}, \lambda_{X'})] \in lin([(X, \leq_X, \lambda_X)])$ if, and only if $X' = X$, $\lambda_{X'} = \lambda_X$, and $\leq_X \subseteq \leq_{X'}$ where $x_1 \leq_{X'} x_2$ or $x_2 \leq_{X'} x_1$ for all $x_1, x_2 \in X'$.

A basic interaction is given by an event-labeled pomset $[(X,\leq_X,\lambda_X)]$ such

that conflicting events do not occur concurrently, i.e., $x_1, x_2 \in X$ with $\lambda_X(x_1) \not\asymp \lambda_X(x_2)$, then $x_1 \leq_X x_2 \vee x_2 \leq_X x_1$. The empty interaction [(∅,∅,∅)] is denoted by ε.

System Runs

System states are formalized by *dynamic bases* [CK01]. A dynamic basis comprises an implementation of the predefined OCL types and their operations as well as the set of current instances of classes of a static basis together with their attribute valuations, connections to other instances, and implementations of operations, as well as predefined OCL/RT extensions. However, dynamic basis do not contain event instances.

A *system-run* ρ is a finite or infinite sequence of pairs of dynamic bases and finite sets of events possibly containing the silent event τ

$$(\omega_0,H_0),(\omega_1,H_1),(\omega_2,H_2),... \in (\Sigma \times \wp_{\leq_\omega}\mathbb{E}_\tau)^* \cup (\Sigma \times \wp_{\leq_\omega}\mathbb{E}_\tau)^\infty$$

such that $at(\eta_1) < at(\eta_2)$ $\forall \eta_1 \in H_i \setminus \{\tau\}$, $\eta_2 \in H_j \setminus \{\tau\}$ with $i < j$. The dynamic basis ω_0 defines the initial system state; ω_n is transformed into ω_{n+1} by a single system step where the events in H_n occur.

We denote by $\omega(\rho)_n$ the nth dynamic basis in ρ, by $H(\rho)_n$ the nth set of events in ρ, that is ω_n and H_n respectively. Moreover, $\rho(n)$ denotes the nth pair of dynamic basis and set of events in the system-run ρ. The empty system-run is denoted by ε.

Given an OCL/RT constraint c to be evaluated over a system-run ρ and a variable environment γ assigning values to variables (including `self`), we write $(\rho, i);\gamma \vdash \varphi \downarrow v$ for the judgement that φ evaluates to the value v at the ith system state in the trace ρ using the variable environment γ.

System Run Operators

We define concurrence, strict sequencing, weak sequencing and filtering of system-runs. Previously we present the following auxiliary definitions.

Given a system-run ρ, we define the partition of a system-run ρ^* as the set of pairs (ρ_1,ρ_2) with ρ_1 and ρ_2 system-runs such that $\forall\ j \in \mathbb{N}.\ \omega(\rho_i)_j = \omega(\rho)_j$, $i = 1,2$ and $\forall\ j \in \mathbb{N}.\ \{H(\rho_1)_j,\ H(\rho_2)_j\}$ is a partition of the set of events $H(\rho)_j$.

Given a system-run ρ, we define $rem(\rho)$ as the system-run ρ' resulting from removing every pair $(\omega_i,\emptyset)$ from ρ.

The definitions of the operators are given as follows.

Given a system-run ρ, we define concurrence $\rho_{\|}$ as the set of pairs of system-runs ($rem(\rho_1)$, $rem(\rho_2)$) such that (ρ_1,ρ_2) $\in \rho^*$.

Given a system-run ρ, we define strict sequencing $\rho_;$ as the set of pairs (ρ_1,ρ_2) $\in \rho_{\|}$ such that $\exists\ k \in \mathbb{N}\ \forall i \in \mathbb{N}.\ 0 \leq i \leq k \Rightarrow H(\rho_1)_i = H(\rho)_i$ and $k < i \Rightarrow H(\rho_2)_{i-k} = H(\rho)_i$.

Given a system-run ρ, we define weak sequencing $\rho_{;\!\!\approx}$ as the set of pairs of system-runs (ρ_1,ρ_2) $\in \rho_{\|}$ such that $\forall \eta_1 \in \bigcup_n H(\rho_1)_n \setminus \{\tau\}$, $\eta_2 \in \bigcup_n H(\rho_2)_n \setminus \{\tau\}$. $\eta_1 \approx\!\!\!\!\!\!\;\;\; \eta_2 \Rightarrow at(\eta_1) < at(\eta_2)$.

Given a system-run ρ, we define filtering of system-runs $\mathit{filter}(M)(\rho)$ as the set of system-runs resulting from removing some elements of ρ whose labeling events correspond to some message in $M \in \mathbb{M}$ and eliminating empty sets of events, i.e., $rem(\rho') \in \mathit{filter}(M)(\rho)$ if $\forall i \in \mathbb{N}.\ \omega(\rho)_i = \omega(\rho')_i$ and $H(\rho')_i \subseteq H(\rho)_i$, and $\forall \eta \in H(\rho)_i \backslash H(\rho')_i$. η is either of the form $\texttt{snd}(s,r,m)$ or $\texttt{rcv}(s,r,m)$ for $m \in M$.

3.2.2 The Positive Fragment

The semantics of the positive fragment of the language (with no occurrences of negation and assertion) is defined by an inductive positive satisfaction relation between system-runs and interactions as in [CK04b]. The relation is denoted by $\rho \models_p S$ and read system-run ρ *positively satisfies* interaction S. The semantics is shown in Figure 3.4.

The base case considers a system-run possibly containing one or more occurrences of the silent event τ. Silent events are ignored in order to verify the satisfaction of a given system-run. A system-run satisfies a basic interaction if every possible linearization of the events (with respect to their occurrence time) within the system-run (ignoring silent events) is included in the linearization of the basic interaction (with respect to its associated partial ordering).

In order to satisfy `strict`, `seq` and `par` operators, the system-run is partitioned in two and the satisfaction verified for each resulting system-run. Each partition keeps unchanged the dynamic bases but each set of events is partitioned. This partition is defined by the concurrence, strict sequencing and weak sequencing of system-runs presented before. Performing a partition of a system-run may produce empty set of events H_i in the resulting system-runs. Those sets of events are removed, with their corresponding dynamic bases, since they represent transitions taken by some other parts of the interaction.

$\rho \models_p B$ if $lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}) \subseteq lin(B)$
$\rho \models_p \texttt{strict}(S_1,S_2)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{;}.\ \rho_i \models_p S_i\ (i = 1,2)$
$\rho \models_p \texttt{seq}(S_1,S_2)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{;_{\gg}}.\ \rho_i \models_p S_i\ (i = 1,2)$
$\rho \models_p \texttt{par}(S_1,S_2)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{\|}.\ \rho_i \models_p S_i\ (i = 1,2)$
$\rho \models_p \texttt{loop}(0,0,S)$ if $\bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset$
$\rho \models_p \texttt{loop}(0,n{+}1,S)$ if $\bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset \vee \rho \models_p \texttt{seq}(S,\texttt{loop}(0,n,S))$
$\rho \models_p \texttt{loop}(m{+}1,n{+}1,S)$ if $\rho \models_p \texttt{seq}(S,\texttt{loop}(m,n,S))$
$\rho \models_p \texttt{loop}(m,\infty,S)$ if $\exists\ n \geq m.\rho \models_p \texttt{loop}(m,n,S)$
$\rho \models_p \texttt{ignore}(M,S)$ if $\exists\ \rho_1.\ \rho_1 \in \mathit{filter}(M)(\rho) \wedge \rho_1 \models_p S$
$\rho \models_p \texttt{alt}(S_1,S_2)$ if $\rho \models_p S_1 \vee \rho \models_p S_2$

$\rho \models_p \texttt{constraint}(\varphi,S)$ if $(\rho, 0); \emptyset \vdash \varphi \downarrow \text{true} \wedge \rho \models_p S$

Figure 3.4: Semantics of the positive fragment

The `ignore` operator is evaluated from filtering some events of the original system-run. The `loop` operator is considered equivalent to its unfolding as weak sequencing of its interaction arguments. The `alt` operator is evaluated as the disjunction of the satisfaction of the system-run in both interaction fragments.

Finally, there is a satisfaction rule to evaluate whether a system-run positively satisfies an OCL/RT-constrained interaction. The `constraint` combined fragment is interpreted as a conjunction where both the constraint and the interaction must be satisfied in order to consider positive a system-run. In this case, it is not possible to consider positive a system-run where constraints are not valid, not even the empty system-run (hard constraints).

Each constraint is defined locally, this means that in `constraint`(φ,S) the constraint φ exclusively predicates over the interaction S. This concept is broadly discussed in Chapter 6. This allows a compositional definition of the semantics avoiding global constraints. Up to now, this knowledge is needed to understand that in the evaluation process we are discarding those dynamic bases processed by other fragments, so if the set of events associated is empty we discard that part of the trace. An empty set of events H_i represents that the transition between the system states represented by the dynamic bases ω_i and ω_{i+1} is processed by some other fragment of the interaction, since it will not be part of the scope of a future constraint, it can be ignored.

Example (Positive Fragment)

Consider the example given in Section 3.1 and the following system-runs.

$$\rho_1 = (\omega_0, \{\ \texttt{snd}(a,b,n,2),\ \texttt{rcv}(a,b,n,4)\ \})$$

$$\rho_2 = (\omega_0, \{\ \texttt{snd}(b,a,m,6),\ \texttt{rcv}(b,a,m,8)\ \})$$

$$\rho_3 = (\omega_0, \{\ \texttt{snd}(a,b,n,2),\ \texttt{rcv}(a,b,n,4)\})\ (\omega_1, \{\ \texttt{snd}(b,a,m,6),\ \texttt{rcv}(b,a,m,8)\})$$

For the interaction S_1 we have that ρ_1 and ρ_3 positively satisfy $\texttt{seq}(B_1, \texttt{loop}(0,N,B_2))$. In the case of ρ_1, it satisfies the interaction due to the operator `loop` accepts the empty system-run and $(\rho_1, \varepsilon) \in \rho_{1;\!\!\gtrdot}$. Since there are no occurrences of message m, the constraint φ_1 is trivially satisfied, consequently $\rho_1 \models_p \texttt{constraint}(\varphi_1, \texttt{seq}(B_1, \texttt{loop}(0,N,B_2)))$.

Besides, the system-run ρ_3 positively satisfies S_1 since it traverses the interaction (there is only one iteration of the loop) and also the send of m (at=6) is done before 10 time units since the reception of n (at=4). The proof of this is developed as follows.

$$\dfrac{\dfrac{\rho_3(0) \models_p B_1 \quad \rho_3(1) \models_p \texttt{loop}(0,N,B_2) \quad (\rho_3(0), \rho_3(1)) \in \rho_{3;\!\!\gtrdot}}{\rho_3 \models_p \texttt{seq}(B_1,\texttt{loop}(0,N,B_2))} \qquad (\rho_3, 0); \emptyset \vdash \varphi_1 \downarrow \text{true}}{\rho_3 \models_p \texttt{constraint}(\varphi_1,\texttt{seq}(B_1,\texttt{loop}(0,N,B_2)))}$$

If the value of *at* for $\texttt{snd}(b,a,m)$ is, for example, 15, the constraint is not satisfied and consequently the system-run is not positive.

For the interaction S_2 we will see in the next subsection that only the system-run ρ_1 could positively satisfy it (we need first to introduce the positive satisfaction rule for `assert`). The system-run ρ_2 is not positive but negative as we will see in the next subsection. The system-run ρ_3 is neither positive nor negative for this interaction.

3.2.3 The Negative Fragment

The semantics of the negative fragment of the language (with occurrences of negation and assertion) is defined by an inductive negative satisfaction relation between system-runs and interactions as in [CK04b]. The relation is denoted by $\rho \models_n S$ and read system-run ρ *negatively satisfies* interaction S. The semantics is shown in Figure 3.5.

As in [CK04b] we regard the empty system-run and also those with events besides possibly τ as being positive for $\texttt{neg}(S)$. We made a reformulation

$\rho \models_p \texttt{neg}(S)$ if $\bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset$
$\rho \models_p \texttt{assert}(S)$ if $\rho \models_p S$

$\rho \models_n \texttt{strict}(S_1,S_2)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{;}.\ (\rho_1 \models_n S_1 \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2))$
$\rho \models_n \texttt{seq}(S_1,S_2)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{;\!\!\!\diagup}.\ (\rho_1 \models_n S_1 \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2))$
$\rho \models_n \texttt{par}(S_1,S_2)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{\|}.\ ((\rho_1 \models_n S_1 \wedge \rho_2 \models_n S_2) \vee (\rho_1 \models_n S_1 \wedge \rho_2 \models_p S_2) \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2))$
$\rho \models_n \texttt{loop}(0,n{+}1,S)$ if $\rho \models_n \texttt{seq}(S,\texttt{loop}(0,n,S))$
$\rho \models_n \texttt{loop}(m{+}1,n{+}1,S)$ if $\rho \models_n \texttt{seq}(S,\texttt{loop}(m,n,S))$
$\rho \models_n \texttt{loop}(m,\infty,S)$ if $\exists\ n \geq m.\ \rho \models_n \text{loop}(m,n,S)$
$\rho \models_n \texttt{ignore}(M,S)$ if $\exists\ \rho_1.\ \rho_1 \in \mathit{filter}(M)(\rho) \wedge \rho_1 \models_n S$
$\rho \models_n \texttt{alt}(S_1,S_2)$ if $\rho \models_n S_1 \wedge \rho \models_n S_2$
$\rho \models_n \texttt{neg}(S)$ if $\exists\ (\rho_1,\rho_2) \in \rho_{;}.\ \rho_1 \models_p S$
$\rho \models_n \texttt{assert}(S)$ if $\rho \not\models_p S$

$\rho \models_n \texttt{constraint}(\varphi,S)$ if $(\rho,0); \emptyset \vdash \varphi \downarrow \text{false} \vee \rho \models_n S$

Figure 3.5: Semantics of the negative fragment

for the negative satisfaction rule following the same idea as sequential operators (`seq` and `strict`): after traversing a negative fragment a system-run will always be negative no matter what happens afterwards. A system-run positively satisfying S is positive for $\texttt{assert}(S)$ since S is the only valid execution, otherwise the system-run is negative.

For the operators `strict` and `seq` we adopt the view that only those system-runs are negative that either run through the first operand negatively or fulfil the first operand positively but the second operand negatively. A similar stance is taken towards `par` where either both operands have to be run through negatively or one of the operands negatively and the other one positively in order to make a run negative. The `loop` and `ignore` operators follow the same idea as in the positive fragment of the semantics, while for the `alt` operator both operands have to be run through negatively (negation of the positive satisfaction rule).

Finally, there is a satisfaction rule to evaluate whether a system-run negatively satisfies an OCL/RT constrained interaction. The negative rule is interpreted as the "negation" of the positive satisfaction rule: a system-run ρ negatively satisfies a constrained interaction $\texttt{constraint}(\varphi,S)$ if either the constraint φ evaluates to false or the system-run ρ negatively satisfies S.

Example (Negative Fragment)

Consider the example introduced in Section 3.1 and the last two system-runs from the example given in Section 3.2.2.

$$\rho_2 = (\omega_0, \{\ \texttt{snd}(b,a,m,6),\ \texttt{rcv}(b,a,m,8)\ \})$$

$$\rho_3 = (\omega_0, \{\ \texttt{snd}(a,b,n,2),\ \texttt{rcv}(a,b,n,4)\})\ (\omega_1, \{\ \texttt{snd}(b,a,m,6),\ \texttt{rcv}(b,a,m,8)\})$$

We have that only the system-run ρ_3 would negatively satisfy the interaction S_1 depending on the value of *at* for $\texttt{snd}(b,a,m)$. If the time stamp were 15, the constraint would not be satisfied and consequently the system-run would be negative for S_1. The proof of this case is as follows.

$$\dfrac{\dfrac{\rho_3(0) \models_p B_1 \quad \rho_3(1) \models_p \texttt{loop}(0,N,B_2) \quad (\rho_3(0),\rho_3(1)) \in \rho_{3;\gtrless}}{\rho_3 \models_p \texttt{seq}(B_1,\texttt{loop}(0,N,B_2))} \quad (\rho_3,0);\emptyset \vdash \varphi_1 \downarrow \text{false}}{\rho_3 \models_n \texttt{constraint}(\varphi_1,\texttt{seq}(B_1,\texttt{loop}(0,N,B_2)))}$$

For the interaction S_2 we have that the system-run ρ_1 could positively satisfy the interaction since it completely traverses the first fragment of the interaction. The positive satisfaction depends on the value of the attribute *ok* of *a* to verify the constraint φ_2. The system-run ρ_1 cannot negatively satisfy the interaction S_2. The system-run ρ_2 negatively satisfies the interaction S_2 since it traverses the negative region defined by the operator `neg` and the assertion cannot be satisfied, as shown next.

$$\dfrac{\dfrac{\dfrac{\rho_2 \not\models_p B_1}{\rho_2 \models_n \texttt{assert}(B_1)}}{\rho_2 \models_n \texttt{constraint}(\varphi_2,\texttt{assert}(B_1))} \quad \dfrac{\rho_2 \models_p B_2}{\rho_2 \models_n \texttt{neg}(B_2)}}{\rho_2 \models_n \texttt{alt}(\texttt{constraint}(\varphi_2,\texttt{assert}(B_1)),\texttt{neg}(B_2))}$$

3.2.4 Summary of Constraints Satisfaction

The semantics defined in the previous sections, plainly worded, states that "the system-run is *bad* as soon as it leaves a negative region, it is *good* if both it is exhaustive (i.e., the interaction does not specify any event beyond the system-run's last event) and it only traverses positive regions, and it is *inconclusive* otherwise." [CK04b]

A system-run ρ is positive for an interaction $\texttt{constraint}(\varphi,S)$ if the system-run positively satisfies the interaction S and also φ evaluates to true in ρ. If ρ negatively satisfies the interaction S (no matter what happens with φ) or φ evaluates to false in ρ (no matter what happens with S), the system-run ρ

negatively satisfies the interaction $\texttt{constraint}(\varphi,S)$. In any other case, the system-run ρ is considered inconclusive. Figure 3.6 resumes these results.

Notation: We use the abbreviations $\rho \models_i S$ for $\rho \not\models_p S \wedge \rho \not\models_n S$, and $\rho \vdash \varphi$ for $(\rho, 0); \emptyset \vdash \varphi$.

	$\rho \vdash \varphi \downarrow$ true	$\rho \vdash \varphi \downarrow$ false	$\rho \vdash \varphi \downarrow$ undef
$\rho \models_p S$	$\rho \models_p \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_i \texttt{constraint}(\varphi,S)$
$\rho \models_n S$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$
$\rho \models_i S$	$\rho \models_i \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_i \texttt{constraint}(\varphi,S)$

Figure 3.6: Summary of constraints satisfaction

Finally, a system-run can be both positive and negative. In this case, the interaction is called an *overspecified* interaction. As an example, consider the interaction $\texttt{strict}(\texttt{neg}(S),S)$ and a system-run positively satisfying S.

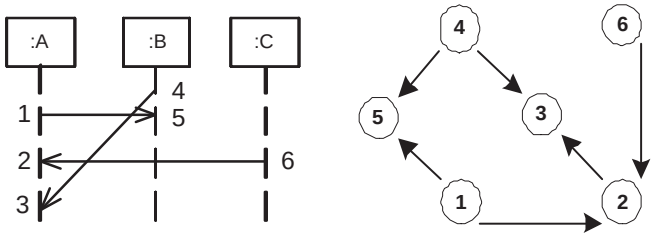
3.3 Discussion: Basic vs Event

In the definition of the semantics we use as the base case the basic interaction *Basic*, where a basic interaction can be seen as an old sequence diagram without any operator involved. The advantage of this is that we can model a basic interaction as a pomset which defines an ordering within the set of events. However, it is interesting to know if we can use an event as a simpler base case.

Pomset theory states that every pomset is representable as the set of its linearizations. As *Basic* is represented as a pomset, we can represent a basic interaction as an alternative (operator `alt`) of all its possible linearizations. However, this is not possible if *Basic* is infinite, because we cannot represent an infinite disjunction with a finitely generated abstract syntax. UML 2.0 specification does not specify whether basic interactions are finite or infinite. Intuition leads us to consider a basic interaction as finite due to the fact that graphical representation of interactions only allows representing finite dispatch of messages, except for the operator `loop` that is not considered in a basic interaction.

Notation: By abuse of notation we abbreviate
$\texttt{strict}(S_1, \texttt{strict}(S_2, \texttt{strict}(. \ . \ . \ , S_n)))$ to $\texttt{strict}(S_1, S_2, \ . \ . \ . \ , S_n)$ and $\texttt{alt}(S_1, \texttt{alt}(S_2, \texttt{alt}(. \ . \ . \ , S_n)))$ to $\texttt{alt}(S_1, S_2, \ . \ . \ . \ , S_n)$.

If we consider *Basic* as finite, then we can represent it as an alternative of



(a) Basic interaction (b) Graph of dependencies

Figure 3.7: Basic interaction

its linearizations. Consider the example shown in Figure 3.7(a), the basic interaction (call it B) can be described as `alt`(B_1, B_2,..., B_n) where the B_i's are the possible linearizations, defined as follows.

Definition 3.3.1. (Syntactic Linearization)
Let B be a basic interaction represented by the pomset $[(X,\leq_X,\lambda_X)]$, then B_i represents a linearization of B if B_i = `strict`(η_1,...,η_n) such that λ_X: $X \rightarrow \bigcup \eta_i$ and $\forall i, j.\ \eta_i = \lambda_X(x), \eta_j = \lambda_X(y),\ x \neq y.\ x \leq_X y \Rightarrow i < j$.

□

If we consider the graph shown in Figure 3.7(b), representing the implicit partial order (where each arrow from a to b represents that a occurs before b), possible linearizations are `strict`(1,4,5,6,2,3), `strict`(4,1,5,6,2,3) and `strict`(4,1,6,2,5,3), among others.

If only finite event-labeled pomsets are used to define a basic interaction, and given that, on the one hand, these pomsets are equivalent to the disjunction of their linearizations and, on the other, the operator `alt` provides disjunction, then events can be used as base case instead of *Basic* in the grammar of Figure 3.4. The base case of the satisfaction relation should state that $\rho \models_p \eta$ if $\bigcup H(\rho)_n \setminus \{\tau\} = \{\eta\}$, where η is an event. In this way, the semantic definition of (constrained) interactions can be substantially simplified.

3.4 Related Work

There is a large amount of work defining the semantics of scenario-based descriptions including UML 1.x Sequence Diagrams [LLH04, Kna99] and Message Sequence Charts [Krü00, LKK+02].

To our interest, there are works which define the semantics of UML 2.0 Interaction. Cengarle and Knapp [CK04b], Haugen and Stølen [HKHS05], and Störrle [Stö04c] define denotational semantics closely related to our semantics. Cengarle and Knapp in [CK05] and Lund and Stølen [LS06] also define a formal semantics of UML 2.0 Interactions. We will not include these two works in our comparison since they use an operational approach.

Finally, we will briefly present related work from Grosu and Smolka [GS05], Hammal [Ham06], and Cavarra and Küster-Filipe [CKF04]. These works present an automata-theoretic semantics for UML 2.0 Interactions.

3.4.1 Related Semantics

Our semantics is a *true concurrency linear-time trace-based semantics of complete traces with real time*. In contrast, Störrle's semantics, as the one developed by Cengarle and Knapp, are interleaving semantics, while the one by Haugen and Stølen is true-concurrent. The original semantics of Cengarle and Knapp does not consider time at all while the others consider abstract time.

Positive and Negative Behavior

The semantics of the positive fragment defined above coincides with the interpretation given by Störrle where a trace must completely traverse a positive region in order to be considered positive. The semantics of the negative fragment has strong relationship with the one defined by Haugen and Stølen [HS03].

Störrle considers three different interpretations of `neg`(S). All of them coincide in declaring negative for `neg`(S) all those traces that are positive for S. For the positive traces of `neg`(S), interpretation (1), called "not the [valid] traces of S", assigns no positive traces to `neg`(S) (our definition coincides with this interpretation); interpretation (2), called "anything but the [valid] traces of S", makes all traces that are not positive for S the positive traces of `neg`(S); interpretation (3) declares the negative traces of S to be the positive traces for `neg`(S). Employing the interpretations (1) or (3), the usage of negation inside combined fragments leads to the undesirable consequence that the overall interaction shows no positive traces at all. Interpretation (2) excludes the possibility of inconclusive traces for `neg`(S).

In contrast, Haugen and Stølen use another interpretation: `neg`(S) only states what we cannot do, but it does not state what we can do. So, the only thing to do is to do nothing (the empty trace ε). The proposal of

Haugen and Stølen states that "[. . .] any trace that [completely traverses a negative region] is a negative scenario. Anything may happen [afterwards], it will never make it positive". It is not explicitly said that further steps cannot make the trace inconclusive. Our semantics follows the same ideas.

Our definition of the negative satisfaction relation for the `alt` operator differs from the view of Störrle, and Haugen and Stølen. They define `alt` as the disjunction of the behaviors specified by each operand. In this sense, one of the operands has to be run through negatively in order to make a run negative. In some cases this seems intuitive as in the example of Figure 3.8, introduced in [HKHS05]. An execution sending and receiving the message *gravy* is considered negative since gravy sauce is not wanted anymore by the cooking apprentice, but beárnaise sauce instead. For our semantics, this case is considered inconclusive since the system-run is negative for the first operand and inconclusive for the other. However, this definition has an important disadvantage which we will discuss in the next chapter.

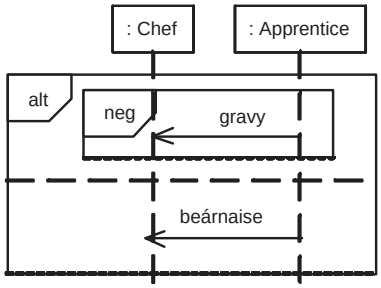


Figure 3.8: Alternative as a disjunction of behaviors

A further deviation from the proposal of Haugen and Stølen, and Störle, is the existence of overspecified interactions (which let some traces be positive and negative at the same time). Ignoring the existence of such traces is restrictive from a semantical point of view.

Our proposal adds another aspect to consider. In the original semantics from Cengarle and Knapp, a negative region was only determined by the trace and the interaction. Now, the positivity/negativity of a system-run is determined also by the OCL/RT constraints. A new satisfaction rule was added in order to evaluate where a system-run positively/negatively satisfies an OCL/RT constrained interaction. Even with constraints we still have inconclusive system-runs and overspecified interactions. Störrle, and Haugen and Stølen incorporate time constraints in their semantics. There,

only basic constraints can be expressed involving duration intervals between two time points (events). The use of OCL/RT allows a powerful specification of constraints, including those considered in related works.

Events and Operators

We use only two kinds of events: send and receive message events. Haugen and Stølen consider another event corresponding to the consumption of the message. This involves storage of events before consumption, scheduling of them, among others. These aspects are associated with a powerful system specification but are left for future study.

Some events from the OCL/RT proposal are discarded because they represent behavior not related to the interactions being modeled, as `SignalEvent`, `ChangeEvent`, `TerminationEvent` and so on. These events will not be part of the OCL/RT constraints specified in this work, but they will be used in a complete specification of the system. All these events are represented within a system-run by the silent event τ and discarded during evaluation.

Some UML 2.0 operators for interactions are not considered, namely `break` and `critical`, and also other interaction components as message parameters, references, gates and continuations (some of them are considered by Haugen and Stølen). In order to complete the semantics given in this work, further work is needed.

Former Semantics

Some changes were made to the semantics defined by Cengarle and Knapp [CK04b]. We use a more expressive notion of trace: a system-run. This notion allows the use of OCL/RT constraints in interactions and the fusion of both semantics. In consequence, the concepts of concurrence, strict sequencing, weak sequencing and filtering of system-runs were redefined in order to adapt the satisfaction rules for `par`, `strict`, `seq` and `ignore`, respectively.

We updated the negative satisfaction relation. The semantics of negation was redefined in the style of the semantics of the sequential composition: after traversing a negative region, a system-run will always be negative no matter what happens afterwards.

Finally, we remark a distinction between the semantics in this work and the original developed by Cengarle and Knapp. As it is proven in Theorem 4.3.2 below, the new semantics is a concretization of the original semantics. This means that the set of positive system-runs accepted by the new semantics is a subset of the positive traces accepted by the original one, negative system-

runs may now be inconclusive, and inconclusive system-runs remain being inconclusive.

3.4.2 Other Semantics

Hammal [Ham06] defines a formal semantics for UML 2.0 Interactions by using a branching time structure rather than traces. This model (a lattice-like graph) records both traces of all interaction components together with possible execution bifurcations and can be directly unfolded into a transition system capturing the intended behavior. It also proposes a method to extract time properties from interactions and add them into the graph in order to achieve timeliness and performance analysis. However, constraints taken into consideration are only basic constraints as in the works of Störrle, and Haugen and Stølen.

Grosu and Smolka [GS05] present an automata-theoretic semantics for interactions that solves how to assign a precise meaning to a set of interactions without compromising refinement. The semantics relies on the observation that negative and positive behavior should be regarded as safety and liveness properties, respectively. Given a set of interactions they construct a safety Büchi automaton from the negative behavior and a liveness Büchi automaton [Büc62] from the positive behavior. The safety and liveness automata ensure that a trace that may either lead to the completion of a bad scenario or prevent the completion of a good scenario is rejected. They take the product of these two automata as the operational semantics of the original set of interactions and the corresponding language as the denotational semantics. The semantics does not consider constraints at interaction level.

Cavarra and Küster-Filipe [CKF04] present an operational semantics which is based on abstract state machines. This is closely related to the idea of state-based constraints satisfaction (as liveness properties). They describe a set of operational rules formalizing the behavior of instances involved in the interaction. According to the location where an instance lies, the semantics decides what step to execute and allows the trace to progress to the next location. The semantics ignores some important combined fragments as `ignore` and `loop` and does not consider constraints. It also adds some features in order to represent mandatory and possible behavior as in Live Sequence Charts. In this way, they add the notion of *locations*, which are points in the diagram. This notion, in our semantics, can not be represented since sometimes a location represents a time point without any event occurring on it (as the beginning of the execution of a combined fragment). In our approach sequence diagrams model behavior in the form of possible interactions. In contrast, our definitions for `opt` and `assert` allows the incorporation of possible and mandatory behavior, respectively.

Summary

In this chapter an extended abstract syntax of UML 2.0 Interactions with OCL/RT constraints support was given, and a true concurrency linear-time trace-based semantics of complete traces with real-time was developed. The semantics allows recognizing positive and negative system-runs with constraints satisfaction, improving the potential of the language for inter-component behavioral specifications.

We also discussed the possibility of simplifying the defined semantics by considering an event as the base case of every interaction instead of a basic interaction. Finally, we made a comparison with other approaches in order to analyze the potential of our work. In this case, as far as we know our work is the only one which considers complex constraints in interactions with a formal semantics supporting it.

Chapter 4

Properties of Constrained Interactions

It is useful to prove that properties from the original semantics given in [CK04b] still hold after defining the modifications. It is also of interest to derive new properties from the semantics to improve its comprehension.

This chapter is structured as follows. Section 4.1 presents basic properties from the original semantics. Section 4.2 presents properties of interest with respect to constrained interactions. Finally, Section 4.3 proves that the semantics of interactions with constraints is a concretization of the previous semantics of interactions without constraints.

4.1 Basic Properties

Basic properties in the original semantics still hold in the new one. In particular, both forms of sequential composition are associative, and parallel and alternative composition are associative and commutative.

Lemma 4.1.1. Let S_1, S_2, and S_3 be interactions, and ρ be a system-run.

1a. $\rho \models_p \texttt{strict}(S_1,\texttt{strict}(S_2,S_3)) \Leftrightarrow \rho \models_p \texttt{strict}(\texttt{strict}(S_1,S_2),S_3)$

Proof. If $\rho \models_p \texttt{strict}(S_1,\texttt{strict}(S_2,S_3))$ then $\exists\ (\rho_1,\rho_2) \in \rho_;$ where $\rho_1 \models_p S_1 \wedge \rho_2 \models_p \texttt{strict}(S_2,S_3)$. In the same way, $\exists\ (\rho_3,\rho_4) \in \rho_{2;}$ where $\rho_3 \models_p S_2 \wedge \rho_4 \models_p S_3$. According to the definition of strict sequencing, ρ is divided into three parts corresponding to ρ_1, ρ_3 and ρ_4 in this order. In this way, $(\rho_1,\rho_3) \in \rho_{5;} \wedge \rho_{5;} \models_p \texttt{strict}(S_1,S_2) \wedge \rho_4 \models_p S_3$. Finally, $(\rho_5,\rho_4) \in \rho$ and $\rho \models_p \texttt{strict}(\texttt{strict}(S_1,S_2),S_3)$.

1b. $\rho \models_n \texttt{strict}(S_1,\texttt{strict}(S_2,S_3)) \Leftrightarrow \rho \models_n \texttt{strict}(\texttt{strict}(S_1,S_2),S_3)$

Proof. If $\rho \models_n \texttt{strict}(S_1,\texttt{strict}(S_2,S_3))$ then $\exists\ (\rho_1,\rho_2) \in \rho_{;}$ where $\rho_1 \models_n S_1$ or $(\rho_1 \models_p S_1 \wedge \rho_2 \models_n \texttt{strict}(S_2,S_3))$. In the same way, $\exists\ (\rho_3,\rho_4) \in \rho_{2;}$ where $\rho_3 \models_n S_2$ or $(\rho_3 \models_p S_2 \wedge \rho_4 \models_n S_3)$. Making some calculations $\rho_1 \models_n S_1$ or $(\rho_1 \models_p S_1 \wedge \rho_3 \models_n S_2)$ or $(\rho_1 \models_p S_1 \wedge \rho_3 \models_p S_2 \wedge \rho_4 \models_n S_3)$. As in the last proof, according to the definition of strict sequencing ρ is divided into three parts corresponding to ρ_1, ρ_3 and ρ_4 in this order. In this way, $(\rho_1,\rho_3) \in \rho_{5;} \wedge \rho_{5;} \models_n \texttt{strict}(S_1,S_2)$ or $(\rho_{5;} \models_p \texttt{strict}(S_1,S_2) \wedge \rho_4 \models_n S_3)$. Finally, $(\rho_5,\rho_4) \in \rho \wedge \rho \models_n \texttt{strict}(\texttt{strict}(S_1,S_2),S_3)$.

2a. $\rho \models_p \texttt{seq}(S_1,\texttt{seq}(S_2,S_3)) \Leftrightarrow \rho \models_p \texttt{seq}(\texttt{seq}(S_1,S_2),S_3)$

Proof. If $\rho \models_p \texttt{seq}(S_1,\texttt{seq}(S_2,S_3))$ then $\exists\ (\rho_1,\rho_2) \in \rho_{;\gg}$ where $\rho_1 \models_p S_1 \wedge \rho_2 \models_p \texttt{seq}(S_2,S_3)$. In the same way, $\exists\ (\rho_3,\rho_4) \in \rho_{2;\gg}$ where $\rho_3 \models_p S_2 \wedge \rho_4 \models_p S_3$. According to the definition of weak sequencing, ρ is partitioned in ρ_1, ρ_3 and ρ_4 without loosing events nor dynamic basis, as well as the order given by *at* is kept. We can compone the system-runs in order to get the original system-run ρ (the composition will result in a set of system-runs with the occurrence of ρ). Consequently, $(\rho_1,\rho_3) \in \rho_{5;\gg}$ and $(\rho_5,\rho_4) \in \rho$ with $\rho_5 \models_p \texttt{seq}(S_1,S_2) \wedge \rho_4 \models_p S_3$. Finally, $\rho \models_p \texttt{seq}(\texttt{seq}(S_1,S_2),S_3)$.

2b. $\rho \models_n \texttt{seq}(S_1,\texttt{seq}(S_2,S_3)) \Leftrightarrow \rho \models_n \texttt{seq}(\texttt{seq}(S_1,S_2),S_3)$

Proof Sketch. This proof is the same as in the case of `strict` with the considerations done in the last proof about decomposition and composition of system-runs using the definition of weak sequencing.

3a. $\rho \models_p \texttt{par}(S_1,\texttt{par}(S_2,S_3)) \Leftrightarrow \rho \models_p \texttt{par}(\texttt{par}(S_1,S_2),S_3)$

3b. $\rho \models_n \texttt{par}(S_1,\texttt{par}(S_2,S_3)) \Leftrightarrow \rho \models_n \texttt{par}(\texttt{par}(S_1,S_2),S_3)$

Proof Sketch. The proof for the positive satisfaction relation is the same as in the case of `strict` and `seq`. Demonstrations for the negative satisfaction relation involve a few more calculations and are straightforward.

4a. $\rho \models_p \texttt{par}(S_1,S_2) \Leftrightarrow \rho \models_p \texttt{par}(S_2,S_1)$

4b. $\rho \models_n \texttt{par}(S_1,S_2) \Leftrightarrow \rho \models_n \texttt{par}(S_2,S_1)$

Proof. If $\rho \models_p \texttt{par}(S_1,S_2)$ then $\exists\ (\rho_1,\rho_2) \in \rho_{\|}$ such that $\rho_1 \models_p S_1 \wedge \rho_2 \models_p S_2$. Finally, by the definition of concurrence, if $(\rho_1,\rho_2) \in \rho_{\|}$ then $(\rho_2,\rho_1) \in \rho_{\|}$ and thus $\rho \models_p \texttt{par}(S_2,S_1)$. Demonstration for the negative satisfaction relation is straightforward.

5a. $\rho \models_p \texttt{alt}(S_1,\texttt{alt}(S_2,S_3)) \Leftrightarrow \rho \models_p \texttt{alt}(\texttt{alt}(S_1,S_2),S_3)$

Proof. $\rho \models_p \texttt{alt}(S_1,\texttt{alt}(S_2,S_3))$ then $\rho \models_p S_1 \vee \rho \models_p \texttt{alt}(S_2,S_3)$. Following the same reasoning, $\rho \models_p \texttt{alt}(S_2,S_3)$ if $\rho \models_p S_2 \vee \rho \models_p S_3$. Finally, we have

the same conditions that in $\rho \models_p \texttt{alt}(\texttt{alt}(S_1,S_2),S_3)$.

5b. $\rho \models_n \texttt{alt}(S_1,\texttt{alt}(S_2,S_3)) \Leftrightarrow \rho \models_n \texttt{alt}(\texttt{alt}(S_1,S_2),S_3)$

Proof. $\rho \models_n \texttt{alt}(S_1,\texttt{alt}(S_2,S_3))$ then $\rho \models_n S_1 \wedge \rho \models_n \texttt{alt}(S_2,S_3)$. Following the same reasoning, $\rho \models_n \texttt{alt}(S_2,S_3)$ if $\rho \models_n S_2 \wedge \rho \models_n S_3$. Finally, we have the same conditions that in $\rho \models_n \texttt{alt}(\texttt{alt}(S_1,S_2),S_3)$.

6a. $\rho \models_p \texttt{alt}(S_1,S_2) \Leftrightarrow \rho \models_p \texttt{alt}(S_2,S_1)$

Proof. If $\rho \models_p \texttt{alt}(S_1,S_2)$ then $\rho \models_p S_1 \vee \rho \models_p S_2$, that is the same condition that in $\texttt{alt}(S_2,S_1)$.

6b. $\rho \models_n \texttt{alt}(S_1,S_2) \Leftrightarrow \rho \models_n \texttt{alt}(S_2,S_1)$

Proof. If $\rho \models_n \texttt{alt}(S_1,S_2)$ then $\rho \models_n S_1 \wedge \rho \models_n S_2$, that is the same condition that in $\texttt{alt}(S_2,S_1)$.

□

4.2 Constraint Properties

We can prove some elementary properties using the satisfaction relation on constraints.

Lemma 4.2.1. Let S be a non overspecified interaction, φ be a constraint and ρ be a system-run.

1. $\rho \models_p \texttt{constraint}(\varphi,S) \Rightarrow \rho \models_n \texttt{constraint}(\texttt{not}(\varphi),S)$

Proof. If $\rho \models_p \texttt{constraint}(\varphi,S)$ then $(\rho,0);\emptyset \vdash \varphi \downarrow \text{true}$. Thus $(\rho,0);\emptyset \vdash \texttt{not}\ \varphi \downarrow \text{false}$, and finally $\rho \models_n \texttt{constraint}(\texttt{not}\ \varphi,S)$.

2. $\rho \models_p \texttt{constraint}(\texttt{not}(\varphi),S) \Rightarrow \rho \models_n \texttt{constraint}(\varphi,S)$

Proof. If $\rho \models_p \texttt{constraint}(\texttt{not}\ \varphi,S)$ then $(\rho,0);\emptyset \vdash \texttt{not}\ \varphi \downarrow \text{true}$. Thus $(\rho,0);\emptyset \vdash \varphi \downarrow \text{false}$, and finally $\rho \models_n \texttt{constraint}(\varphi,S)$.

3. $\rho \models_n \texttt{constraint}(\varphi,S) \wedge \rho \models_p S \Rightarrow \rho \models_p \texttt{constraint}(\texttt{not}(\varphi),S)$

Proof. If $\rho \models_n \texttt{constraint}(\varphi,S) \wedge \rho \models_p S$ then $(\rho,0);\emptyset \vdash \varphi \downarrow \text{false}$. Thus, $(\rho,0);\emptyset \vdash \texttt{not}\ \varphi \downarrow \text{true}$ and finally $\rho \models_p \texttt{constraint}(\texttt{not}\ \varphi,S)$.

4. $\rho \models_n \texttt{constraint}(\texttt{not}(\varphi),S) \wedge \rho \models_p S \Rightarrow \rho \models_p \texttt{constraint}(\varphi,S)$

Proof. If $\rho \models_n \texttt{constraint}(\texttt{not}\ \varphi,S) \wedge \rho \models_p S$ then $(\rho,0);\emptyset \vdash \texttt{not}\ \varphi \downarrow \text{false}$. Thus, $(\rho,0);\emptyset \vdash \varphi \downarrow \text{true}$, and finally $\rho \models_p \texttt{constraint}(\varphi,S)$.

□

4.3 Semantical Concretization

We can prove that the semantics of interactions with constraints is a concretization of the original semantics of interactions without constraints, i.e., the new semantics recognizes more specific behavior than the original one due to the incorporation of constraints. In other words, if some execution is considered valid for a given interaction by the new semantics then it is also considered valid by the previous semantics discarding constraint satisfaction.

To prove this we define the idea of interactions without OCL/RT constraints.

Definition 4.3.1. (Untimed Interactions)
The function *untimes* from interactions with OCL/RT constraints to interactions without them is given by recursion on the syntactic structure of its arguments as follows:

$$\begin{aligned}
\mathit{untimes}(B) &= B \\
\mathit{untimes}(\texttt{strict}(S_1,S_2)) &= \texttt{strict}(\mathit{untimes}(S_1),\mathit{untimes}(S_2)) \\
\mathit{untimes}(\texttt{seq}(S_1,S_2)) &= \texttt{seq}(\mathit{untimes}(S_1),\mathit{untimes}(S_2)) \\
\mathit{untimes}(\texttt{par}(S_1,S_2)) &= \texttt{par}(\mathit{untimes}(S_1),\mathit{untimes}(S_2)) \\
\mathit{untimes}(\texttt{loop}(m,\bar{n},S)) &= \texttt{loop}(m,\bar{n},\mathit{untimes}(S)) \\
\mathit{untimes}(\texttt{ignore}(M,S)) &= \texttt{ignore}(M,\mathit{untimes}(S)) \\
\mathit{untimes}(\texttt{alt}(S_1,S_2)) &= \texttt{alt}(\mathit{untimes}(S_1),\mathit{untimes}(S_2)) \\
\mathit{untimes}(\texttt{neg}(S)) &= \texttt{neg}(\mathit{untimes}(S)) \\
\mathit{untimes}(\texttt{assert}(S)) &= \texttt{assert}(\mathit{untimes}(S)) \\
\mathit{untimes}(\texttt{constraint}(\varphi,S)) &= \mathit{untimes}(S)
\end{aligned}$$

where B ranges over basic interactions, S and S_i over interactions, M over sets of messages, m over the natural numbers, and $\bar{n}$ over the natural numbers or ∞.

□

The original semantics is an interleaving semantics (only one event can happen at any given point in time) while the new one is a real-time semantics (events can occur concurrently). Thus, to prove that the new semantics is a concretization of the previous one, we have to prove that if a system-run ρ positively satisfies an interaction considering the new positive satisfaction relation, then every trace in $lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$ positively satisfies the untimed interaction considering the original positive satisfaction relation ($\models_p^*$, shown in Figure 4.1).

$t \models_p^* B$ if $t \in lin(B)$
$t \models_p^* \texttt{strict}(S_1,S_2)$ if $\exists\ t_1,t_2.t = t_1;t_2 \wedge t_1 \models_p^* S_1 \wedge t_2 \models_p^* S_2$
$t \models_p^* \texttt{seq}(S_1,S_2)$ if $\exists\ t_1,t_2.t \in lin(t_1 ;_{\gg} t_2) \wedge t_1 \models_p^* S_1 \wedge t_2 \models_p^* S_2$
$t \models_p^* \texttt{par}(S_1,S_2)$ if $\exists\ t_1,t_2.t \in lin(t_1 \parallel t_2) \wedge t_1 \models_p^* S_1 \wedge t_2 \models_p^* S_2$
$t \models_p^* \texttt{loop}(0,0,S)$ if $t = \varepsilon$
$t \models_p^* \texttt{loop}(0,n+1,S)$ if $t = \varepsilon \vee t \models_p^* \texttt{seq}(S,\texttt{loop}(0,n,S))$
$t \models_p^* \texttt{loop}(m+1,n+1,S)$ if $t \models_p^* \text{seq}(S,\texttt{loop}(m,n,S))$
$t \models_p^* \texttt{loop}(m,\infty,S)$ if $\exists\ n \geq m.\ t \models_p^* \texttt{loop}(m,n,S)$
$t \models_p^* \texttt{ignore}(M,S)$ if $\exists\ t_1.\ t_1 \in \mathit{filter}(M)(t) \wedge t_1 \models_p^* S$
$t \models_p^* \texttt{alt}(S_1,S_2)$ if $t \models_p^* S_1 \vee t \models_p^* S_2$

Figure 4.1: Original semantics of the positive fragment [CK04b]

Notation: For the following demonstrations we consider that t, t_1 and t_3 are traces as defined in [CK04b]: event-labeled pomset whose ordering is total.

Theorem 4.3.2. Let ρ be a system-run, S be an interaction and $\models_p^*$ the positive satisfaction relation defined in [CK04b].

$$\rho \models_p S \Rightarrow \forall t \in lin_{at}(\textstyle\bigcup_i H(\rho)_i \setminus \{\tau\}).\ t \models_p^* \mathit{untimes}(S)$$

Proof. By induction on the syntactic structure of S.

(`Basic`) If $\rho \models_p B$, where B is a basic interaction, then (by definition of positive satisfaction relation) $lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}) \subseteq lin(B)$, this means that $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}).\ t\ \in lin(B)$. Considering also that $\mathit{untimes}(B) = B$ (by definition of *untimes*) then $t \in lin(\mathit{untimes}(B))$. Finally, (by definition of the original positive satisfaction relation) $t \models_p^* \mathit{untimes}(B)$

(`Strict`) If $\rho \models_p \texttt{strict}(S_1,S_2)$ then (by definition of positive satisfaction relation) $\exists\ \rho_1,\rho_2.\ (\rho_1,\rho_2) \in \rho_{;} \wedge \rho_1 \models_p S_1 \wedge \rho_2 \models_p S_2$. Also, (by Lemma 4.3.3(1)) $\forall t\ \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})\ \exists t_1,t_2.\ t = t_1;t_2$ with $t_1 \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$. By inductive hypothesis, $t_1 \models_p^* \mathit{untimes}(S_1)$ and also $t_2 \models_p^* \mathit{untimes}(S_2)$. Thus, by the original positive satisfaction relation, $t \models_p^* \texttt{strict}(\mathit{untimes}(S_1),\mathit{untimes}(S_2))$, and finally (by definition of *untimes*) $t \models_p^* \mathit{untimes}(\texttt{strict}(S_1,S_2))$.

(`Seq`) If $\rho \models_p \texttt{seq}(S_1,S_2)$ then (by definition of positive satisfaction relation) $\exists\ \rho_1,\rho_2.\ (\rho_1,\rho_2) \in \rho_{;_{\gg}} \wedge \rho_1 \models_p S_1 \wedge \rho_2 \models_p S_2$. Also, (by Lemma 4.3.3(2)) $\forall t\ \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})\ \exists t_1,t_2.\ t\ \in lin(t_1 ;_{\gg} t_2)$ with $t_1 \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$. By inductive hypothesis, $t_1 \models_p^* \mathit{untimes}(S_1)$ and also $t_2 \models_p^* \mathit{untimes}(S_2)$. Thus, by the original positive satisfaction relation, $t \models_p^* \texttt{seq}(\mathit{untimes}(S_1),\mathit{untimes}(S_2))$,

and finally (by definition of *untimes*) $t \models_p^* untimes(\texttt{seq}(S_1,S_2))$.

(Par) If $\rho \models_p \texttt{par}(S_1,S_2)$ then (by definition of positive satisfaction relation) $\exists\, \rho_1,\rho_2.\ (\rho_1,\rho_2) \in \rho_{\|} \wedge \rho_1 \models_p S_1 \wedge \rho_2 \models_p S_2$. Also, (by Lemma 4.3.3(3)) $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})\ \exists t_1,t_2.\ t \in lin(t_1 \parallel t_2)$ with $t_1 \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$. By inductive hypothesis, $t_1 \models_p^* untimes(S_1)$ and also $t_2 \models_p^* untimes(S_2)$. Thus, by the original positive satisfaction relation, $t \models_p^* \texttt{par}(untimes(S_1),untimes(S_2))$, and finally (by definition of *untimes*) $t \models_p^* untimes(\texttt{par}(S_1,S_2))$.

($\texttt{Loop}_1$) If $\rho \models_p \texttt{loop}(0,0,S)$ then (by definition of positive satisfaction relation) $\bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset$ and consequently $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}).\ t = \varepsilon$. Finally, by definition of original positive satisfaction relation, $t \models_p^* \texttt{loop}(0,0,untimes(S))$, and then (by definition of *untimes*)
$t \models_p^* untimes(\texttt{loop}(0,0,S))$

($\texttt{Loop}_2$) If $\rho \models_p \texttt{loop}(0,n+1,S)$ then (by definition of positive satisfaction relation) $\bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset \vee \rho \models_p \texttt{seq}(S,\texttt{loop}(0,n,S))$. If $\bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset$ then the proof is the same as the case $\texttt{Loop}_1$. If $\rho \models_p \texttt{seq}(S,\texttt{loop}(0,n,S))$ then by (Seq), $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}).\ t \models_p^* untimes(\texttt{seq}(S,\texttt{loop}(0,n,S)))$, and by definition of *untimes* $t \models_p^* \texttt{seq}(untimes(S),\texttt{loop}(0,n,untimes(S)))$. By definition of original positive satisfaction relation, $t \models_p^* \texttt{loop}(0,n+1,untimes(S))$. Finally, (by definition of *untimes*) $t \models_p^* untimes(\texttt{loop}(0,n+1,S))$

($\texttt{Loop}_3$) If $\rho \models_p \texttt{loop}(m{+}1,n{+}1,S)$ then (by definition of positive satisfaction relation) $\rho \models_p \texttt{seq}(S,\texttt{loop}(m,n,S))$.
By (Seq), $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}).\ t \models_p^* untimes(\texttt{seq}(S,\texttt{loop}(m,n,S)))$, and by definition of *untimes* $t \models_p^* \texttt{seq}(untimes(S),\text{loop}(m,n,untimes(S)))$. By definition of original positive satisfaction relation,
$t \models_p^* \texttt{loop}(m{+}1,n{+}1,untimes(S))$. Finally, (by definition of *untimes*)
$t \models_p^* untimes(\texttt{loop}(m{+}1,n{+}1,S))$

($\texttt{Loop}_4$) If $\rho \models_p \texttt{loop}(m,\infty,\text{S})$ then (by definition of positive satisfaction relation) $\exists\ n \geq m.\rho \models_p \texttt{loop}(m,n,\text{S})$. By one of the cases of (Loop) above, $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}).\ t \models_p^* untimes(\texttt{loop}(m,n,\text{S}))$, and by definition of *untimes*, $t \models_p^* \texttt{loop}(m,n,untimes(\text{S}))$. Finally, by definition of original positive satisfaction relation $t \models_p^* \texttt{loop}(m,\infty,untimes(\text{S}))$, and then, (by definition of *untimes*) $t \models_p^* untimes(\texttt{loop}(m,\infty,\text{S}))$

(Ignore) If $\rho \models_p \texttt{ignore}(M,S)$ then (by definition of positive satisfaction relation) $\exists\ \rho_1.\ \rho_1 \in filter(M)(\rho) \wedge \rho_1 \models_p S$. By Lemma 4.3.3(4), $\forall t_1 \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$, $\exists t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ such that $t \in filter_{pom}(M)(t_1)$. Also (by inductive hypothesis) $\forall t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\}).\ t \models_p^* untimes(S)$. Thus, by definition of original positive satisfaction relation,
$t_1 \models_p^* \texttt{ignore}(M,untimes(S))$. Finally, (by definition of *untimes*) $t_1 \models_p^* untimes(\texttt{ignore}(M,S))$

(`Alt`) If $\rho \models_p \texttt{alt}(S_1,S_2)$ then (by definition of positive satisfaction relation) $\rho \models_p S_1 \vee \rho \models_p S_2$. By inductive hypothesis, $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$. $t \models_p^* untimes(S_1) \vee t \models_p^* untimes(S_2)$. Thus, by original positive satisfaction relation $t \models_p^* \texttt{alt}(untimes(S_1),untimes(S_2))$. Finally, (by definition of *untimes*) $t \models_p^* untimes(\texttt{alt}(S_1,S_2))$

(`Constraint`) If $\rho \models_p \texttt{constraint}(\varphi,S)$ then (by definition of positive satisfaction relation) $(\rho,0)$; $\emptyset \vdash \varphi \downarrow \text{true} \wedge \rho \models_p S$. By inductive hypothesis, $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$. $t \models_p^* untimes(S)$. Thus, (by definition of *untimes*) $t \models_p^* untimes(\texttt{constraint}(\varphi,S))$

(`Neg`) If $\rho \models_p \texttt{neg}(S)$ then (by definition of positive satisfaction relation) $\rho = \varepsilon$ and consequently $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$. $t = \varepsilon$. Finally, by definition of original positive satisfaction relation, $t \models_p^* \texttt{neg}(untimes(S))$, and then (by definition of definition of *untimes*) $t \models_p^* untimes(\texttt{neg}(S))$

(`Assert`) If $\rho \models_p \texttt{assert}(S)$ then (by definition of positive satisfaction relation) $\rho \models_p S$. By inductive hypothesis, $\forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$. $t \models_p^* untimes(S)$. Thus, by definition of original positive satisfaction relation $t \models_p^* \texttt{assert}(untimes(S))$. Finally, (by definition of *untimes*) $t \models_p^* untimes(\texttt{assert}(S))$

□

The proof of the last theorem considering $\models_n$ instead of $\models_p$ is not possible since false constraints make negative a system-run that positively satisfies an interaction. Valid system-runs according to the previous semantics can be considered valid, invalid or inconclusive according to the new semantics since the semantics is more restrictive, due to the addition of constraints are added. In the same way, if some system-run is considered invalid according to the previous semantics it can be considered inconclusive according to the new semantics.

In order to prove this result, we need to prove some properties about linearization.

Lemma 4.3.3. Let ρ, ρ_1 and ρ_2 be system-runs

1. $(\rho_1,\rho_2) \in \rho_; \Rightarrow \forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})\ \exists t_1,t_2.\ t = t_1;t_2$ with $t_1 \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$

Proof. Every $t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$ is a trace of the form $[(X,\leq_X,\lambda_X)]$. Since $(\rho_1,\rho_2) \in \rho_;$, $t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\} \cup \bigcup_i H(\rho_2)_i \setminus \{\tau\})$ for some ρ_1, ρ_2. Thus, t can be written as follows $[(X_1 \cup X_2, (\leq_{X_1} \cup \leq_{X_2} \cup (X_1 \times X_2))^*, \lambda_{X_1} \cup \lambda_{X_2})]$, which is equal to $[(X_1,\leq_{X_1},\lambda_{X_1})];[(X_2,\leq_{X_2},\lambda_{X_2})]$. We then define the traces $t_1 = [(X_1,\leq_{X_1},\lambda_{X_1})] \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 = [(X_2,\leq_{X_2},\lambda_{X_2})] \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$ and hence $t = t_1;t_2$ holds trivially

2. $(\rho_1,\rho_2) \in \rho_{;\gg} \Rightarrow \forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})\ \exists t_1, t_2.\ t \in lin(t_1 ;_{\gg} t_2)$ with $t_1 \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$

Proof. Every $t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$ is a trace of the form $[(X,\leq_X,\lambda_X)]$. Since $(\rho_1,\rho_2) \in \rho_{;\gg}$, $t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\} \cup \bigcup_i H(\rho_2)_i \setminus \{\tau\})$ for some ρ_1, ρ_2. Thus, t can be written as follows $[(X_1 \cup X_2, (\leq_{X_1} \cup \leq_{X_2} \cup \{(x_1, x_2) \in X_1 \times X_2 \mid \lambda_{X_1}(x_1) \gg \lambda_{X_2}(x_2)\})^*, \lambda_{X_1} \cup \lambda_{X_2})]$, which is equal to $[(X_1,\leq_{X_1},\lambda_{X_1})]$ $;_{\gg}$ $[(X_2,\leq_{X_2},\lambda_{X_2})]$. We then define the traces $t_1 = [(X_1,\leq_{X_1},\lambda_{X_1})] \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 = [(X_2,\leq_{X_2},\lambda_{X_2})] \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$ and hence $t \in t_1 ;_{\gg} t_2$. Since t_1 and t_2 are already linearized, $t \in lin(t_1 ;_{\gg} t_2)$ holds.

3. $(\rho_1,\rho_2) \in \rho_{\|} \Rightarrow \forall t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})\ \exists t_1, t_2.\ t \in lin(t_1 \parallel t_2)$ with $t_1 \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$

Proof. Every $t \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$ is a trace of the form $[(X,\leq_X,\lambda_X)]$. Since $(\rho_1,\rho_2) \in \rho_{\|}$, $t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\} \cup \bigcup_i H(\rho_2)_i \setminus \{\tau\})$ for some ρ_1, ρ_2. Thus, t can be written as follows $[(X_1 \cup X_2, \leq_{X_1} \cup \leq_{X_2}, \lambda_{X_1} \cup \lambda_{X_2})]$, which is equal to $[(X_1,\leq_{X_1},\lambda_{X_1})] \parallel [(X_2,\leq_{X_2},\lambda_{X_2})]$. We then define the traces $t_1 = [(X_1,\leq_{X_1},\lambda_{X_1})] \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ and $t_2 = [(X_2,\leq_{X_2},\lambda_{X_2})] \in lin_{at}(\bigcup_i H(\rho_2)_i \setminus \{\tau\})$ and hence $t \in t_1 \parallel t_2$. Since t_1 and t_2 are already linearized, $t \in lin(t_1 \parallel t_2)$ holds.

4. $\rho_1 \in \mathit{filter}(M)(\rho) \Rightarrow \forall t_1 \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$,
$\exists t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\}).\ t \in filter_{pom}(M)(t_1)$
($filter_{pom}$ is the filtering relation for pomsets defined in [CK04b])

Proof. By definition of lin_{at}, every $t_1 \in lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\})$ is a trace of the form $[(X_1,\leq_{X_1},\lambda_{X_1})]$ with λ_{X_1}: $X_1 \rightarrow \bigcup_i H(\rho)_i$ and also every $t \in lin_{at}(\bigcup_i H(\rho_1)_i \setminus \{\tau\})$ is a trace of the form $[(X,\leq_X,\lambda_X)]$ with λ_X: $X \rightarrow \bigcup_i H(\rho_1)_i$. Since $\rho_1 \in \mathit{filter}(M)(\rho)$ then $\leq_X \subseteq \leq_{X_1}$ and $X \subseteq X_1$ where the elements removed are those corresponding to events in M. Finally, taking $\leq_X$ as $\leq_{X_1}$ restricted to those elements in X we conclude that $t \in filter_{pom}(M)(t_1)$ holds.

□

Summary

In this chapter some properties of the semantics were derived, in particular there is a property which states that the semantics of interactions with constraints is just a concretization of the original semantics. This result states a strong relationship between our work and the one in [CK04b].

Chapter 5

Implementation and Refinement

Refinement means to add information to a specification to make it closer to an implementation. This definition is intended for formal verification since during development we want to preserve some properties of the system while it is being refined, and this can be achieved by formally defining the notion of interactions refinement. In this work the notion of refinement by model inclusion is used (as in [CK04b]): a concrete specification refines an abstract specification if any model of the concrete specification is also a model in the abstract one. We adapt these notions to our semantic domains and we analyze their properties. We also introduce the notion of constraint refinement since we are working with constrained interactions.

This chapter is structured as follows. Section 5.1 presents the notions of implementation and refinement of an interaction, and analyzes some refinement properties. Section 5.2 introduces the notion of constraints refinement in the same sense as interactions refinement. Finally, Section 5.3 presents a comparison with other approaches to refinement.

5.1 Definitions

The set of definitions given in [CK04b] about implementation and refinement of an interaction are modified in order to consider our semantic domains. An implementation of an interaction is related to the valid behavior specified by the interaction. We define that a an arbitrary set of system-runs implements some interaction if every system-run within the set does not negatively satisfy the interaction, and there is at least one system-run positively satisfying

the interaction.

Definition 5.1.1. (Implementation of Interactions)
A *process* is an arbitrary set of system-runs. A process I is an implementation of an interaction S, written $I \models S$, if

1. $\exists\ \rho \in I.\ \rho \models_p S$
2. $\forall\ \rho \in I.\ \rho \not\models_n S$

An interaction S is implementable if there is a process I such that $I \models S$. A interaction is contradictory if it is not implementable.

□

As Haugen et al. define in [HKHS05], "refinement means to add information to a specification such that the specification becomes closer to an implementation". It is desirable to count with a formal definition of a system when trying to establish properties at an abstract level and preserve them during refinement. The idea of refinement used in this work is based on the idea of "model inclusion", in the same sense of traditional algebraic specifications: a concrete specification refines an abstract specification if any model of the concrete specification is also a model of the abstract one.

Definition 5.1.2. (Refinement)
An interaction S' refines an interaction S, written $S \rightsquigarrow S$', if any implementation of S' is also an implementation of S, i.e., $\forall\ I.\ I \models S' \Rightarrow I \models S$

□

Refinement implies that the set of genuine positive system-runs (system-runs that are positive and not overspecified) cannot be enlarged, negative system-runs remain negative, inconclusive system-runs can be transformed into negative system-runs and at least one genuine positive system-run is kept, as shown in the following Lemma (originally given in [CK04b]) and resumed in Figure 5.1.

Lemma 5.1.3. Let S and S' be interactions with $S \rightsquigarrow S'$.

1. $\forall\ \rho.\ \rho \not\models_p S \vee \rho \models_n S \Rightarrow \rho \not\models_p S' \vee \rho \models_n S'$

Proof. Suppose $\rho \models_p S' \wedge \rho \not\models_n S'$. Then $\{\rho\} \models S'$, and also $\{\rho\} \models S$ since $S \rightsquigarrow S'$. Thus $\rho \models_p S$ and $\rho \not\models_n S$.

2. If S' is implementable then $\forall\ \rho.\ \rho \models_n S \Rightarrow \rho \models_n S'$

Proof. Let I be a process such that $I \models S'$ and let ρ be a system-run such that $\rho \not\models_n S'$. Then, $I \cup \{\rho\} \models S'$, and thus $I \cup \{\rho\} \models S$ because $S \rightsquigarrow S'$. Finally, $\rho \not\models_n S$.

3. If S' is implementable then $\exists\ \rho.\ \rho \models_p S \wedge \rho \models_p S$'

Proof. Since S' is implementable, there is a system-run ρ such that $\rho \models_p S$' and $\rho \not\models_n S$'. Then, $\{\rho\} \models S$', and since $S \rightsquigarrow S$', $\{\rho\} \models S$. Finally, $\rho \models_p S$. □

System Run	Refined System Run
genuine positive	genuine positive, negative or inconclusive
negative	negative
inconclusive	inconclusive or negative

Figure 5.1: Refinement of Interactions

A desirable property of refinement is that the operators be monotonic with respect to it, since monotonicity ensures compositionality in the sense that the different operands of a specification may be refined separately. Unfortunately, this is not true unless the monotonicity is restricted to refinements by non-overspecified interactions. Although for some operators, like alternative composition, this is enough, in some other cases (as sequential operators when the refinement is done in the first argument) it is necessary to assume that all positive system-runs before refinement are still positive after it. This leads to the following definition.

Definition 5.1.4. (Positive Refinement)
An interaction S' positively refines an interaction S, written $S \rightsquigarrow_p S$', if $S \rightsquigarrow S' \wedge \forall\ \rho.\ \rho \models_p S \Rightarrow \rho \models_p S$'.

□

With the last definition it is possible to define some refinement rules as shown in Figure 5.2. First we prove those rules originally given in [CK04b].

Lemma 5.1.5. Let S, S_1, S_2, S', S_1' and S_2' be interactions with S', S_1' and S_2' implementable and not overspecified.

1. $S_1 \rightsquigarrow_p S_1' \Rightarrow \texttt{strict}(S_1,S_2) \rightsquigarrow_p \texttt{strict}(S_1',S_2)$.

Proof. Let I be a process such that $I \models \texttt{strict}(S_1',S_2)$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{strict}(S_1',S_2)$. Thus (by definition of positive satisfaction relation), $\exists\ (\rho_1, \rho_2) \in \rho_{;}.\ \rho_1 \models_p S_1' \wedge \rho_2 \models_p S_2$. Due to the fact that S_1' is not overspecified, $\rho_1 \not\models_n S_1'$, and thus $\rho_1 \models_p S_1$ and $\rho_1 \not\models_n S_1$ (by Lemma 5.1.3(1)). Finally, $\rho \models_p \texttt{strict}(S_1,S_2)$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{strict}(S_1',S_2)$. Thus (by definition of negative satisfaction relation), $\forall\ (\rho_1, \rho_2) \in \rho_{;}.\ \rho_1 \not\models_n S_1' \wedge (\rho_1 \not\models_p S_1' \vee \rho_2 \not\models_n S_2)$. If $\rho_1 \not\models_n S_1'$ then

(by Lemma 5.1.3(2)) $\rho_1 \not\models_n S_1$. If $\rho_1 \not\models_p S_1$' then (by definition of positive refinement) $\rho_1 \not\models_p S_1$. Putting all the information together, we conclude that $\rho \not\models_n \texttt{strict}(S_1,S_2)$.

From (a) and (b) we conclude that $I \models \texttt{strict}(S_1,S_2)$, thus if $S_1 \rightsquigarrow_p S_1$', then $\texttt{strict}(S_1,S_2) \rightsquigarrow_p \texttt{strict}(S_1',S_2)$.

2. $S_2 \rightsquigarrow S_2' \Rightarrow \texttt{strict}(S_1,S_2) \rightsquigarrow \texttt{strict}(S_1,S_2')$.

Proof. Let I be a process such that $I \models \texttt{strict}(S_1,S_2')$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{strict}(S_1,S_2')$. Thus (by definition of positive satisfaction relation), $\exists\ (\rho_1,\rho_2) \in \rho_{;}.\ \rho_1 \models_p S_1 \wedge \rho_2 \models_p S_2'$. Due to the fact that S_2' is not overspecified, $\rho_2 \not\models_n S_2'$, and thus $\rho_2 \models_p S_2$ and $\rho_2 \not\models_n S_2$ (by Lemma 5.1.3(1)). Finally, $\rho \models_p \texttt{strict}(S_1,S_2)$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{strict}(S_1,S_2')$. Thus (by definition of negative satisfaction relation), $\forall.\ (\rho_1,\rho_2) \in \rho_{;}.\ \rho_1 \not\models_n S_1 \wedge (\rho_1 \not\models_p S_1 \vee \rho_2 \not\models_n S_2')$. Then, (by Lemma 5.1.3(2)) $\rho_2 \not\models_n S_2$. Finally, $\rho \not\models_n \texttt{strict}(S_1,S_2)$.

From (a) and (b) we conclude that $I \models \texttt{strict}(S_1,S_2)$, thus if $S_2 \rightsquigarrow S_2'$, then $\texttt{strict}(S_1,S_2) \rightsquigarrow \texttt{strict}(S_1,S_2')$.

3. $S_1 \rightsquigarrow_p S_1' \Rightarrow \texttt{seq}(S_1,S_2) \rightsquigarrow_p \texttt{seq}(S_1',S_2)$.

Proof Sketch. This proof is exactly the same as the first property of `strict`, only changing ; by $;_{\gg}$.

4. $S_2 \rightsquigarrow S_2' \Rightarrow \texttt{seq}(S_1,S_2) \rightsquigarrow \texttt{seq}(S_1,S_2')$.

Proof Sketch. This proof is exactly the same as the second property of `strict`, only changing ; by $;_{\gg}$.

5. $S_1 \rightsquigarrow_p S_1' \Rightarrow \texttt{par}(S_1,S_2) \rightsquigarrow_p \texttt{par}(S_1',S_2)$.

Proof. Let I be a process such that $I \models \texttt{par}(S_1',S_2)$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{par}(S_1',S_2)$. Thus (by definition of positive satisfaction relation), $\exists\ (\rho_1,\rho_2) \in \rho_{\|}.\ \rho_1 \models_p S_1' \wedge \rho_2 \models_p S_2$. Due to the fact that S_1' is not overspecified, $\rho_1 \not\models_n S_1'$, and thus $\rho_1 \models_p S_1$ and $\rho_1 \not\models_n S_1$ (by Lemma 5.1.3(1)). Finally, $\rho \models_p \texttt{par}(S_1,S_2)$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{par}(S_1',S_2)$. Thus (by definition of negative satisfaction relation), $\forall\ (\rho_1,\rho_2) \in \rho_{\|}.\ (\rho_1 \not\models_n S_1' \vee \rho_2 \not\models_n S_2) \wedge (\rho_1 \not\models_n S_1' \vee \rho_2 \not\models_p S_2) \wedge (\rho_1 \not\models_p S_1' \vee \rho_2 \not\models_n S_2)$. If $\rho_1 \not\models_n S_1'$ then (by Lemma 5.1.3(2)) $\rho_1 \not\models_n S_1$. Else, if $\rho_1 \not\models_p S_1'$ then (by definition of positive refinement) $\rho_1 \not\models_p S_1$. Putting all the information together, we conclude that $\rho \not\models_n \texttt{par}(S_1,S_2)$.

From (a) and (b) we conclude that $I \models \texttt{par}(S_1,S_2)$, thus if $S_1 \rightsquigarrow_p S_1'$, then $\texttt{par}(S_1,S_2) \rightsquigarrow_p \texttt{par}(S_1',S_2)$.

6. $S_2 \rightsquigarrow_p S_2\text{'} \Rightarrow \texttt{par}(S_1,S_2) \rightsquigarrow_p \texttt{par}(S_1,S_2\text{'})$.

Proof. By the last property, we have that $S_2 \rightsquigarrow_p S_2\text{'} \Rightarrow \texttt{par}(S_2,S_1) \rightsquigarrow_p \texttt{par}(S_2\text{'},S_1)$. Now, using the fact that concurrence is commutative (Lemma 4.1.1(4)) we can conclude that $\texttt{par}(S_1,S_2) \rightsquigarrow_p \texttt{par}(S_1,S_2\text{'})$.

7. $S_1 \rightsquigarrow S_1\text{'} \Rightarrow \texttt{alt}(S_1,S_2) \rightsquigarrow \texttt{alt}(S_1\text{'},S_2)$.

Proof. Let I be a process such that $I \models \texttt{alt}(S_1\text{'},S_2)$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{alt}(S_1\text{'},S_2)$. Thus (by definition of positive satisfaction relation), $\rho \models_p S_1\text{'} \vee \rho \models_p S_2$. Due to the fact that S_1' is not overspecified, $\rho_1 \not\models_n S_1$', and thus $\rho_1 \models_p S_1$ and $\rho_1 \not\models_n S_1$ (by Lemma 5.1.3(1)). Finally, $\rho \models_p \texttt{alt}(S_1,S_2)$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{alt}(S_1\text{'},S_2)$. Thus (by definition of negative satisfaction relation), $\rho \not\models_n S_1\text{'} \vee \rho \not\models_n S_2$. If $\rho \not\models_n S_1$' then (by Lemma 5.1.3(2)) $\rho \not\models_n S_1$. Finally, $\rho \not\models_n \texttt{alt}(S_1,S_2)$.

From (a) and (b) we conclude that $I \models \texttt{alt}(S_1,S_2)$, thus if $S_1 \rightsquigarrow S_1$', then $\texttt{alt}(S_1,S_2) \rightsquigarrow \texttt{alt}(S_1\text{'},S_2)$.

8. $S_2 \rightsquigarrow S_2\text{'} \Rightarrow \texttt{alt}(S_1,S_2) \rightsquigarrow \texttt{alt}(S_1,S_2\text{'})$.

Proof. By the last property, we have that $S_2 \rightsquigarrow S_2\text{'} \Rightarrow \texttt{alt}(S_2,S_1) \rightsquigarrow \texttt{alt}(S_2\text{'},S_1)$. Now, using the fact that alternative composition is commutative (Lemma 4.1.1(6)) we can conclude that $\texttt{alt}(S_1,S_2) \rightsquigarrow \texttt{alt}(S_1,S_2\text{'})$.

9. $\texttt{alt}(S_1, S_2) \rightsquigarrow S_1$

Proof. Let I be a process such that $I \models S_1$. On the one hand $\exists\ \rho \in I$ with $\rho \models_p S_1$ and thus $\rho \models_p \texttt{alt}(S_1, S_2)$. On the other hand $\forall\ \rho \in I.\ \rho \not\models_n S_1$ and hence $\rho \not\models_n \texttt{alt}(S_1,S_2)$ for all $\rho \in I$. Finally, we conclude that $I \models \texttt{alt}(S_1, S_2)$, thus $\texttt{alt}(S_1, S_2) \rightsquigarrow S_1$

10. $\texttt{alt}(S_1, S_2) \rightsquigarrow S_2$

Proof Sketch. This proof is developed in the same way as the last property.

11. $S \rightsquigarrow S\text{'} \Rightarrow \texttt{assert}(S) \rightsquigarrow \texttt{assert}(S\text{'})$.

Proof. Let I be a process such that $I \models \texttt{assert}(S\text{'})$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{assert}(S\text{'})$. Thus (by definition of positive satisfaction relation), $\rho \models_p S\text{'}$. Due to the fact that S' is not overspecified, $\rho \not\models_n S$', and thus $\rho \models_p S$ and $\rho \not\models_n S$ (by Lemma 5.1.3(1)). Finally, $\rho \models_p \texttt{assert}(S)$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{assert}(S\text{'})$. Thus (by definition of negative satisfaction relation), $\rho \models_p S\text{'}$. Due to the fact that S' is not overspecified, $\rho \not\models_n S$', and thus $\rho \models_p S$ and $\rho \not\models_n S$ (by Lemma 5.1.3(1)). Finally, $\rho \not\models_n \texttt{assert}(S)$.

From (a) and (b) we conclude that $I \models \texttt{assert}(S)$, thus if $S \rightsquigarrow S'$, then $\texttt{assert}(S) \rightsquigarrow \texttt{assert}(S')$.

12. $S \rightsquigarrow_p S' \Rightarrow \texttt{neg}(S) \rightsquigarrow_p \texttt{neg}(S')$.

Proof. Let I be a process such that $I \models \texttt{neg}(S')$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{neg}(S')$. Thus (by definition of positive satisfaction relation), $\rho = \varepsilon$. Then, $\rho \models_p \texttt{neg}(S)$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{neg}(S')$. Thus (by definition of negative satisfaction relation), $\forall\ (\rho_1, \rho_2) \in \rho_; \wedge\ \rho_1 \not\models_p S'$. Then (by definition of positive refinement) $\rho_1 \not\models_p S$. Finally, $\rho \not\models_n \texttt{neg}(S)$.

From (a) and (b) we conclude that $I \models \texttt{neg}(S)$, thus if $S \rightsquigarrow_p S'$, then $\texttt{neg}(S) \rightsquigarrow_p \texttt{neg}(S')$.

13. $S \rightsquigarrow S' \Rightarrow \texttt{neg}(S') \rightsquigarrow \texttt{neg}(S)$.

Proof. Let I be a process such that $I \models \texttt{neg}(S)$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{neg}(S)$. Thus (by definition of positive satisfaction relation), $\rho = \varepsilon$. Then, $\rho \models_p \texttt{neg}(S')$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{neg}(S)$. Thus (by definition of negative satisfaction relation), $\forall\ (\rho_1, \rho_2) \in \rho_; \wedge\ \rho_1 \not\models_p S$. Then, (by Lemma 5.1.3(1)) $\rho_1 \not\models_p S'$ or $\rho_1 \models_n S'$. If $\rho_1 \not\models_p S'$, then $\rho \not\models_n \texttt{neg}(S')$. If $\rho_1 \models_n S'$, since S' is not overspecified, $\rho_1 \not\models_p S'$, thus $\rho \not\models_n \texttt{neg}(S')$.

From (a) and (b) we conclude that $I \models \texttt{neg}(S')$, thus if $S \rightsquigarrow S'$, then $\texttt{neg}(S') \rightsquigarrow \texttt{neg}(S)$.

□

Now we can complement the set of refinement rules with new ones for `ignore`, `loop` and `constraint`, as shown in the following lemmas.

Lemma 5.1.6. Let S and S' be interactions with S' implementable and not overspecified. Let M be a set of messages. If $S \rightsquigarrow S'$, then $\texttt{ignore}(M,S) \rightsquigarrow \texttt{ignore}(M,S')$

Proof. Let I be a process such that $I \models \texttt{ignore}(M,S')$, then:

(a) $\exists\ \rho \in I.\ \rho \models_p \texttt{ignore}(M,S')$. Thus (by definition of positive satisfaction relation), $\exists\ \rho_1 \in \mathit{filter}(M)(\rho).\ \rho_1 \models_p S'$. Then $\rho_1 \not\models_n S'$ as S' is not overspecified, and by Lemma 5.1.3(1) $\rho_1 \models_p S$ and $\rho_1 \not\models_n S$. Finally, $\rho \models_p \texttt{ignore}(M,S)$

(b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{ignore}(M,S')$. Thus (by definition of negative satisfaction relation), $\forall \rho_1 \in \mathit{filter}(M)(\rho).\ \rho_1 \not\models_n S'$. Finally, by Lemma 5.1.3(2) $\rho_1 \not\models_n S$, and hence, $\rho \not\models_n \texttt{ignore}(M,S)$

From (a) and (b) we conclude that $I \models \texttt{ignore}(M,S)$, thus if $S \rightsquigarrow S'$, then $\texttt{ignore}(M,S) \rightsquigarrow \texttt{ignore}(M,S')$

□

Lemma 5.1.7. Let S and S' be interactions and S' be implementable and not overspecified. If $S \rightsquigarrow_p S'$, then $\forall\ m,n,\ m \leq n.\ \texttt{loop}(m,n,S) \rightsquigarrow_p \texttt{loop}(m,n,S')$

Proof. This is proven by induction on the pair (m,n), following the semantic rules that define the positive and negative satisfaction relation of `loop` (shown in Figure 3.4 and Figure 3.5, respectively).

`Base Case` $\Rightarrow$ `loop(0,0,-).` Let I be a process such that $I \models \texttt{loop}(0,0,S')$, then (a) $\exists\ \rho \in I.\ \rho \models_p \texttt{loop}(0,0,S')$. Thus (by definition of positive satisfaction relation) $\rho = \varepsilon$, and then $\rho \models_p \texttt{loop}(0,0,S)$. Also, (b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{loop}(0,0,S')$, and trivially (because $\texttt{loop}(0,0,\text{-})$ is never negative) $\rho \not\models_n \texttt{loop}(0,0,S)$. Finally, from (a) and (b) we conclude that $I \models \texttt{loop}(0,0,S)$, thus if $S \rightsquigarrow_p S'$, then $\texttt{loop}(0,0,S) \rightsquigarrow_p \texttt{loop}(0,0,S')$.

`Inductive Step` $\Rightarrow$ `loop(0,`n`+1,-).` The inductive hypothesis is: If $S \rightsquigarrow_p S'$, then $\texttt{loop}(0,n,S) \rightsquigarrow_p \texttt{loop}(0,n,S')$. Let I be a process such that $I \models \texttt{loop}(0,n+1,S')$, then (a) $\exists\ \rho \in I.\ \rho \models_p \texttt{loop}(0,n+1,S')$. Thus (by definition of positive satisfaction relation) $\rho = \varepsilon$ or $\rho \models_p \texttt{seq}(S',\texttt{loop}(0,n,S'))$. If $\rho = \varepsilon$ then $\rho \models_p \texttt{loop}(0,n+1,S)$. In the other case we use the transitivity property of refinement, in particular, combining the refinement properties ($\texttt{seq}_1$) and ($\texttt{seq}_2$) as follows: if $S \rightsquigarrow_p S'$ (hypothesis) and $\texttt{loop}(0,n,S) \rightsquigarrow_p \texttt{loop}(0,n,S')$ (inductive hypothesis) then $\texttt{seq}(S,\texttt{loop}(0,n,S)) \rightsquigarrow_p \texttt{seq}(S',\texttt{loop}(0,n,S'))$. Using this result, if $\rho \models_p \texttt{seq}(S',\texttt{loop}(0,n,S'))$ then $\rho \models_p \texttt{seq}(S,\texttt{loop}(0,n,S))$ (by Lemma 5.1.3(1) and the fact that S' is not overspecified) and finally $\rho \models_p \texttt{loop}(0,n+1,S)$. Also, (b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{loop}(0,n+1,S')$, implies (by definition of negative satisfaction relation) $\rho \not\models_n \texttt{seq}(S',\texttt{loop}(0,n,S'))$. Using the same reasoning as above with the Lemma 5.1.3(2), $\rho \not\models_n \texttt{seq}(S,\texttt{loop}(0,n,S))$, and then, $\rho \not\models_n \texttt{loop}(0,n+1,S)$.
Finally, from (a) and (b) we conclude that $I \models \texttt{loop}(0,n+1,S')$, thus if $S \rightsquigarrow_p S'$, then $\texttt{loop}(0,n+1,S) \rightsquigarrow_p \texttt{loop}(0,n+1,S')$.

`Inductive Step` $\Rightarrow$ `loop(`m`+1,`n`+1,-).` The inductive hypothesis is: If $S \rightsquigarrow_p S'$, then $\texttt{loop}(0,n,S) \rightsquigarrow_p \texttt{loop}(0,n,S')$. This case is proven exactly as the second case due to positive/negative satisfaction relation also uses `seq`. So, if $S \rightsquigarrow_p S'$, then $\texttt{loop}(m+1,n+1,S) \rightsquigarrow_p \texttt{loop}(m+1,n+1,S')$.

`Inductive Step` $\Rightarrow$ `loop(`m`,`∞`,-).` Let I be a process such that $I \models \texttt{loop}(m,\infty,S')$, then (a) $\exists\ \rho \in I.\ \rho \models_p \texttt{loop}(m,\infty,S')$. Thus (by definition of positive satisfaction relation) $\exists\ m \leq n.\ \rho \models_p \texttt{loop}(m,n,S')$ and this corresponds to one of the cases proven before. Also, (b) $\forall\ \rho \in I.\ \rho \not\models_n \texttt{loop}(m,\infty,S')$, thus (by definition of negative satisfaction relation) $\forall\ m \leq n.$

$\rho \not\models_n$ `loop`(m,n,S') and, one more time, this is one of the cases proven before. Finally, from (a) and (b) we conclude that $I \models$ `loop`(m,∞,S'), thus if $S \rightsquigarrow_p S$', then `loop`(m,∞,S) $\rightsquigarrow_p$ `loop`(m,∞,S').

□

Lemma 5.1.8. Let S and S' be interactions with S' implementable and not overspecified, let φ be a constraint. If $S \rightsquigarrow S$', then `constraint`(φ,S) $\rightsquigarrow$ `constraint`(φ,S').

Proof. Let I be a process such that $I \models$ `constraint`(φ,S'), then:

(a) $\exists\ \rho \in I.\ \rho \models_p$ `constraint`(φ,S'). Thus (by definition of positive satisfaction relation) $\rho \vdash \varphi \downarrow$ true $\wedge\ \rho \models_p S$', and thus $\rho \not\models_n S$' as S' is not overspecified. Then $\rho \models_p S$ by definition of $\rightsquigarrow$, and thus, $\rho \models_p$ `constraint`(φ,S)

(b) $\forall\ \rho \in I.\ \rho \not\models_n$ `constraint`(φ,S'). Thus (by definition of negative satisfaction relation) $\rho \not\vdash \varphi \downarrow$ false $\wedge\ \rho \not\models_n S$'. Finally, by Lemma 5.1.3(2) $\rho \not\models_n S$, and hence, $\rho \not\models_n$ `constraint`(φ,S).

From (a) and (b) we conclude that $I \models$ `constraint`(φ,S), thus if $S \rightsquigarrow S$', then `constraint`(φ,S) $\rightsquigarrow$ `constraint`(φ,S')

□

Finally, we can prove that if some constraint is added to an interaction then the interaction is refined, as shown by the following lemma.

Lemma 5.1.9. Let φ be a constraint, and S be an interaction.
$S \rightsquigarrow$ `constraint`(φ,S).

Proof. Let I be a process such that $I \models$ `constraint`(φ,S), then:

(a) $\exists\ \rho \in I.\ \rho \models_p$ `constraint`(φ,S). Thus (by definition of positive satisfaction relation) $\rho \vdash \varphi \downarrow$ true $\wedge\ \rho \models_p S$.

(b) $\forall\ \rho \in I.\ \rho \not\models_n$ `constraint`(φ,S). Thus (by definition of negative satisfaction relation) $\rho \not\vdash \varphi \downarrow$ false $\wedge\ \rho \not\models_n S$

From (a) and (b) we conclude that $I \models S$, thus $S \rightsquigarrow$ `constraint`(φ,S).

□

A summary of refinement rules is shown in Figure 5.2 where S, S_1, S_2, S', S_1' and S_2' are interactions and S', S_1' and S_2' are implementable and not overspecified.

Corollary 5.1.10. The set of refinement rules shown in Figure 5.2 is correct due to the proofs given in lemmas 5.1.5, 5.1.6, 5.1.7, 5.1.8 and 5.1.9.

□

$$(str_1)\frac{S_1 \rightsquigarrow_p S_1'}{\texttt{strict}(S_1, S_2) \rightsquigarrow_p \texttt{strict}(S_1', S_2)} \qquad (seq_1)\frac{S_1 \rightsquigarrow_p S_1'}{\texttt{seq}(S_1, S_2) \rightsquigarrow_p \texttt{seq}(S_1', S_2)}$$

$$(str_2)\frac{S_2 \rightsquigarrow S_2'}{\texttt{strict}(S_1, S_2) \rightsquigarrow \texttt{strict}(S_1, S_2')} \qquad (seq_2)\frac{S_2 \rightsquigarrow S_2'}{\texttt{seq}(S_1, S_2) \rightsquigarrow \texttt{seq}(S_1, S_2')}$$

$$(par_1)\frac{S_1 \rightsquigarrow_p S_1'}{\texttt{par}(S_1, S_2) \rightsquigarrow_p \texttt{par}(S_1', S_2)} \qquad (par_2)\frac{S_2 \rightsquigarrow_p S_2'}{\texttt{par}(S_1, S_2) \rightsquigarrow_p \texttt{par}(S_1, S_2')}$$

$$(alt_1)\frac{S_1 \rightsquigarrow S_1'}{\texttt{alt}(S_1, S_2) \rightsquigarrow \texttt{alt}(S_1', S_2)} \qquad (alt_2)\frac{S_2 \rightsquigarrow S_2'}{\texttt{alt}(S_1, S_2) \rightsquigarrow \texttt{alt}(S_1, S_2')}$$

$$(alt_3)\ \ \texttt{alt}(S_1, S_2) \rightsquigarrow S_1 \qquad (alt_4)\ \ \texttt{alt}(S_1, S_2) \rightsquigarrow S_2$$

$$(loop)\frac{S \rightsquigarrow_p S'}{\texttt{loop}(m, n, S) \rightsquigarrow_p \texttt{loop}(m, n, S')} \qquad (ass)\frac{S \rightsquigarrow S'}{\texttt{assert}(S) \rightsquigarrow \texttt{assert}(S')}$$

$$(ign)\frac{S \rightsquigarrow S'}{\texttt{ignore}(M, S) \rightsquigarrow \texttt{ignore}(M, S')} \qquad (neg_1)\frac{S \rightsquigarrow S'}{\texttt{neg}(S') \rightsquigarrow \texttt{neg}(S)}$$

$$(neg_2)\frac{S \rightsquigarrow_p S'}{\texttt{neg}(S) \rightsquigarrow_p \texttt{neg}(S')} \qquad (const_2)\ \ S \rightsquigarrow \texttt{constraint}(\varphi,S)$$

$$(const_1)\frac{S \rightsquigarrow S'}{\texttt{constraint}(\varphi, S) \rightsquigarrow \texttt{constraint}(\varphi, S')}$$

Figure 5.2: Compositional refinements of interactions

We proved the essential *soundness* property of our calculus for interaction refinement, that is, every statement that can be derived syntactically from the set of refinement rules is valid from a semantical point of view. This assertion is supported by the proofs given for each refinement rule, stating that syntactic rules are semantically correct. However, we did not prove the highly desirable reverse implication, called *completeness*, which means that every statement semantically correct can be derived syntactically from the refinement rules. Since this proof is not directly related with constrained interactions, it is left for future study.

5.2 Constraint Refinement

We are tempted to define the notion of constraint refinement in the same way as interaction refinement. A constraint refines another one if for every system-run the evaluation of the constraints satisfies the table in Figure 5.3,

formally written as follows.

Definition 5.2.1. (Constraint Refinement)
A constraint φ' refines a constraint φ , written $\varphi \rightsquigarrow \varphi$', if $\forall\ \rho$

- $\rho \vdash \varphi$'$\downarrow$ true $\Rightarrow$ $\rho \vdash \varphi \downarrow$ true
- $\rho \vdash \varphi$'$\downarrow$ undef $\Rightarrow$ $\rho \nvdash \varphi \downarrow$ false

□

Constraint	Refined Constraint
true	true, false or undefined
false	false
undefined	undefined or false

Figure 5.3: Constraint Refinement

We can also define a new refinement rule as those shown in Figure 5.2, but this time related to constraint refinement instead of interaction refinement.

Lemma 5.2.2. Let φ and φ' be constraints, and S be an interaction.
If $\varphi \rightsquigarrow \varphi$', then `constraint`($\varphi$,$S$) $\rightsquigarrow$ `constraint`(φ',S).

Proof. Let I be a process such that $I \models$ `constraint`(φ',S), then:

(a) $\exists\ \rho \in I.\ \rho \models_p$ `constraint`(φ',S). Thus (by definition of positive satisfaction relation) $\rho \vdash \varphi$' $\downarrow$ true $\wedge\ \rho \models_p S$. Then (by definition of `constraint` refinement) $\rho \vdash \varphi \downarrow$ true, and finally, $\rho \models_p$ `constraint`(φ,S)

(b) $\forall\ \rho \in I.\ \rho \not\models_n$ `constraint`(φ',S). Thus (by definition of negative satisfaction relation) one of the following cases occur

(i) $\rho \vdash \varphi$' $\downarrow$ true $\wedge\ \rho \not\models_n S$. Then (by definition of constraint refinement) $\rho \vdash \varphi \downarrow$ true, and finally $\rho \not\models_n$ `constraint`(φ,S)

(ii) $\rho \vdash \varphi$' $\downarrow$ undef $\wedge\ \rho \not\models_n S$. Then (by definition of constraint refinement) $\rho \nvdash \varphi \downarrow$ false and in consequence $\rho \not\models_n$ `constraint`(φ,S)

From (a) and (b) we conclude that $I \models$ `constraint`(φ,S), thus if $\varphi \rightsquigarrow \varphi$', then `constraint`($\varphi$,$S$) $\rightsquigarrow$ `constraint`(φ',S).

□

5.3 Related Work

Unfortunately, our definition of refinement does not ensure monotonicity of operators with respect to refinement, unless some restrictions are introduced (Section 5.1). This is the same problem as in the semantics of Cengarle and Knapp [CK04b].

The works from Störrle, and Haugen and Stølen define `alt` as the disjunction of behaviors specified by each operand. If this approach is taken in the negative satisfaction relation, the rule $\texttt{alt}(S_1, S_2) \rightsquigarrow S_i$ $i = 1{,}2$ is not valid anymore. This is proven as follows. Let I be a process such that $I \models S_1$. Then, by the definition of refinement $\forall\ \rho \in I.\ \ \rho \not\models_n S_1$. However, since the negative satisfaction relation for the alternative is a disjunction of the negative satisfaction of both operands, we must know what happens with S_2 in order to ensure that $\rho \not\models_n \texttt{alt}(S_1, S_2)$. This situation prevents refinement of an alternative by implementing only one of its operands.

Störrle, and Haugen and Stølen have different definitions of refinement which we will discusses in the following subsections. Each definition has specific consequences, so maybe a quite more powerful notion of refinement could be achieved someday. They also define refinement of time constraints. Time constraints are time intervals where the duration between two time points (events) must occur. Refinement means decreasing the intervals. Our notion of refinement allows specifying this kind of constraints in OCL/RT, but it is impossible to consider exactly the same idea for constraint refinement since we do not have the notion of interval but the existence of constraints which evaluate to true/false/undefined. In conclusion, there is no clue about how to define refinement of a constraint in the same terms of those works.

Grosu and Smolka [GS05] consider plain sequence diagrams represented as automata and a set of operators which allows automata composition, and in consequence they represent UML 2.0 Interactions. Refinement is defined in terms of language inclusion. A sequence diagram refines another one if the language accepted by the automaton of the first diagram is included in the language accepted by the automaton of the second diagram. Finally, they show that refinement is compositional with respect to sequential composition, alternative composition, parallel composition and star+ composition (looping).

Haugen and Stølen's Refinement

Haugen and Stølen [HKHS05] define refinement as a set of properties which ensures that negative system-runs must always remain negative in a refinement, while positive system-runs may remain positive or become negative;

inconclusive system-runs may go anywhere. This is summarized in Figure 5.4.

Trace	Refined Trace
positive	positive or negative
negative	negative
inconclusive	inconclusive, positive or negative

Figure 5.4: Haugen and Stølen's refinement

We can define this idea of refinement as follows.

Definition 5.3.1. (Haugen and Stølen's Refinement)
An interaction S' refines an interaction S, written $S \rightsquigarrow^{HS} S$', if

1. $\forall\ \rho.\ \rho \models_n S \Rightarrow \rho \models_n S$'
2. $\forall\ \rho.\ \rho \models_p S \Rightarrow \rho \models_p S' \vee \rho \models_n S$'

□

Note that there is no difference between positive and genuine positive system-runs. In contrast, we loose the idea of implementation of a system which seems natural.

They prove that the semantics is monotonic with respect to operators, except for `assert`. Besides, they define different kinds of refinements: supplementing, narrowing and detailing. The spirit behind all these concepts, however, makes them difficult to compare, since supplementing and narrowing address design evolution, whereas refinement is a tool for formal verification. In contrast with our definition of refinement, they allow inconclusive system-runs become positive after refinement. Also, they allow $S \rightsquigarrow^{HS} \texttt{neg}(S)$. After applying this refinement step, there can be no further refinement steps.

If we use this definition with our semantics, we realize that refinement rules shown in Figure 5.2 can be proven without the idea of positive refinement. As an example, consider the next refinement rule involving `strict`.

Lemma 5.3.2. Let S_1, S_2, and S_1' be interactions. If $S_1 \rightsquigarrow S_1$', then $\texttt{strict}(S_1,S_2) \rightsquigarrow \texttt{strict}(S_1',S_2)$.

Proof. Both properties in the definition of Haugen and Stølen's refinement are used.

(a) $\forall\ \rho.\ \rho \models_n \texttt{strict}(S_1,S_2)$ then (by negative relation definition) $\exists\ \rho_1,\rho_2.$ $(\rho_1,\rho_2) \in \rho; \wedge\ (\rho_1 \models_n S_1 \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2))$. Now, if $\rho_1 \models_n S_1$ then

(by $S \rightsquigarrow S$') $\rho_1 \models_n S_1$', else, if $\rho_1 \models_p S_1$ then (by $S \rightsquigarrow S$') $\rho_1 \models_p S_1$' or $\rho_1 \models_n S_1$'. Putting all the cases together, $\rho \models_n \texttt{strict}(S_1\text{'},S_2)$.

(b) $\forall\ \rho.\ \rho \models_p \texttt{strict}(S_1,S_2)$ then (by positive relation definition) $\exists\ \rho_1,\rho_2.\ (\rho_1,\rho_2) \in \rho;\ \wedge\ \rho_1 \models_p S_1 \wedge \rho_2 \models_p S_2$. Now, if $\rho_1 \models_p S_1$ then (by $S_1 \rightsquigarrow S_1$') $\rho_1 \models_p S_1$' (and consequently $\rho \models_p \texttt{strict}(S_1\text{'},S_2)$) or $\rho_1 \models_n S_1$' (and consequently $\rho \models_n \texttt{strict}(S_1\text{'},S_2)$).

From (a) and (b) we conclude that if $S_1 \rightsquigarrow S_1$', then $\texttt{strict}(S_1,S_2) \rightsquigarrow \texttt{strict}(S_1\text{'},S_2)$.

□

Störrle's Refinement

Störrle [Stö03a] defines an *elaboration* as the "relation between two interactions such that one of them contains more detail and less uncertainty than the other". In this sense, an *enrichment* is a relationship where one interaction has more valid system-runs than another while a *restriction* is a relationship where one interaction has more invalid system-runs than another. Then, a *refinement* is a relationship that is an *enrichment* and a *restriction* at the same time. Refinement tends to reduce uncertainty since the set of inconclusive system-runs for a given interaction is reduced in successive refinement steps becoming positive or negative. Notice that this approach seems similar to our definition of positive refinement since positive system-runs remain positive after refinement. This is summarized in Figure 5.5.

Trace	Refined Trace
positive	positive
negative	negative
inconclusive	inconclusive, positive or negative

Figure 5.5: Störrle's refinement

We can define this idea of refinement as follows.

Definition 5.3.3. (Störrle's Refinement)
An interaction S' refines an interaction S, written $S \rightsquigarrow^{S} S$', if

1. $\forall\ \rho.\ \rho \models_n S \Rightarrow \rho \models_n S$'
2. $\forall\ \rho.\ \rho \models_p S \Rightarrow \rho \models_p S$'

□

Störrle does not adopt a concrete semantics. Instead, he discusses different possible interpretations of the specification in [OMG05b]. Consequently, it superficially discusses the idea of refinement depending on the interpretation of `neg` and `assert`. He observes that only the interpretation of negation called "anything but the [valid] traces of S" and the interpretation of assertion as affirmation (the same idea we use) constitute elaborations in the sense of $\texttt{neg}(S) \rightsquigarrow^S S$ and $\texttt{assert}(S) \rightsquigarrow^S S$. None of the other interpretations constitute any other of the relationships defined. These two interpretations have in common that they completely remove contingency but in contrast there can be no further refinement steps afterwards. Within our interpretation of refinement this means that negative system-runs can be inconclusive after refinement.

Summary

In this chapter the notions of implementation and of interaction refinement were presented and also a set of refinement rules was introduced intended for formal verification. Refinement implies that the set of genuine positive system-runs (system-runs that are positive and not overspecified) cannot be enlarged, negative system-runs remain negative, inconclusive system-runs can be transformed into negative system-runs and at least one genuine positive system-run is kept. Also, the notion of constraint refinement was introduced in the same sense as interactions refinement.

Some alternative definitions of refinement have been analyzed. Nowadays there is no consensus in the community about the ideal definition of refinement. Each definition has its own benefits and problems, and sometimes its own application field, but there are no strong arguments to use one or another. Perhaps, depending on the system which might be constructed we can select the more appropriate approach.

Chapter 6

In Practice

The semantical background was not exhaustively analyzed from a practical point of view. This is mandatory for a fine-tuning and a careful utilization of the proposal. In this chapter, we show some examples of interactions with OCL/RT constraints and we study the consequences of this approach considering the abstract syntax generated, the use of positive/negative satisfaction relations in order to evaluate validity of system-runs, and the application of refinement. We also discuss the support for guards on operands and state invariants.

This chapter is structured as follows. Section 6.1 analyzes the use of UML 2.0 Interactions with OCL/RT constraints from a practical point of view. Section 6.2 shows the application of our approach in some examples taken from the literature detailing practical consequences of the results given in past sections, and Section 6.3 present some discussions intended for an enhancement of the proposal. Finally, Section 6.4 present related work.

6.1 OCL/RT in use

By definition of the semantics, *every constraint is local*, i.e., it only restricts the interaction within its scope. The scope of a constraint of the form `constraint`(φ,S) is the interaction S. We illustrate this with the next example. Consider the interaction in Figure 6.1 and add a constraint φ in the second operand. The interaction can be represented as `strict`(B_1,`constraint`(φ,B_2)) with message m as a basic interaction B_1 and message n as a basic interaction B_2.

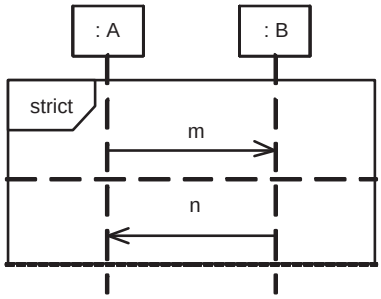


Figure 6.1: Interaction with a strict sequencing

If φ restricts the sending of message n to be within 3 units of time after the reception of message m, the constraint φ has to make reference to an event in the first operand as shown next.

```
context B
def: rcvM : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent)
                          and e.message.name = "m")->any()
def: sndN : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent)
                          and e.message.name = "n")->any()
constr: always((sndN.at - rcvM.at < 3)@sndN)
```

The satisfaction rule for strict sequencing states that $\rho \models_p \texttt{strict}(B_1,\texttt{constraint}(\varphi,B_2))$ if $\exists \rho_1, \rho_2$ such that $(\rho_1,\rho_2) \in \rho; \wedge\ \rho_1 \models_p S_1 \wedge \rho_2 \models_p \texttt{constraint}(\varphi,B_2)$. The second condition requires $(\rho_2, 0); \emptyset \vdash \varphi \downarrow$ true, but there is no information about ρ_1 when evaluating $\rho_2 \models_p\ B_2$.

This is because the scope of φ is only the interaction B_2. To solve the problem in this particular case, all the interaction must be constrained, i.e., $\texttt{constraint}(\varphi, \texttt{strict}(B_1,B_2))$.

In other situations, *a constraint needs to reference a concrete event within its scope*. As an example, suppose the interaction of Figure 6.2(a) only the first message m is constrained, neither the second nor the third; and also consider the example of Figure 6.2(b) where the constraint is specified using an explicit identifier.

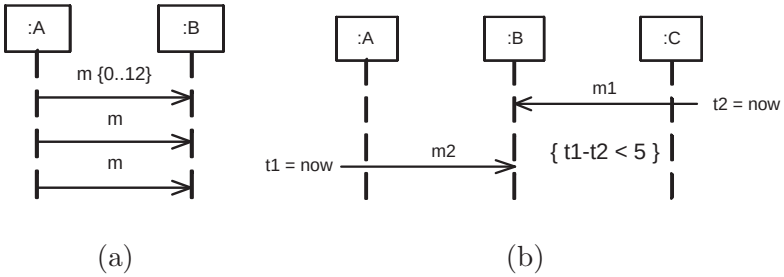


Figure 6.2: Example of constraints referencing concrete events

Within a basic interaction the total order of events in a lifeline can be used to identify any event. With the inclusion of `optional`, `ignore` and `loop` fragments there is no general strategy to do so. We can think of a general strategy consisting of looking for an event depending on the basic interaction it belongs to. If the event has some label identifying the basic interaction it belongs to, we can isolate the set of events of any basic interactions and then use the ordering within lifelines to identify each event. For this we can use the path in the syntax tree of each event and also its iteration number (in the case of loops).

This strategy is useful only with the appropriate semantical support. We want a formal semantics which verifies the positive/negative satisfaction of a given system-run considering now constraints which look for events by using the labels of the events. Since the events in the system-run are not equipped with labels, the semantics must dynamically assign labels to each event. Due to the fact that a system-run could satisfy an interaction in multiple ways, there will be multiple assignments for each event and only the right assignment will satisfy the constraints. In this context, a change in the semantics is not an easy business. Consequently, this subject is left for future study.

6.2 Examples

This section shows the application of our approach in some examples taken from the literature detailing practical consequences of the results given in past sections. In particular, we show how to use OCL/RT to write basic temporal constraints, we show how a system-run is satisfied by the using of the satisfaction relations, we compare our approach with the theory of Live Sequence Charts, and finally we analyze the meaning of refinement.

6.2.1 Basic Constraints

The first example shows how to specify OCL/RT constraints to express time and duration observations, and time and duration constraints as shown in [OMG05b]. Figure 6.3 presents the example (taken from [OMG05b]) consisting in one duration observation d (duration of the message *Code*), one time observation t (reception of the message *CardOut*), two duration constraints: duration of message *CardOut* and interval between the sending of *Code* and the reception of *OK*, and one time constraint: time point of the reception of *CardOut*.

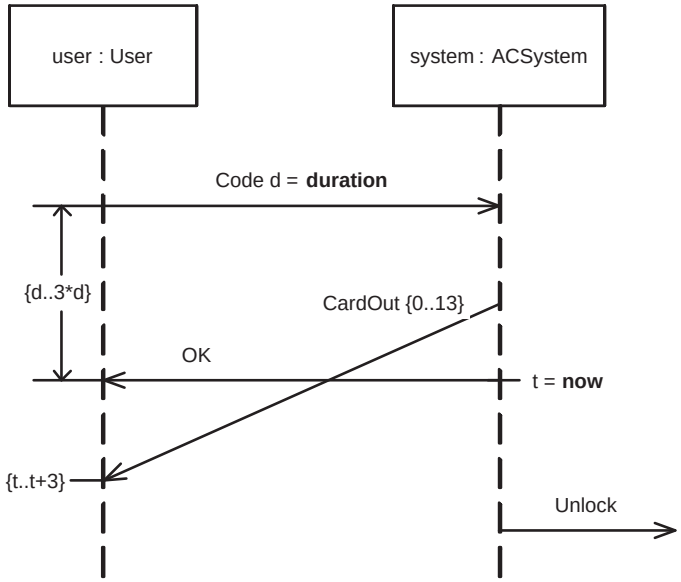


Figure 6.3: Basic interaction with basic time constraints

When representing a basic interaction we can use a more concrete, set-based notation like writing $\{\texttt{snd}(s,r,m) \leq \texttt{rcv}(s,r,m)\}$ instead of $[(\{e_1,e_2\},\{e_1 \leq e_2\}, \{e_1 \mapsto \texttt{snd}(s,r,m), e_2 \mapsto \texttt{rcv}(s,r,m)\})]$.

The abstract syntax of the example is just the definition of a basic interaction B enclosed by two OCL/RT constraints φ_1 and φ_2. First, the basic interaction is as follows.

$$\{\texttt{snd}(user,system,Code) \leq \texttt{rcv}(user,system,Code),$$
$$\texttt{snd}(system,user,CardOut) \leq \texttt{rcv}(system,user,CardOut),$$
$$\texttt{snd}(system,user,OK) \leq \texttt{rcv}(system,user,OK),$$
$$\texttt{snd}(user,system,Code) \leq \texttt{rcv}(system,user,OK) \leq$$
$$\texttt{rcv}(system,user,CardOut),$$
$$\texttt{rcv}(user,system,Code) \leq \texttt{snd}(system,user,CardOut) \leq$$
$$\texttt{snd}(system,user,OK) \leq \texttt{snd}(system,\text{-},Unlock)\}$$

The order is defined as follows: (a) the reception of a message occurs after sending it, and (b) events within a lifeline are ordered from top to bottom. The constraints of the example can be specified in OCL/RT as follows.

```
context User
def: t : Real =
       ACSystem.allInstances()->any().events->
          select(e | e.oclIsTypeOf(SndMsgEvent) and
                     e.message.name = "OK")->any().at
def: sndCardOut : Event =
       ACSystem.allInstances()->any().events->
          select(e | e.oclIsTypeOf(SndMsgEvent) and
                     e.message.name = "CardOut")->any()
def: rcvCardOut : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent) and
                          e.message.name = "CardOut")->any()

-- Duration of message CardOut and time of receiving CardOut
constr: always((rcvCardOut.at - sndCardOut.at <= 13)@rcvCardOut
               and (t <= rcvCardOut.at)@rcvCardOut
               and (rcvCardOut.at <= t + 3)@rcvCardOut)

context User
def: sndCode : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                          e.message.name = "Code")->any()
def: rcvCode : Event =
       ACSystem.allInstances()->any().events->
          select(e | e.oclIsTypeOf(RcvMsgEvent) and
                     e.message.name = "Code")->any()
def: rcvOK : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent) and
                          e.message.name = "OK")->any()
def: duration : Real = rcvCode.at - sndCode.at
```

```
-- Interval between sending Code and receiving OK
constr: always((rcvOK.at - sndCode.at <= 3 * duration)@rcvOK
               and (duration <= rcvOK.at - sndCode.at)@rcvOK)
```

One system-run ρ positive for the interaction is of the form (in order to simplify the following discussion some the value of *at* was added as a parameter in each event description)

$$(\omega_0, \{\ \texttt{snd}(user,system,Code,at_1),\ \texttt{rcv}(user,system,Code,at_2),\ \texttt{snd}(system,user,CardOut,at_3)\ \})$$
$$(\omega_1, \{\ \texttt{snd}(system,user,OK,at_4),\ \texttt{rcv}(system,user,OK,at_5)\ \})$$
$$(\omega_2, \{\ \texttt{rcv}(system,user,CardOut,at_6),\ \texttt{snd}(system,\text{-},Unlock,at_7)\})$$

A system-run positively satisfies the interaction S if the system-run: (a) satisfies the structure of the interaction (the events of the system-run correspond to the events within the interaction), (b) is well-timed (the occurrence time of the events satisfies the constraints), and (c) runs through valid system states (system states of the system-run satisfy the constraints which involve system instances).

The system-run ρ is valid for B depending on the assignments for at_i. In Figure 6.4, two different assignments for the system-run are provided. It is easy to show that for both assignments $\rho \models_p B$ since $lin_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}) \subseteq lin(B)$.

However, the whole interaction is $\texttt{constraint}(\varphi_1,\texttt{constraint}(c_2,B))$. While with the first assignment ρ positively satisfies the whole interaction, with the right one ρ is invalid (both constraints are not satisfied). We can notice that with the new semantics a system-run can be negative even if the interaction does not have any occurrence of `neg` or `assert`. A basic interaction cannot be overspecified, and any other system-run different from the one above is inconclusive.

Basic interactions cannot be refined by adding or deleting events. However, they can be refined by its time constraints (constraint refinement using Lemma 5.2.2). A constraint refinement for the interaction in the example can be done by restricting the constrained intervals, e.g., restricting the interval between the sending of *Code* and the reception of *OK* to {d..2*d}. Also in this case the first assignment of Figure 6.4 makes the system-run invalid. The basic interaction can also be refined by adding new constraints, e.g., {t..t+4} between the sending of *OK* and the sending of *Unlock*. In this case, only the first assignment makes the system-run valid for both interactions, the original and the refined one.

at_i	Assignment 1	Assignment 2
at_1	0	0
at_2	2	1
at_3	3	3
at_4	4	4
at_5	5	5
at_6	7	8
at_7	6	9

Figure 6.4: Two time assignments for the basic interaction of the example

6.2.2 Satisfying System Runs

The interaction in Figure 6.5 shows an operator receiving simple and emergency calls in parallel and redirecting them to the right destination.

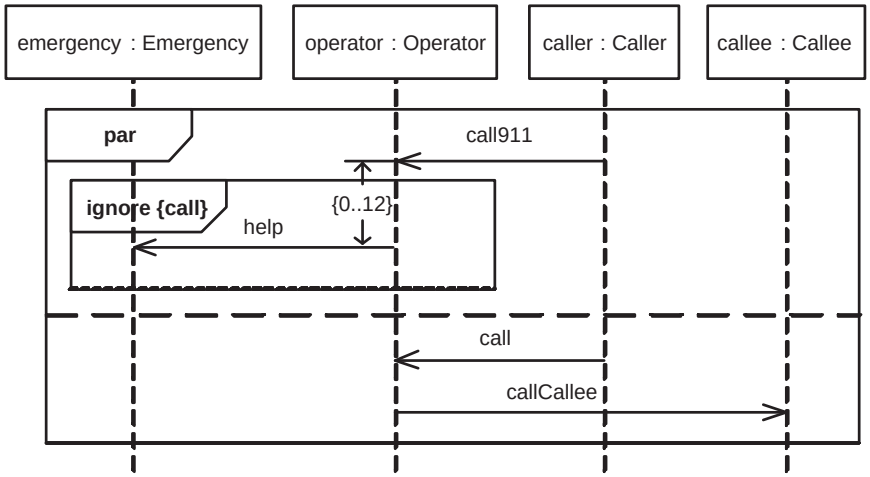


Figure 6.5: Simple and emergency calls

If there is an emergency call, the operator must send the emergency call to the emergency before 12 time units (constraint φ_1). Any other simple call must be ignored during the emergency call. There is another constraint (φ_2, not a real-time one), not shown in the diagram, which expresses that the operator makes simple calls only if there are lines available. These constraints can be expressed as follows.

```
context Operator
def: rcvCall911 : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent) and
                          e.message.name = "call911")->any()
def: sndHelp : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                          e.message.name = "help")->any()
constr: always((sndHelp.at - rcvCall911.at <= 12)@sndHelp)

context Operator
def: sndCallCallee : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                        e.message.name = "callCallee")->any()
constr: always((self.lines >= 0)@sndCallCallee)
```

For both constraints the modality `always` is used since the constraints must be satisfied during all system-runs. Also, the operator `@`η (with η an event) is used in order to verify each constraint at the state where some specific event occurred.

The interaction has three basic interactions corresponding to: message *call911* (denoted by B_1), messages within the ignore fragment (denoted by B_2) and messages within the second operand of the parallel fragment (denoted by B_3). The resulting abstract syntax is as follows.

`par(constraint(`φ_1, `seq(`B_1,`ignore(`call, B_2))), `constraint(`φ_2,B_3))

(ω_0, { `snd`(*caller*,*operator*,*call*,0), `snd`(*caller*,*operator*,*call911*,0),
`rcv`(*caller*,*operator*,*call911*,5) })
(ω_1, { `snd`(*operator*,*emergency*,*help*,13), `rcv`(*caller*,*operator*,*call*,6), τ,
`snd`(*caller*,*operator*,*call*,10), `rcv`(*operator*,*emergency*,*help*,15) })
(ω_2, { `snd`(*operator*,*callee*,*callCallee*,17), `rcv`(*caller*,*operator*,*call*,16),
`rcv`(*operator*,*callee*,*callCallee*,23), τ})

Figure 6.6: Example of a system-run

Take the system-run ρ shown in Figure 6.6 which shows some silent events. In order to simplify the notation, the value of *at* was added in each event description. Also, assume that the operator has always available lines.

With this information, we can prove that the system-run ρ positively satisfies the interaction S. To prove this, decompose the system-run ρ as $(\rho_4,\rho_3) \in \rho_{\|}$ and $(\rho_1,\rho_2) \in \rho_{4;_{\gtrdot}}$ such that

$$\rho_1 = (\omega_0, \{\ \texttt{snd}(caller,operator,call911),\ \texttt{rcv}(caller,operator,call911)\ \})\ (\omega_1, \{\ \tau\ \})$$

$$\rho_2 = (\omega_1, \{\ \texttt{snd}(operator,emergency,help),\ \texttt{snd}(caller,operator,call),\ \texttt{rcv}(operator,emergency,help)\ \})\ (\omega_2, \{\ \texttt{rcv}(caller,operator,call),\ \tau\})$$

$$\rho_3 = (\omega_0, \{\ \texttt{snd}(caller,operator,call)\ \})\ (\omega_1, \{\ \texttt{rcv}(caller,operator,call)\ \})\ (\omega_2, \{\ \texttt{snd}(operator,callee,callCallee),\ \texttt{rcv}(operator,callee,callCallee)\ \})$$

It is easy to show that $\rho_i \models_p B_i$ $i = 1..3$. With this information we can build the following proof using the positive fragment of the semantics. On one side, we have that

$$\dfrac{\dfrac{\dfrac{\exists \rho_2' \in filter(\{\text{call}\})(\rho_2)\, \rho_2' \models_p B_2}{\rho_2 \models_p \texttt{ignore}(\{\text{call}\}, B_2)} \quad \rho_1 \models_p B_1 \quad (\rho_1, \rho_2) \in \rho_{4;\gg}}{\rho_4 \models_p \texttt{seq}(B_1,\texttt{ignore}(\{\text{call}\}, B_2))} \quad (\rho_4, 0);\ \emptyset \vdash \varphi_1 \downarrow \text{true}}{\rho_4 \models_p \texttt{constraint}(\varphi_1,\texttt{seq}(B_1,\texttt{ignore}(\{\text{call}\}, B_2)))}$$

While, on the other side $\dfrac{\rho_3 \models_p B_3 \qquad (\rho_3, 0);\ \emptyset \vdash \varphi_2 \downarrow \text{true}}{\rho_3 \models_p \texttt{constraint}(\varphi_2, B_3)}$

Finally,

$$\dfrac{\begin{array}{c}(\rho_4, \rho_3) \in \rho_{\|} \\ \rho_4 \models_p \texttt{constraint}(\varphi_1,\texttt{seq}(B_1,\texttt{ignore}(\{\text{call}\}, B_2))) \\ \rho_3 \models_p \texttt{constraint}(\varphi_2, B_3)\end{array}}{\rho \models_p \texttt{par}(\texttt{constraint}(\varphi_1,\texttt{seq}(B_1,\texttt{ignore}(\{\text{call}\}, B_2))),\texttt{constraint}(\varphi_2, B_3))}$$

If any constraint evaluates to false the system-run does not positively satisfy the interaction but negatively as explained next.

A given system-run negatively satisfies an interaction if any constraint evaluates to false while the others evaluate to true in that system-run, even if it runs positively through the interaction lacking of constraints. In the example, if the operator sends the emergency call to the emergency after 12 time units or it does not have any line available, the system-run ρ negatively satisfies the interaction S. Next we show the negative satisfaction of ρ by making false the first constraint (by assigning more than 12 time units between the reception of *call911* and the sending of *help*).

$$\dfrac{\dfrac{\dfrac{\exists \rho_2' \in filter(\{\text{call}\})(\rho_2)\, \rho_2' \models_p B_2}{\rho_2 \models_p \text{ignore}(\{\text{call}\}, B_2)} \quad \rho_1 \models_p B_1\, (\rho_1, \rho_2) \in \rho_{4;\gg}}{\rho_4 \models_p \text{seq}(B_1,\text{ignore}(\{\text{call}\}, B_2))} \quad (\rho_4, 0);\ \emptyset \vdash \varphi_1 \downarrow \text{false}}{\rho_4 \models_n \text{constraint}(\varphi_1,\text{seq}(B_1,\text{ignore}(\{\text{call}\}, B_2)))}$$

Even if the system-run is positive for the second fragment of the interaction, we have that

$$\frac{\begin{array}{c}(\rho_4, \rho_3) \in \rho_{\|} \\ \rho_4 \models_n \text{constraint}(\varphi_1,\text{seq}(B_1,\text{ignore}(\{\text{call}\}, B_2))) \\ \rho_3 \models_p \text{constraint}(\varphi_2, B_3)\end{array}}{\rho \models_n \text{par}(\text{constraint}(\varphi_1,\text{seq}(B_1,\text{ignore}(\{\text{call}\}, B_2))),\text{constraint}(\varphi_2, B_3))}$$

We can also introduce negative behavior without constraints by adding a `neg`/`assert` fragment. Let us modify the example as shown in Figure 6.7. Now, after receiving an emergency call the only valid execution is to ask for help.

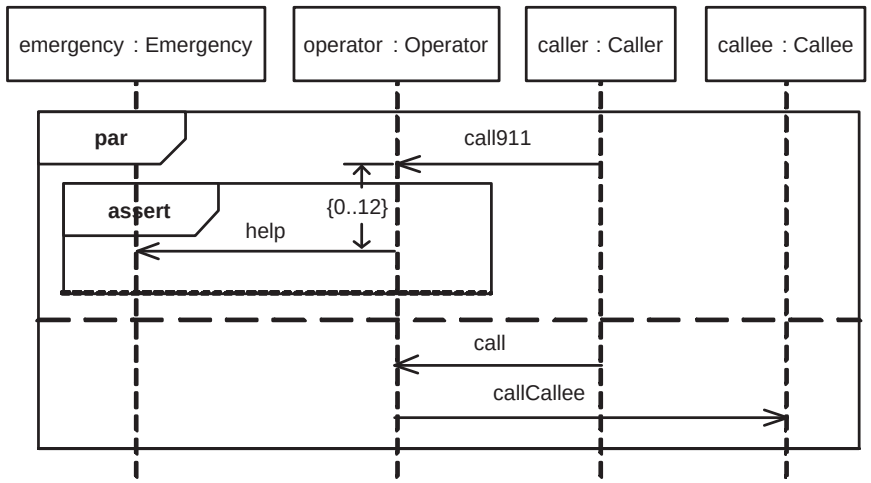


Figure 6.7: The example modified with an assertion

With this change the system-run ρ negatively satisfies the interaction due to the assert fragment could not be positively satisfied by the system-run ρ_2, as shown next.

$$\frac{\dfrac{\dfrac{\rho_2 \not\models_p B_2}{\rho_2 \models_n \text{assert}(B_2)} \qquad \rho_1 \models_p \; B_1\,(\rho_1, \rho_2) \in \rho_{4;\!\!\gg}}{\rho_4 \models_n \text{seq}(B_1,\text{assert}(B_2))} \qquad (\rho_4, 0);\ \emptyset \vdash \varphi_1 \downarrow \text{false}}{\rho_4 \models_n \text{constraint}(\varphi_1,\text{seq}(B_1,\text{assert}(B_2)))}$$

6.2.3 Hot and Cold Constraints

Damm and Damm and Harel [DH01] define an extension of Message Sequence Charts [IT96] called Live Sequence Charts (LSCs). LSCs distinguish between cold and hot conditions. Cold conditions *may* be true otherwise control moves out of the current block or chart. Hot conditions *must* be true otherwise the system aborts. This behavior can be represented with our approach. As an example we will use part of the "The NetPhone Example" from [HM02].

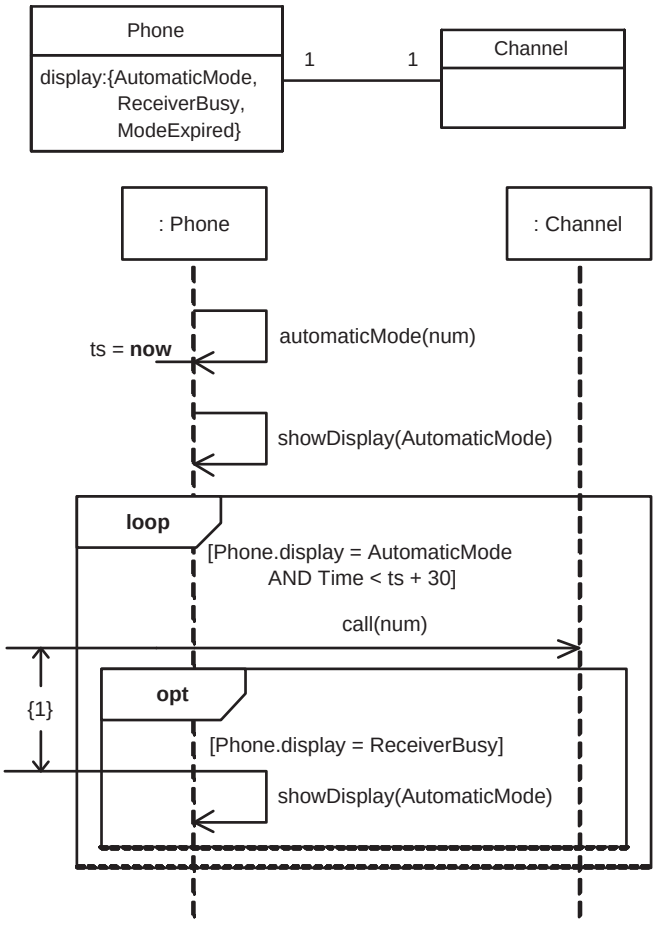


Figure 6.8: UML domain model and interaction for the NetPhone example in automatic mode

A phone in automatic mode "displays the message *Automatic Mode*, and enters an unbounded loop, which iterates for as long as the display shows

Automatic Mode, but no longer than 30 seconds after the mode was entered. This loop control is captured by the cold condition inside the loop, whose semantics prescribes exiting the current loop upon becoming false [...] and tries calling the desired number. It then waits one second for the communication protocol to be over and checks the message on the display (hot condition). If it says *Receiver Busy*, the display is set to show *Automatic Mode* again. After the loop, there is a cold time constraint." Figure 6.8 shows this behavior and a UML domain model for the example, considering only one phone and one channel.

The abstract syntax is shown next.

$$\texttt{constraint}(\varphi_1, \texttt{seq}(B_1, \texttt{loop}(0,\infty, \texttt{constraint}(\varphi_3, \texttt{seq}(B_2, \texttt{opt}(\texttt{constraint}(\varphi_2, B_3)))))))$$

There are two guards (φ_1 restricting the loop fragment and φ_2 restricting the optional fragment) and one constraint with an interval (φ_3). Also, there are three basic interactions: B_1 for the first two messages, B_2 for the *call* message and B_3 for the message *showDisplay* within the optional fragment.

Guards can be specified by OCL/RT constraints as we will discussed in Section 6.3.1. We will first specify the loop's guard (φ_1). However, there is no notion about when a guard has to be validated, because there is no special event for that (Live Sequence Charts have the notion of *location* for this purpose). We decide that a guard has to be evaluated when the first message within the fragment that it restricts occurs. In this case, the constraint must be validated each time *call* is sent.

```
context Phone
def: ts : Real =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent) and
                          e.message.name
                                 = "automaticMode")->any().at
def: sndCall : Set(Event) =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                          e.message.name = "call")
constr: always(sndCall->forall(e | (e.at < ts + 30 and
                       self.phone.display = AutomaticMode)@e))
```

The second guard φ_2 restricts the optional fragment. In this case the guard is defined locally for the optional fragment. Consequently, the only events a system-run must have at this point will be those of the message *showDisplay*. If the system-run is empty at this point the constraint is not verified since the optional fragment is defined as $\texttt{opt}(S) = \texttt{alt}(\texttt{skip}, S)$

```
context Phone
def: sndShow : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                          e.message.name = "showDisplay")->any()
constr: always((self.display = ReceiverBusy)@sndShow)
```

The interval constraint φ_3 is defined inside the loop right before the weak sequence. Due to this, the system-run up to this point during evaluation will only have one *call* message and possibly one *showDisplay* message (local definition of constraints). The constraint must be satisfied (hot condition) but it depends on the existence of the second message (restricted by the cold condition).

```
context Phone
def: sndCall : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                          e.message.name = "call")->any()
def: sndShow : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent) and
                          e.message.name = "showDisplay")->any()
constr: always((not sndShow.oclIsUndefined()) implies
                                  sndShow.at - sndCall.at = 1)
```

In summary, cold constraints are specified by guards in optional fragments (if the guard is not satisfied then the fragment is not executed) while hot constraints are specified by simple constraints (if the constraint is not satisfied then the system-run is negative). This analysis has been restricted to hot and cold constraints while Live Sequence Charts present some other constructs like cold/hot messages and locations that are not presented in UML 2.0, as discussed in [CKF05].

6.2.4 Refinement at Work

In this section we will work with a model of a simple train system [DLDvL05]. The system is composed of three classes: a train controller (controlling operations such as start, stop, open doors, and close doors), a train actuator/sensor (controlling the movement and emergency alarms), and passengers, as shown in Figure 6.9.

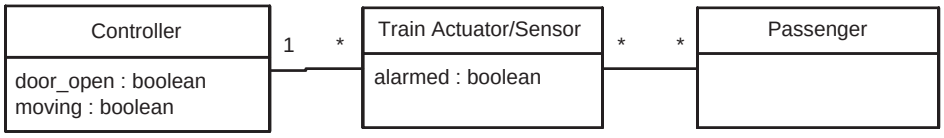


Figure 6.9: UML domain model for the train system

A safety goal requires train doors to remain closed while the train is moving, but if the passenger presses the alarm button, the controller must stop the train first and then open the doors in emergency (shown in Figure 6.10).

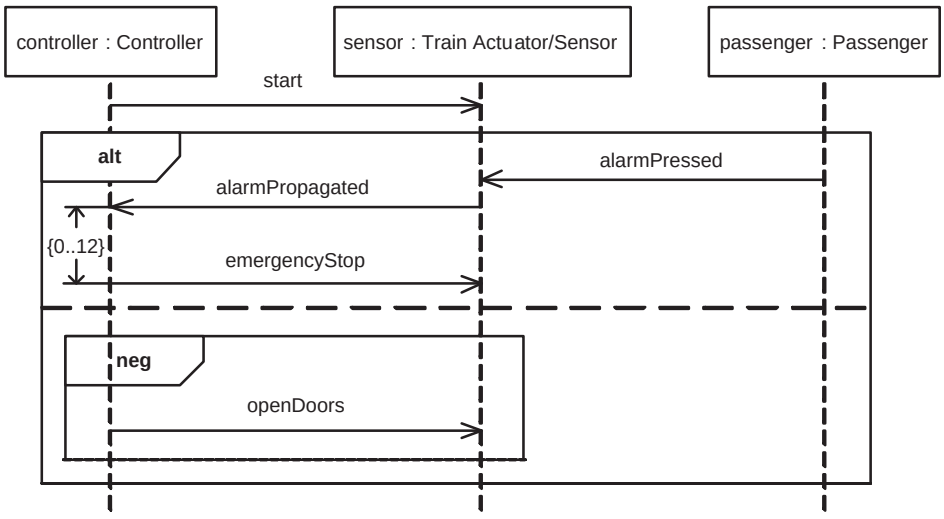


Figure 6.10: Interaction for emergency stop and door closing while moving

There is a time constraint between the reception of the message *alarmPropagated* and the sending of the message *emergencyStop*, specified as follows.

```
context Controller
def: rcvAlarmPropagated : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent)
                          and e.message.name
                               = "alarmPropagated")->any()
def: sndEmergencyStop : Event =
       events->select(e | e.oclIsTypeOf(SndMsgEvent)
                          and e.message.name
                               = "emergencyStop")->any()
constr: always((sndEmergencyStop.at -
                rcvAlarmPropagated.at <= 12)@sndEmergencyStop)
```

The abstract syntax of the interaction is as follows.

$$\texttt{seq}(B_1,\texttt{alt}(\texttt{constraint}(\varphi,B_2),\texttt{neg}(B_3)))$$

where B_1 is a basic interaction composed by the message *start*, B_2 is the basic interaction in the first argument of the alternative composition, and B_3 is the basic interaction composed by the message *openDoors*.

A possible refinement of the last interaction can be made by taking apart the positive behavior, this means that,

$$\texttt{seq}(B_1,\texttt{alt}(\texttt{constraint}(\varphi,B_2),\texttt{neg}(B_3))) \rightsquigarrow \texttt{seq}(B_1,\texttt{constraint}(\varphi,B_2))$$

$$(seq_2)\frac{(alt_3)\ \dfrac{\texttt{alt}(\texttt{constraint}(\varphi,B_2),\texttt{neg}(B_3))}{\texttt{alt}(\texttt{constraint}(\varphi,B_2),\texttt{neg}(B_3)) \rightsquigarrow \texttt{constraint}(\varphi,B_2)}}{\texttt{seq}(B_1,\texttt{alt}(\texttt{constraint}(\varphi,B_2),\texttt{neg}(B_3))) \rightsquigarrow \texttt{seq}(B_1,\texttt{constraint}(\varphi,B_2))}$$

A constraint refinement can be made by adding another constraint that restricts the set of positive system-runs allowed by the interaction. In the example we can restrict that after the reception of an emergency stop if an alarm was activated and the train was stopped then the doors must be opened, as follows.

```
context TrainActuator
def: rcvEmergencyStop : Event =
       events->select(e | e.oclIsTypeOf(RcvMsgEvent)
                          and e.message.name
                               = "emergencyStop")->any()
constr: always(rcvEmergencyStop.sender.moving = false
                and alarmed => self.openDoors())
```

Taking S as the last interaction, φ as the first constraint and φ' as the conjunction of the first and the second constraints, it is straightforward to show that $\varphi \rightsquigarrow \varphi$', and in consequence `constraint`(φ,S) $\rightsquigarrow$ `constraint`(φ',S).

6.3 Discussions

This section is devoted to the discussion of how constraints can be used to specify guards on operands and state invariants, and the impact on the semantics we defined.

6.3.1 Guards on Operands

According to the specification of UML 2.0 Interactions [OMG05b], every interaction operand could have an interaction constraint associated to it. The constraint must be a boolean expression evaluating to true in order to execute that operand. Every operand has a guard, when the guard is omitted, true is assumed.

Guards are represented in square brackets at the top of the operand over the leftmost lifeline, as shown in Figure 6.11. There are special guards, for example an *else* guard (commonly used in an alternative fragment) can be used within the last operand of some combined fragment representing "*the negation of the disjunction of all other guards in the enclosing combined fragment*" [OMG05b]. Also, loops have a mandatory guard representing the minimum and maximum number of iterations.

The semantics developed in previous sections allows specifying this guards without any change, due to the fact that every interaction can have an OCL/RT constraint, and OCL/RT is enough powerfull to represent any boolean guard. In fact, OCL/RT is quite more expressive than necessary.

Consider the interaction in Figure 6.11, where there exist some guarded interaction fragments. Its syntax is (`opt`(S) is used as `alt`(`skip`,S)):

$$\texttt{alt(constraint(}\varphi_1\texttt{,seq(loop(0,}\infty\texttt{,constraint(}\varphi_2\texttt{,}B_2\texttt{))),}$$
$$\texttt{constraint(not(}\varphi_2\texttt{),}B_2\texttt{)),}$$
$$\texttt{constraint(not(}\varphi_1\texttt{),opt(constraint(}\varphi_3\texttt{,}B_3\texttt{))))}$$

Notice that each guard is represented by an OCL/RT constraint and that an *else* guard is represented as the negation of the disjunction of all other guards in the enclosing combined fragment.

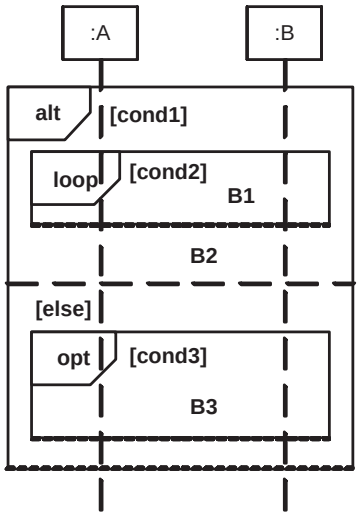


Figure 6.11: Guarded Interaction. *B*i's are basic interactions and *cond*i's are OCL/RT constraints

There is a special case involving `loop`. Every loop has a guard that may include the lower and upper number of iterations as well as a boolean expression. The first case is supported by the semantics in Section 3.2.2 and 3.2.3. The second case can be represented by a loop with an infinite upper bound where each of the iterations is constrained by the guard of the loop. In this way, a system-run positively satisfies the loop if the constraint evaluates to true in each of the iterations. But, what happens after the loop?.

- If the interaction continues after it, we have to assure that the constraint evaluates to false within the next step in the system-run. This can be done by evaluating the negation of the loop's guard outside the loop.
- If the system-run is empty after the loop, we have also to assure that the constraint evaluates to false. But, we do not know if an OCL/RT

constraint [trivially] evaluates to false in an empty system-run, because there is no explicit rule for an empty system-run and the result depends on the system state when the constraint is evaluated. Moreover, if an explicit rule is added maybe it could follow the same ideas as in classical logic, where a universal quantified formula evaluates to true in an empty universe. In conclusion, if we want to assume this we have to review the theory about OCL/RT, so future work is required.

6.3.2 State Invariants

The UML 2.0 specification defines a State Invariant as "*a runtime constraint on the participants of the interaction*". A State Invariant is considered an interaction fragment and it is placed on a Lifeline. The constraint is evaluated immediately prior to the execution of the next event in the same lifeline. If the constraint is true the system-run is a valid system-run; invalid otherwise.

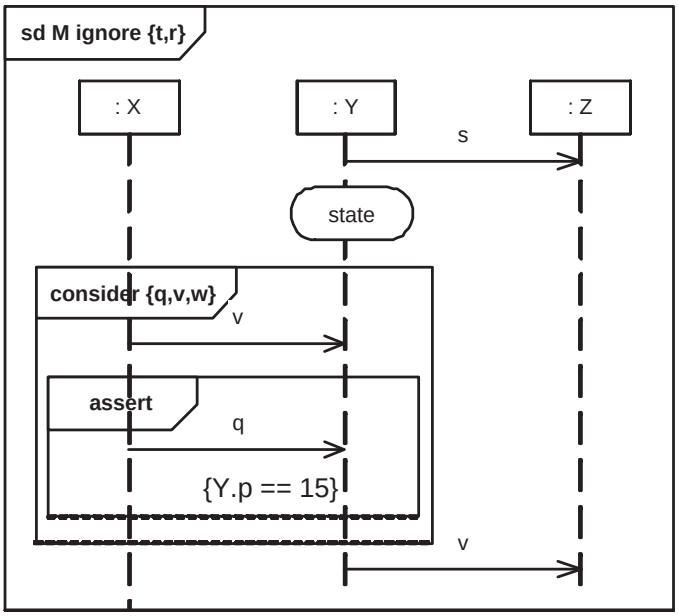


Figure 6.12: Interaction with state invariants

In Figure 6.12 (taken from [OMG05b]) we have an interaction M which ignores (message occurrences are irrelevant for this interaction) message t and r. The state invariant given as a state *state* will be evaluated at runtime directly prior to whatever event occurs on Y after *state*. This may be the

reception of v as specified within the consider fragment, or it may be an event that is specified to be insignificant by the filters. The assert fragment is nested in a consider fragment to mean that we expect a q message to occur once a v has occurred here. Any occurrences of messages other than v,w and q will be ignored in a test situation; thus the appearance of a w message after the v is an invalid system-run.

The constraint of a state invariant can be described by the actual internal state of the lifeline or by an external state machine. The second alternative could not be taken into consideration in this section since it is out of the scope of this thesis. Note that the first state invariant in Figure 6.12 associates the invariant to the state named *state* of some external state machine associated with the instance represented by the lifeline (we will consider it just like any other constraint), while the second one only checks the internal state of the instance.

As in the case of guards on operands (discussed in Section 6.3.1) the semantics developed in previous sections allows specifying state invariants without any change. In the example, the abstract syntax of the interaction can be embedded within a `constraint` fragment with the next constraints ($@\eta$ operator is used in order to check the constraints "state" and {Y.p==15} when the next event occurs).

```
context Y
def: ev(TYPE:Type, ID:String) : Event =
       events->select(e | e.oclIsTypeOf(TYPE) and
                          e.message.name = ID)->any()
def: msgY() : Sequence(Event) =
       events->select(e | e.sender = self or
                          e.receiver = self)->sorted()
def: next(EVE:Event) : Event =
       msgY()->at(msgY()->indexOf(EVE) + 1)

-- first state invariant that must be satisfied right before
-- the next event occurrence after sending message "s"
constr: always(
            self@next(ev(SndMsgEvent,"s")).oclInState(state))

-- second state invariant that must be satisfied right before
-- the next event occurrence after receiving message "q"
constr: always(self.p@next(ev(RcvMsgEvent,"q")) = 15)
```

In the last example we use a function *sorted* on collections of events that returns the events in a sequence and ordered according to their attribute **at**.

It is defined as follows.

```
Collection(Event)::sorted() : Sequence(Event) =
  iterate(iter; result : Sequence(Event) = Sequence{} |
    if result->isEmpty() or result->last().at <= iter.at then
      result->append(iterator)
    else
      let position : Integer = result->indexOf(
           result->select(item | item.at > iter.at)->first())
      in
        result->insertAt(position, iter)
    endif
```

6.4 Related Work

Complementing our research with the reviews made by Cengarle and Knapp [CK02], and Flake [Fla03] we can notice that there are several works that analyze different OCL temporal extensions. Most of them do not consider real-time constraints and in the ones that do, they are mostly defined in an informal way.

Some OCL extensions are closely related to state-based specifications [Fla03, FM04, MCM04]. These languages allow constraints in system models specified by state diagrams. However, the use of state diagrams is more used for intra-component than inter-component communications, so there are not closely related to our proposal.

Only the work made by Cavarra and Küster-Filipe [CKF05] analyzes an OCL extension to be used with interactions. They address the problem of synchronization and the specification of liveness properties (normally expressed at the level of state diagrams). Our work does not address the problem of synchronization but the use of OCL/RT allows specifying some liveness properties. In this sense, OCL/RT has some limitations, for example to state that some event will *eventually* be received, since this modality is not yet considered. In order to deal with liveness they shift the context of an interaction from one instance to the whole interaction or a specific interaction diagram. They also extend the language by adding some properties that apply to all objects to know in what interaction fragment an object is. OCL/RT keeps OCL without big changes, but knowing were an event is located could be very helpful in practice. Cavarra and Küster-Filipe do not focus on a formal semantics but on how the language can be enriched for expressing inter-object behavior. Moreover, the language and the formal semantics defined in [CKF04] are never related. We think that our work and

the work from Cavarra and Küster-Filipe are complementary, and a fusion of both can be made in order to investigate the relation between constrained interactions and timed state machines.

Finally, some basic constraint can be graphically specified in UML 2.0 Interactions, as defined in [OMG05b]. These kind of constraints are simple time and duration observations, and time and duration constraints. These kind of constraints are also presented in other similar languages like Message Sequence Charts and Live Sequence Charts. These constraints can be expressed using OCL/RT assigning to them a concrete meaning in the evaluation of a system-run. Besides, OCL/RT allows specifying more complex constraints involving also system instances, message events and global variables. Consequently our approach is one of the most powerful proposals in order to specify real-time constraints in UML 2.0 Interactions.

Summary

In this chapter we analyzed the practical application of the results presented in this work, intended for future improvements and careful utilization of them. First, we remark that by construction of the semantics, constraints are defined locally, this means, the constraint will only consider the interaction attached to it. We also remark that it could be useful to refer to concrete events in a system-run but this involves changing the semantics, which is not straightforward.

Some examples taken from the literature were shown. These examples involved the representation of UML 2.0 Interactions basic temporal constraints and hot and cold constraints from Live Sequence Charts. We also showed other examples detailing practical consequences of the results given in past sections.

Finally, there has been some discussion about the possibility of specifying guards on operands and state invariants. As a conclusion we can say that both concepts can be represented within our approach but some remarks must be taken into consideration. For example, there are some problems involving guards on loops.

Chapter 7

Conclusions and Future Work

In this work, a denotational semantics of an enhanced UML 2.0 Interactions language with OCL/RT constraints support was developed, and an extensive analysis of their properties from a formal and practical point of view was made. The main results and conclusions are summarized in Section 7.1. Future work is outlined in Section 7.2.

7.1 Summary and Conclusions

This work presents a true concurrency linear-time trace-based semantics of complete traces of UML 2.0 Interactions with OCL/RT constraints support. This semantics is developed from the fusion of UML 2.0 Interactions and OCL/RT existing semantics proposed by Cengarle and Knapp in [CK04b] and [CK02], respectively.

In this sense, the UML 2.0 Interactions abstract syntax was extended with a new operator `constraint`(φ,S) where S is an interaction and φ is an OLC/RT constraint. Not all the features introduced by the UML 2.0 specification were considered. Some of them, as `break` and `critical`, are left for future study. Also, the event-model for OCL/RT was modified to consider only send and receive events within a system-run. The denotational semantics was defined by the fusion of both semantic domains, by considering system-runs as a powerful representation for traces, and modifying the original UML 2.0 Interactions denotational semantics to support them.

The resulting denotational semantics evaluates the positivity/negativity of a system-run within an interaction, considering both interaction structure

and constraints satisfaction. A system-run ρ is positive for an interaction $\texttt{constraint}(\varphi,S)$ if the system-run positively satisfies the interaction S and also φ evaluates to true in ρ. If ρ negatively satisfies the interaction S (no matter what happens with φ) or φ evaluates to false in ρ (no matter what happens with S), the system-run ρ negatively satisfies the interaction $\texttt{constraint}(\varphi,S)$. A system-run can be both positive and negative. In this case, the interaction is called an *overspecified* interaction. If the semantics cannot decide whether the system-run is positive or negative for some interaction, the system-run is called *inconclusive*. Figure 7.1 resumes these results.

	$\rho \vdash \varphi \downarrow$ true	$\rho \vdash \varphi \downarrow$ false	$\rho \vdash \varphi \downarrow$ undef
$\rho \models_p S$	$\rho \models_p \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_i \texttt{constraint}(\varphi,S)$
$\rho \models_n S$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$
$\rho \models_i S$	$\rho \models_i \texttt{constraint}(\varphi,S)$	$\rho \models_n \texttt{constraint}(\varphi,S)$	$\rho \models_i \texttt{constraint}(\varphi,S)$

Figure 7.1: Satisfaction relations resume

The inclusion of the OCL/RT in interactions allows specifying a wide variety of constraints, including real-time ones, since the OCL/RT can predicate over system-runs with timed events. Since some kind of constraints are not relevant for the interactions (we are working exclusively with message events), OCL/RT is not used in its full potential. As an example, system-runs do not consider termination events for operations and in consequence post-conditions cannot be verified. Moreover, synchronization is not considered since there is no agreement about the moment when an operation or a signal starts its execution after the reception event is received. However, we can think of extending the event-model considered in a system-run in order to develop a powerful behavioral specification model.

It is easy to check that properties in the original semantics also hold in the new one. In particular, both forms of sequential composition are associative, and parallel and alternative composition are associative and commutative. Some elementary properties about the satisfaction relation on constraints and the negation of a constraint were proven. Also, it was proven that the new semantics is just a concretization of the original semantics, i.e., the new semantics recognizes more specific behavior than the previous one due to the incorporation of constraints.

The notions of implementation and refinement of interactions, intended for formal verification, were adapted to our semantic domain. Refinement implies that the set of genuine positive system-runs (system-runs that are posi-

tive and not overspecified) cannot be enlarged, negative system-runs remain negative, inconclusive system-runs can be transformed into negative system-runs and at least one genuine positive system-run is kept. Their properties were analyzed, and also the set of derivation rules was extended with `loop`, `ignore` and `constraint` as shown in Figure 7.2. Also, the notion of constraint refinement was introduced. The *soundness* property of the calculus for interaction refinement was proved and a *completeness* proof was left for future study.

$$(loop)\frac{S \rightsquigarrow_p S'}{\texttt{loop}(m,n,S) \rightsquigarrow_p \texttt{loop}(m,n,S')}$$

$$(ign)\frac{S \rightsquigarrow S'}{\texttt{ignore}(M,S) \rightsquigarrow \texttt{ignore}(M,S')}$$

$$(const_1)\frac{S \rightsquigarrow S'}{\texttt{constraint}(\varphi,S) \rightsquigarrow \texttt{constraint}(\varphi,S')}$$

$$(const_2)\quad S \rightsquigarrow \texttt{constraint}(\varphi,S)$$

$$(const_2)\frac{\varphi \rightsquigarrow \varphi'}{\texttt{constraint}(\varphi,S) \rightsquigarrow \texttt{constraint}(\varphi',S)}$$

Figure 7.2: Compositional refinements of interactions

We discussed how the semantics allows specifying guards on operands and state invariants without any modifications. This can be done except for a special case involving guarded loops, which requires the guard not to be satisfied after the loop (and this must be assumed if the system-run is empty after the loop). In this case some revision to the theory behind OCL/RT is needed. A discussion about if it is possible to use Event instead of Basic as the base case of the semantics was done. This is possible if Basic is finite since an infinite disjunction with a finitely generated abstract syntax cannot be represented. It was shown how to modify the semantics assuming this, generating a simpler semantic definition.

It was also shown how this theory works from a practical point of view. In this sense, some examples were introduced showing how to apply the approach to existing examples in the literature (basic UML 2.0 Interactions constraints, and Hot and Cold constraints). The examples also detailed practical consequences and served as a proof of concepts.

Finally, a detailed comparison between this work and related ones was made, in particular, those developed by Störrle, Haugen and Stølen, Cavarra and Küster-Filipe, and the former work from Cengarle and Knapp on which this

work is based. The most important conclusion is that our semantics is the only one with an expressive support of complex constraints in interactions, in particular real-time ones.

Although all the contributions are useful for a general understanding of the problem, a lot of work must be done before the complete inclusion of these ideas in model-driven and formal reasoning tools.

7.2 Future Work

The semantics should be enhanced including those interaction operators not yet included (`break` and `critical`). Other interaction constructors could also be considered, as message parameters, asynchronous messages, references, gates and continuations, among others. These extensions require the study of the relationship between message events and other kind of events, as operation and signal execution, as well as the study of the meaning of an event occurrence in a system model.

As we discussed in Section 6.3, the semantics of OCL/RT must be revised in order to support boolean guarded loops. OCL/RT could also be extended in order to specify state-based constraints suited for temporal logic reasoning, as those presented in other OCL extensions like those from Flake and Mueller [FM02], and Cavarra and Küster-Filipe [CKF05].

A set of refinement rules was introduced but the completeness proof was left for future study as well as the search for new refinement rules.

An important topic for further work is related to the connection between our approach and other development activities as intra-component communications and design by contract. For example, the transformation between constrained interactions and timed state machines (as those in [KGSB98, Krü00, WS00]) could be addressed. This enables model checking [Lun00, KMR02] with tool support (e.g., Kronos [VER99] and Uppal [UU06]).

The simplification of OCL/RT constraints could be an interesting subject of study, as well as the development of a front-end for the semantics. On the one hand, we could incorpore labels to events in order to identify them in a simple way, as discussed in Section 6.1. This leads to the reformulation of the semantics for a dynamic assignment of labels during system-run verification. On the other hand, the semantics could be incorporated into some well known modeling tool. In this sense, a transforming algorithm from the visual representation of an interaction to the abstract syntax could be developed. The semantics could also be specified using some proof assistant (e.g. Coq [INR04] and Isabelle [UM05]), integrating the modeling tool with the proof

assistant.

Finally, it could be of interest to investigate how the semantics can be applied and extended in other contexts as the behavioral specification of software product lines [SCLC06]

References

[BAL97] Hanêne Ben-Abdallah and Stefan Leue. Timing Constraints in Message Sequence Chart Specifications. In *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X*, volume 107 of *IFIP Conference Proceedings*, pages 91–106. Chapman & Hall, 1997.

[BCR00] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An ASM Semantics for UML Activity Diagrams. In *Proc. 8th Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2000.

[Bor05] Borland. Together Architect 2005. `http://www.borland.com/us/products/together/index.html`, 2005.

[Büc62] Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proc. Int. Congress on Logic, Method, and Philosophy of Science. 1960*, pages 1–12. Stanford University Press, 1962.

[CK01] María Victoria Cengarle and Alexander Knapp. A Formal Semantics for OCL 1.4. In *Proc. 4th Int. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 118–133. Springer-Verlag, 2001.

[CK02] María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Proc. 11th Int. Symp. Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 390–409. Springer-Verlag, 2002.

[CK04a] María Victoria Cengarle and Alexander Knapp. OCL 1.4/5 vs. 2.0 Expressions Formal Semantics and Expressiveness. In *Software and Systems Modeling*, volume 3, pages 9–30. Springer, 2004.

[CK04b] María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In Robert France Jan Jürjens, Eduardo Fernandez and Bernhard Rumpe, editors, *Proc. 3rd Intl. Workshop on Critical Systems Development with UML*, pages 85–99. Technische Universität München, 2004.

[CK05] María Victoria Cengarle and Alexander Knapp. Operational Semantics of UML 2.0 Interactions. TUM-Report TUM-I0505, Technische Universität München, 2005.

[CKF04] Alessandra Cavarra and Juliana Küster-Filipe. Formalizing Liveness-Enriched Sequence Diagrams Using ASMs. *Lecture Notes in Computer Science*, 3052:62–77, 2004.

[CKF05] Alessandra Cavarra and Juliana Küster-Filipe. Combining Sequence Diagrams and OCL for Liveness. *Electronic Notes in Theorical Computer Science*, 115:19–38, 2005.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DLDvL05] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating Annotated Behavior Models from End-User Scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[Dou04] Bruce Powel Douglass. *Real-Time UML: Advances in the UML for Real-Time Systems.* Addison Wesley Professional, 3rd edition, 2004.

[Fla03] Stephan Flake. Temporal OCL Extensions for Specification of Real-Time Constraints. *Workshop Specification and Validation of UML models for Real Time and Embedded Systems at UML 2003*, 2003.

[FM02] Stephan Flake and Wolfgang Mueller. An OCL Extension for Real-Time Constraints. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 150–171. Springer-Verlag, 2002.

[FM04] Stephan Flake and Wolfgang Mueller. Past- and Future-Oriented Temporal Time-Bounded Properties with OCL. In J.R. Cuellar and Z. Liu, editors, *Proc. 2nd Int. Conf. on Software Engineering and Formal Methods*, pages 154–163. IEEE Computer Society Press, 2004.

[GS05] Radu Grosu and Scott Smolka. Safety-Liveness Semantics for uml 2.0 sequence diagrams. In *Proc. 5th Intl. Conference on Application of Concurrency to System Design*, pages 6–14. IEEE Computer Society, 2005.

[GZK03] Martin Gogolla, Paul Ziemann, and Sabine Kuske. Towards an Integrated Graph Based Semantics for UML. In *Electronic Notes in Theoretical Computer Science*, volume 72, pages 1–16, 2003.

[Ham06] Youcef Hammal. Branching Time Semantics for UML 2.0 Sequence Diagrams. In *Proc. Formal Techniques for Networked and Distributed Systems*, volume 4229 of *Lecture Notes in Computer Science*, pages 259–274. Springler, 2006.

[Hau04] Øystein Haugen. Comparing UML 2.0 Interactions and MSC-2000. In Daniel Amyot and Alan Williams, editors, *Proc. SDL and MSC Fourth International Workshop*, volume 3319 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, 2004.

[HKB04] Rolf Hennicker, Alexander Knapp, and Huber Baumeister. Semantics of OCL Operation Specifications. In Th. Baar, T. Clark, R. France, R. Hähnle, H. Hußmann, and P. Schmitt, editors, *Proc. OCL 2.0 - Industry standard or scientific playground?*, volume 102 of *Electronic Notes in Theoretical Computer Science*, pages 111–132. Elsevier, 2004.

[HKHS05] Øystein Haugen, Ragnhild Kobro, Knut Husa, and Ketil Stølen. Why Timed Sequence Diagrams Require Three-Event Semantics. In Stefan Leue and Tarja Systä, editors, *Post-proc. of Dagstuhl seminar, Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2005.

[HM02] David Harel and Rami Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 193–202. IEEE Computer Society, 2002.

[HS03] Øystein Haugen and Ketil Stølen. STAIRS - Steps To Analyze Interactions with Refinement Semantics. In Jon Whittle Perdita Stevens and Grady Booch, editors, *Proc. 6th Int. Conf. UML 2003 - The Unified Modeling Language, Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 388–402. Springer-Verlag, 2003.

[HT03] David Harel and P. S. Thiagarajan. Message Sequence Charts. pages 77–105. Kluwer Academic Publishers, 2003.

[HvdZ03] Jozef Hooman and Mark van der Zwaag. A Semantics of Communicating Reactive Objects with Timing. *Proc. Specification and Validation of UML models for Real Time and Embedded Systems*, 2003.

[IL05] I-Logix. Rhapsody Developer Edition. `http://www.ilogix.com/sublevel.aspx?id=53`, 2005.

[IMU05] IBM, Technische Universität München, and Queen's University. Uml 2.0 Semantic Project, 2005.

[INR04] LOGICAL Project INRIA. Coq 8.0. `http://coq.inria.fr/`, 2004.

[IR04] IBM-Rational. Rational Rose Real Time. `http://www-306.ibm.com/software/awdtools/developer/technical/`, 2004.

[IR05] IBM-Rational. Rational Software Architect 6. `http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html`, 2005.

[IT96] ITU-TS. Itu-ts recommendation z.120: Message Sequence Charts (MSC), 1996.

[JEJ04] Yan Jin, Robert Esser, and Jörn Janneck. A Method for Describing the Syntax and Semantics of UML Statecharts. In *Software and Systems Modeling*, volume 3, pages 150–163, 2004.

[Jür02a] Jan Jürjens. Formal Semantics for Interacting UML Subsystems. *Formal Methods for Open Object-Based Distributed Systems*, 2002.

[Jür02b] Jan Jürjens. A UML Statecharts Semantics with Message Passing. *Symposium of Applied Computing, ACM*, 2002.

[KC04] Kennedy-Carter. iUML. `http://www.kc.com/`, 2004.

[Ken02] Stuart Kent. Model Driven Engineering. In *Proc. IFM 2002*, volume 2335 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[KFdB+04] Marcel Kyas, Harald Fecher, Franks S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML Models and OCL Constraints in PVS. In *Semantic Foundations of Engineering Design Languages*, Electronic Notes in Theoretical Computer Science, 2004.

[KGSB98] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In Franz Rammig, editor, *Proc. IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1998.

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking - Timed UML State Machines and Collaborations. In W. Damm and E. Olderog, editors, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springler, 2002.

[Kna99] Alexander Knapp. A Formal Semantics for UML Interactions. In Robert France and Bernhard Rumpe, editors, *Proc. UML 99: The Unified Modeling Language - Beyond the Standard, Second International Conference*, volume 1723 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 1999.

[Krü00] Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[KS01] Jochen Malte Küster and Joachim Stroop. Consistent Design of Embedded Real-Time Systems with UML-RT. In *Proc. 4th Int. Symposium on Object-Oriented Real-Time Distributed Computing*, pages 31–40. IEEE Computer Society, 2001.

[KW00] Anneke Kleppe and Jos Warmer. Extending OCL to include Actions. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conf. UML 2000*, volume 1939 of *Lecture Notes in Computer Science*, pages 440–450. Springer, 2000.

[Kwo00] Gihwon Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statecharts. In A. Evans, S. Kent, and B. Selic, editors, *Proc. International Conference on the Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 528–540. Springler, 2000.

[LKK+02] A. Letichevskii, Yu. Kapitonova, V. Kotlyarov, A. Letichevskii, and V. Volkov. Semantics of Timed Message Sequence Charts. In *Cybernetics and Systems Analysis*, volume 38, pages 475–484. Springer, 2002.

[LL00] Xuandong Li and Johan Lilius. Checking Compositions of UML Sequence Diagrams for Timing Inconsistency. In *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 154–161. IEEE Computer Society, 2000.

[LLH04] Xiaoshan Li, Zhiming Liu, and Jifeng He. A Formal Semantics of UML Sequence Diagram. In *Proc. 15th Australian Software Engineering Conference*, pages 168–177. IEEE Computer Society, 2004.

[LS06] Mass Soldal Lund and Ketil Stølen. A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. In *Proc. 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2006.

[Lun00] Carlos Luna. Especificación y Análisis de Sistemas de Tiempo Real en Teoría de Tipos. Caso de Estudio : The Railroad Crossing Example. Number RT 00-01, 2000.

[MCF03] Stephen Mellor, Anthony Clark, and Takao Futagami. Model-Driven Development. In *IEEE Software*, volume 20, pages 14–18, 2003.

[MCM04] Rafael Marcano, Samuel Colin, and Georges Mariano. A Formal Framework for UML Modelling with Timed Constraints: Application to Railway Control Systems. *Workshop of Specification and Validation of UML models for Real Time and Embedded Systems, UML 2004*, 2004.

[Mic03] Microsoft. Visio for Enterprise Architects. `http://www.microsoft.com/office/visio/prodinfo/default.mspx`, 2003.

[Mil80] Robin Milner. A Calculus of Communicating Systems. volume 92 of *Lecture Notes in Computer Science*. Springler, 1980.

[OMG03] OMG. Model-driven architecture. Guide v1.0.1, Object Management Group, 2003.

[OMG05a] OMG. UML 2.0 OCL. Adopted Specification ptc/05-06-06, Object Management Group, 2005.

[OMG05b] OMG. UML 2.0 Superstructure. Formal Specification formal/05-07-04, Object Management Group, 2005.

[OMO06] OMONDO. EclipseUML Studio. `http://www.eclipsedownload.com/product.html`, 2006.

[Pra86] Vaughan Pratt. Modelling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[RCA01] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 171–186. Springler, 2001.

[Rod00] Roberto Rodrigues. Formalising UML Activity Diagrams using Finite State Processes. 2000.

[Sch06] Douglas Schmidt. Model-Driven Engineering. In *Computer*, volume 39, pages 25–31, 2006.

[SCLC06] Nora Szasz, Daniel Calegari, Carlos Luna, and María Victoria Cengarle. Extensiones de Modelos UML para el Diseño el Comportamiento de Sistemas de Tiempo Real y Líneas de Producción. Proyecto PDT de Investigación Fundamental en Áreas Básicas, 2006.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. In *IEEE Software*, volume 20, pages 19–25, 2003.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[Sof05a] Artisan Software. Real-time Studio Professional. `http://www.artisansw.com/products/professional_overview.asp`, 2005.

[Sof05b] Tri-Pacific Software. Rapid RMA. `http://www.tripac.com/html/prod-toc.html`, 2005.

[SR98] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical Report, IBM Technical library, 1998.

[Ste02] Perdita Stevens. On the Interpretation of Binary Associations in the Unified Modelling Language. In *Software and Systems Modeling*, volume 1, pages 68–79, 2002.

[Stö03a] Harald Störrle. Assert, Negate and Refinement in UML-2 Interactions. *Intl. Ws. Critical Systems Development with UML, at UML'03*, 2003.

[Stö03b] Harald Störrle. Semantics of Interactions in UML 2.0. *Intl. Ws. Visual Languages and Formal Methods, at HCC'03*, 2003.

[Stö04a] Harald Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. 2004.

[Stö04b] Harald Störrle. Semantics of UML 2.0 Activities. In *Proc. Symposium on Visual Languages - Human Centric Computing*, pages 235–242, 2004.

[Stö04c] Harald Störrle. Trace Semantics of Interactions in UML 2.0. Technical Report TR 0403, Technische Universität München, 2004.

[Tig99] Tigris. Argo/UML. `http://argouml.tigris.org/`, 1999.

[UM05] Cambridge University and Technische Universität München. Isabelle. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`, 2005.

[UU06] Uppsala University and Aalborg University. Uppaal 3.6. `http://www.uppaal.com/`, 2006.

[Var02] Daniel Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2002.

[vdB02] Michael von der Beeck. A Structured Operational Semantics for uml-Statecharts. In *Software and System Modeling*, volume 1, pages 130–141, 2002.

[VER99] VERIMAG. Kronos 2.4. `http://www-verimag.imag.fr/TEMPORISE/kronos/index-english.html`, 1999.

[Vig04] Andrés Vignaga. Una Semántica Formal de Primitivas de Modificación de Estados de Sistemas Orientados a Objetos. Tesis de Maestría RT 04-01, InCo-PEDECIBA, 2004.

[WS00] Jon Whittle and Johann Schumann. Generating Statechart Designs from Scenarios. In *Proc. 22nd Int. Conference on on Software Engineering*, pages 314–323, 2000.