

***RoboCert* Reference Manual**

Version 0.1— draft of December 2, 2022

Matt Windsor

WWW.CS.YORK.AC.UK/ROBOSTAR

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

	Introduction	ix
0.1	How to read this manual	ix
0.2	Running examples	x

I	Core Language	
1	Core: metamodel	5
1.1	Introduction	5
1.1.1	Example	5
1.1.2	How to read the rest of this chapter	6
1.2	Top-level	6
1.2.1	CertPackage	6
1.2.2	Group ^a	7
1.2.3	ConstAssignment	7
1.3	Targets	8
1.3.1	ComponentTarget ^a	8
1.3.2	CollectionTarget ^a	8
1.4	Values	9
1.4.1	Use of Expression ^{rc}	9
1.4.2	ValueSpecification ^a	9
1.5	Assertions	10
1.5.1	AssertionGroup	10
1.5.2	Assertion and Property ^a	10
1.5.3	CoreProperty	11
1.5.4	SemanticModel	11

2	Core: textual syntax	13
2.1	Conventions	13
2.1.1	Terminals	13
2.2	Top-level	13
2.2.1	Group [□]	13
2.2.2	ConstAssignment	14
2.3	Targets	14
2.4	Values	14
2.5	Assertions	15
3	Core: well-formedness	17
3.1	Introduction	17
3.2	General conditions	17
3.3	Top-level	18
3.3.1	CertPackage (§ 1.2.1)	18
3.3.2	Group [□] (§ 1.2.2): CG	18
3.3.3	SpecificationGroup: CGs	18
3.3.4	ConstAssignment (§ 1.2.3): CC	19
3.4	Targets	19
3.5	Values	19
3.5.1	ValueSpecification [□] (§ 1.4.2): CV	19
3.5.2	ExpressionValueSpecification: CVe	19
3.5.3	WildcardValueSpecification: CVw	20
3.6	Assertions	20
3.6.1	Property [□]	20
3.6.2	CoreProperty (§ 1.5.3): CPc	20

II

Sequence Notation

4	Sequences: metamodel	25
4.1	Introduction	25
4.1.1	Naming and UML compatibility	25
4.1.2	Ordering and timing	25
4.1.3	Temperature	26
4.2	Interactions	26
4.2.1	Actor [□]	27
4.2.2	Interaction	27
4.3	Interaction fragments	29
4.3.1	OccurrenceFragment	30
4.3.2	BlockFragment [□]	30
4.3.3	DeadlineFragment	30
4.3.4	LoopFragment	31
4.3.5	OptFragment	31
4.3.6	UntilFragment	32
4.3.7	DiscreteBound	34

4.3.8	BranchFragment [□]	34
4.3.9	AltFragment	35
4.3.10	XAltFragment	35
4.3.11	ParFragment	36
4.3.12	InteractionOperand	37
4.3.13	Guard [□]	37
4.4	Occurrences	37
4.4.1	MessageOccurrence	38
4.4.2	WaitOccurrence	39
4.4.3	DeadlockOccurrence	39
4.5	Messages	40
4.5.1	MessageSet [□]	40
4.5.2	NamedMessageSet	40
4.5.3	Message	41
4.5.4	MessageTopic [□]	41
4.6	Assertions	42
4.6.1	SequenceProperty	42
5	Sequences: well-formedness	43
5.1	Sequences	43
5.1.1	Actor [□] (§ 4.2.1)	43
5.1.2	TargetActor	43
5.1.3	ComponentActor: SAC	43
5.1.4	World	43
5.1.5	Interaction (§ 4.2.2)	43
5.2	Fragments	44
5.2.1	InteractionFragment [□] (§ 4.3)	44
5.2.2	OccurrenceFragment (§ 4.3.1)	44
5.2.3	CombinedFragment [□]	44
5.2.4	BlockFragment [□] (§ 4.3.2): SBl	44
5.2.5	DeadlineFragment (§ 4.3.3): SD	44
5.2.6	LoopFragment (§ 4.3.4)	45
5.2.7	OptFragment (§ 4.3.5)	45
5.2.8	UntilFragment (§ 4.3.6: SU)	45
5.2.9	DiscreteBound (§ 4.3.7): SDb	45
5.2.10	BranchFragment [□] (§ 4.3.8): SBr	46
5.2.11	AltFragment (§ 4.3.9)	46
5.2.12	XAltFragment (§ 4.3.10)	46
5.2.13	ParFragment (§ 4.3.11)	46
5.2.14	InteractionOperand (§ 4.3.12)	46
5.2.15	Guard [□] (§ 4.3.13)	46
5.2.16	EmptyGuard	46
5.2.17	ExprGuard: SGe	46
5.2.18	ElseGuard	47
5.3	Occurrences	47
5.3.1	Occurrence [□] (§ 4.4)	47
5.3.2	MessageOccurrence (§ 4.4.1)	47
5.3.3	LifelineOccurrence: SLo	47
5.3.4	WaitOccurrence (§ 4.4.2): SW	47

5.3.5	DeadlockOccurrence (§ 4.4.3)	47
5.4	Messages	47
5.4.1	MessageSet ^a (§ 4.5.1)	47
5.4.2	NamedMessageSet (§ 4.5.2)	48
5.4.3	Message (§ 4.5.3)	48
5.4.4	MessageTopic ^a (§ 4.5.4)	49
5.4.5	EventTopic	49
5.4.6	OperationTopic	49
5.5	Assertions	49
6	Sequences: textual syntax	51
6.1	Interactions	51
6.1.1	Actor ^a	51
6.1.2	Interaction	51
6.2	Fragments	51
6.2.1	BlockFragment ^a	52
6.2.2	DeadlineFragment	52
6.2.3	LoopFragment	52
6.2.4	OptFragment	52
6.2.5	UntilFragment	52
6.2.6	BranchFragment ^a	52
6.2.7	AltFragment	52
6.2.8	XAltFragment	52
6.2.9	ParFragment	53
6.2.10	InteractionOperand	53
6.2.11	Guard ^a	53
6.3	Occurrences	53
6.3.1	MessageOccurrence	53
6.3.2	WaitOccurrence	53
6.3.3	DeadlockOccurrence	53
6.4	Messages	54
6.4.1	MessageSet ^a	54
6.4.2	NamedMessageSet	54
6.4.3	Message	54
6.4.4	MessageTopic ^a	54
6.5	Assertions	55

III

Low-Level Language Interoperability

7	CSP_M	61
7.1	Metamodel	61
7.1.1	CSPGroup	61
7.1.2	CSPProperty	61
7.2	Well-formedness conditions	61
7.2.1	CSPGroup (§ 7.1.1)	62
7.2.2	CSPProperty (§ 7.1.2)	62

8	Introduction	65
8.1	How to read these chapters	65
9	General Definitions	67
9.1	Core language	67
9.1.1	Values (§ 1.4)	67
9.2	Sequence notation	67
9.2.1	Interactions (§ 4.2.2)	68
9.2.2	Fragments (§ 4.3)	68
9.2.3	Occurrences (§ 4.4)	69
9.2.4	Messages (§ 4.5)	70
10	Timed Semantics: <i>tock</i>-CSP	71
10.1	Relationship to the generator	71
10.2	Note to the reader	71
10.3	Dependencies on the <i>RoboChart</i> semantics	72
10.4	Core language	72
10.4.1	Top-level (§ 1.2)	72
10.4.2	Targets (§ 1.3)	72
10.4.3	Values (§ 1.4)	72
10.4.4	Assertions (§ 1.5)	74
10.5	Sequence notation	75
10.5.1	Assertions (§ 4.6)	75
10.5.2	Sequences (§ 4.2)	75
10.5.3	Fragments (§ 4.3)	77
10.5.4	Occurrences (§ 4.4)	78
10.5.5	Messages (§ 4.5)	79
10.5.6	The until process	83
10.5.7	Memory	83
A	Language changelog	85
A.1	This draft	85
A.2	Version 0.1 (2022-05-20)	85
	Credits	87
	Bibliography	89

Introduction

RoboCert is a language for expressing properties of *RoboStar*-family models (*RoboChart*, *RoboSim*, and so on). It provides graphical and textual notations that enable roboticists to express both specifications over, and examples of, robot behaviour. By specifying their properties in *RoboCert*, users benefit from automated checking of those properties through the usual *RoboStar* formal reasoning stack (*tock*-CSP, PRISM, Isabelle/UTP, and more).

Our design aims for *RoboCert* are as follows. First, its notations should expose as many of the characteristic features of *RoboStar* notations (and robotics in general) as possible: platforms, events, operations, timing, probability, and so on. Second, graphical notations should be familiar to practitioners by adopting conventions from existing notations (such as UML and ITU standards). Third, textual notations should be legible with minimal understanding of *RoboCert* specifics.

This document describes version 0.1 of *RoboCert*. This is a working prototype of *RoboCert*, including support for *tock*-CSP checking of both raw CSP refinements and UML-style sequence diagrams against *RoboChart* models.

0.1 How to read this manual

This section discusses the layout and conventions used in this manual.

Structure

This manual is laid out as follows:

1. Information about *RoboCert*, its purpose, and this manual;
2. An introduction to each notation supported by *RoboCert*, discussing the metamodel and any concrete notations attached to each;
3. The formal semantics of *RoboCert*, provided as a separate development for each target verification language (CSP, PRISM, etc.).

Typography

This manual uses various typographical conventions. Some chapters have extra conventions not discussed here; these appear in similar sections at the start of the chapter.

Metamodel

- Class names look like this.
 - Class names with superscript *rc*, like this^{rc}, reference *RoboChart* class names.
- Feature names look like *this*.
 - Feature names with superscript *d*, like *this*^d, are *derived*: their value can be computed from other features and the model object graph. While the tooling defines many derived features, this manual only mentions those that participate in well-formedness conditions or are helpful for understanding the model.
- Enumeration variants look like **THIS**.

Syntax

- Concrete textual syntax looks like `this`. Boldface denotes keywords in the textual language.

RFC 2119

The key words **must**, **must not**, **required**, **shall**, **shall not**, **should**, **should not**, **recommended**, **may**, and **optional**, when in boldface, are to be interpreted as described in RFC 2119.

0.2 Running examples

This manual makes use of the following running examples, corresponding to real, publicly available *RoboChart* case studies:

Segway The *Osoyoo Segway robot*, model version 4, by Baxter and Cavalcanti.



Core Language

1	Core: metamodel	5
1.1	Introduction	
1.2	Top-level	
1.3	Targets	
1.4	Values	
1.5	Assertions	
2	Core: textual syntax	13
2.1	Conventions	
2.2	Top-level	
2.3	Targets	
2.4	Values	
2.5	Assertions	
3	Core: well-formedness	17
3.1	Introduction	
3.2	General conditions	
3.3	Top-level	
3.4	Targets	
3.5	Values	
3.6	Assertions	

The *RoboCert core language* has constructs and syntactic conventions that are common across, or sit above, the various sub-notations. It includes:

- the top-level structure of packages, sub-notations, and assertions;
- the core properties (deadlock freedom, determinism, and so on).

1. Core: metamodel

The metamodel of *RoboCert* is a key part of its definition. Parts of the metamodel map to various concrete notations (for instance, sequences map to UML-style sequence diagrams), have a semantics in terms of various formalisms (chapter 8), and are the subject of well-formedness conditions (for instance, chapter 3).

This chapter discusses the *core* metamodel, covering the key concepts of *RoboCert*. Other chapters and sections define the metamodels of the sub-notations:

- Chapter 4 discusses the metamodel of sequence diagrams;
- Section 7.1 discusses the metamodel of *RoboCert*'s features for CSP integration.

1.1 Introduction

Here, we introduce the core metamodel (as well as concepts used in other metamodel chapters).

1.1.1 Example

Below is an example of a *RoboCert* package with a Target^a (§ 1.3) and multiple Group^as (§ 1.2.2). This package references the Segway example described in § 0.2.

```

// Here is a SpecificationGroup with a Target, a ConstAssignment, two Actors,
// and an Interaction.
// We discuss the precise metamodel and notation for sequences later on.
specification group SM
  // All specifications in this package target this RModule
  target = module Segway with
    AnglePID::P, AnglePID::D, SpeedPID::P, SpeedPID::I, and RotationPID::D set to 0

  actors = { target as T, world as W } // These are available to all Interactions

  sequence disableInterrupts_loopTime
    actors T and W
    any until: T->>W: op disableInterrupts()
    loop: duration (between 0 and loopTime units) on T: any until: T->>W: op disableInterrupts()

// Two AssertionGroups: one contains core property assertions; another, sequence properties.
assertion group CoreProps
  assertion CP1: target of SM is deadlock free
  assertion CP2: target of SM does not terminate
assertion group SeqProps
  assertion SP1: SM::disableInterrupts_loopTime holds in the traces model
  assertion SP2: SM::disableInterrupts_loopTime holds in the timed model

```

1.1.2 How to read the rest of this chapter

Each section below introduces an aspect of core *RoboCert* functionality. These sections contain:

- a class diagram representing the Ecore classes, enumerations, and other components that make up the group being discussed;
- descriptions of the components being shown in the class diagram;
- discussions of how the components relate to counterparts in other notations (such as UML);
- where relevant, examples of the components in terms of the concrete syntaxes of *RoboCert*.

1.2 Top-level

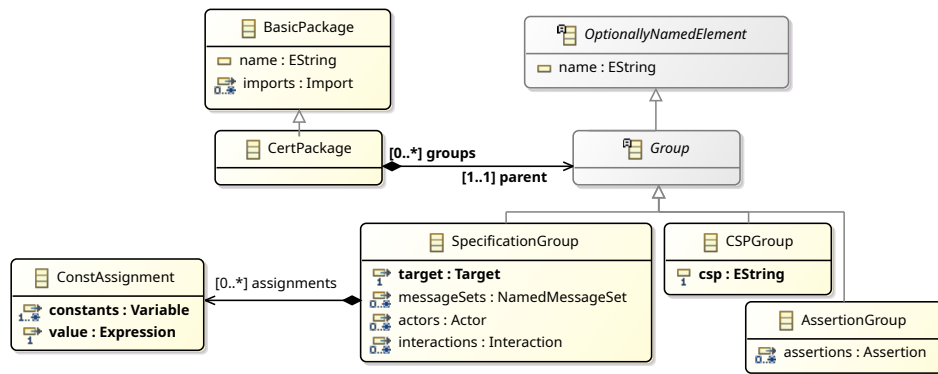


Figure 1.1: Class diagram for the top of the *RoboCert* metamodel.

Figure 1.1 is the top-level metamodel diagram for *RoboCert*.

1.2.1 CertPackage

Each *RoboCert* script corresponds to a *CertPackage*, which is a type of *RoboStar BasicPackage*^{rc}. The *CertPackage* has zero or more *Group*^s (§ 1.2.2), which further organise the specifications and assertions the package contains.

1.2.2 Group^a

```

specification group Group // a SpecificationGroup
  target = module Mod with // 'with' optional if no assignments
    Const1, Const2 set to 5 // ConstAssignment
    Const3 set to 6

  actors = { target as T, world as W }
  message set M1 = universe
  sequence Example1
    actors T and W
    anything until: deadlock on T

assertion group Group2: // an AssertionGroup
  assertion Example1: Group::Example1 holds in the traces model

```

Each CertPackage contains an ordered list of zero or more Group^as. These group specifications and assertions into structured, optionally-named blocks. There are three forms of group in *RoboCert* version 0.1: SpecificationGroups, AssertionGroups, and CSPGroups (§ 7.1.1).

SpecificationGroup

A SpecificationGroup is a Group^a that holds specifications. In version 0.1 of *RoboCert*, these are invariably Interactions (§ 4.2.2). Each SpecificationGroup has:

- a Target^a, *target* (§ 1.3);
- zero or more ConstAssignments, *assignments* (§ 1.2.3);
- zero or more Interactions (§ 4.2.2);
- zero or more auxiliary sequence items (chapter 4):
 - Actor^as (§ 4.2.1);
 - NamedMessageSets (§ 4.5.2).

AssertionGroup

A AssertionGroup is a Group^a subclass that holds Assertions (§ 1.5).

1.2.3 ConstAssignment

```

Const1 set to 0
Const2, Const3 set to 1

```

A ConstAssignment is an assignment of the value of an Expression^{rc} to one or more constant Variable^{rc}s left open in the parameterisation of a Target^a. The set of ConstAssignments inside a SpecificationGroup affect the instantiation of the target of the group, and **must not** overlap each other or any constants already instantiated on the target.

1.3 Targets

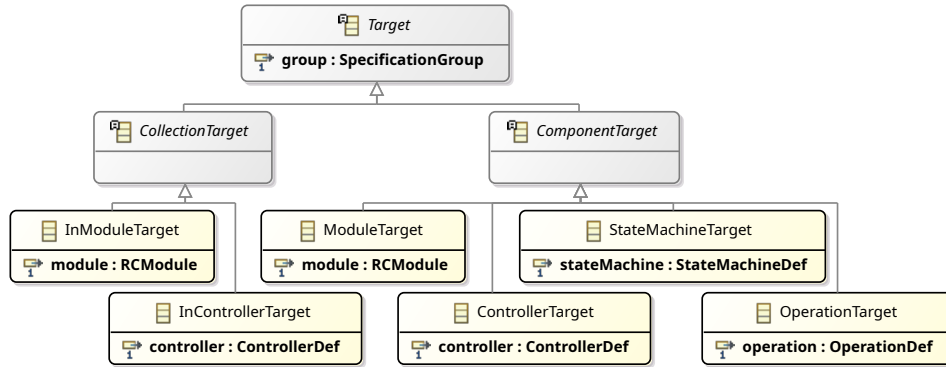


Figure 1.2: Class diagram for the part of the *RoboCert* metamodel dealing with targets.

A *Target*^a (Figure 1.2) is a reference to a *RoboStar* model component (its *element*). Each, as the name suggests, is a target of the *RoboCert* specifications inside a *SpecificationGroup* (§ 1.2.2). Each target stores its element as a distinct feature typed to the appropriate *RoboChart* class.

1.3.1 ComponentTarget^a

```

module AModule           // ModuleTarget
controller AController   // ControllerTarget
state machine AStm       // StateMachineTarget
operation AnOp           // OperationTarget
  
```

ComponentTarget^as capture specifications over interactions between an opaque component element and the ‘world’ (environment). Table 1.1 lists the current subclasses of *ComponentTarget*^a.

CollectionTarget ^a	Model class	Feature	Keyword
ModuleTarget	RModule ^c	<i>module</i>	module
ControllerTarget	Controller ^c	<i>controller</i>	controller
StateMachineTarget	StateMachine ^c	<i>stateMachine</i>	state machine
OperationTarget	Operation ^c	<i>operation</i>	operation

Table 1.1: *ComponentTarget*^as available in *RoboCert* version 0.1. For each: its target component class, the feature referring to that class, and the keyword used in the textual notation.

1.3.2 CollectionTarget^a

```

components of module AModule // InModuleTarget
components of controller AController // InCollectionTarget
  
```

CollectionTarget^as target the collection of *subcomponents* of a model component in terms of each other and of the component’s world. Table 1.2 lists the current subclasses of *CollectionTarget*^a.

CollectionTarget ^a	Component		Keyword (components of +)	Subcomponents <i>Model class</i>
	<i>Model class</i>	<i>Feature</i>		
InModuleTarget	RModule ^{rc}	<i>module</i>	module	Controller ^{rc}
InControllerTarget	Controller ^{rc}	<i>controller</i>	controller	StateMachine ^{rc}

Table 1.2: CollectionTarget^as available in *RoboCert* version 0.1. For each: its target component class, the feature referring to that class, the textual keyword, and the class of the subcomponents.

There are no CollectionTarget^as for state machines or operations, as these have no subcomponents that can be reasoned about as distinct units of behaviour.

1.4 Values

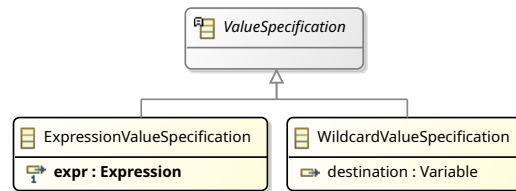


Figure 1.3: Class diagram for the part of the *RoboCert* metamodel dealing with value specifications.

This section discusses *value specifications*, and the use of *RoboChart* expressions by *RoboCert*.

1.4.1 Use of Expression^{rc}

RoboCert uses the existing *RoboChart* expression type, Expression^{rc}, as its expression type. This type, in turn, is based on the expression language of the Z notation.

We adopt this existing language to harmonise expression treatment across the *RoboStar* notations. This choice overrides the usual tendency in *RoboCert* to adopt the conventions of similar property languages (such as UML).

1.4.2 ValueSpecification^a

```

42 // ExpressionValueSpecification containing integer literal
foo // ExpressionValueSpecification containing constant reference

any into x // WildcardValueSpecification, binding to x
any       // WildcardValueSpecification, no binding
anything  // WildcardValueSpecification, alternative phrasing
  
```

A ValueSpecification^a is a pattern that specifies (and possibly binds) a value. There are two types:

- an ExpressionValueSpecification specifies that the value equals that of an Expression^{rc} *expr*;
- a WildcardValueSpecification specifies that the value can be *any* value allowed by the type of the corresponding parameter.

WildcardValueSpecifications have an optional *assignment* feature. If present, this is a Variable^{rc} that receives the actual argument value; precisely which Variable^{rc}s are in scope depends on the context of the value specification.

Differences from UML

ValueSpecification^as correspond in intent to the UML concept of the same name ([4, page 69]), but have a different metamodel. This is mainly to accommodate the existing *RoboChart* expression metamodel (which treats literals as expressions, for instance), but also to allow a *RoboChart*-style data model where we can capture certain observations into specification-level variables.

1.5 Assertions

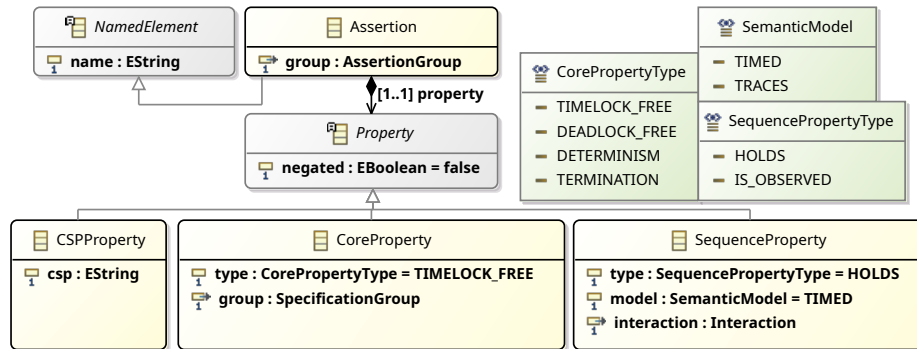


Figure 1.4: Class diagram for the part of the *RoboCert* metamodel dealing with assertions.

Assertions (fig. 1.4) define which properties should be verified, and how the verification should occur. As with many *RoboCert* concepts, assertions inhabit a Group^a subclass (**AssertionGroup**).

1.5.1 AssertionGroup

```

assertion group
  assertion A: SeqGroup::A holds in the timed model

// Assertions at top level implicitly form a singleton unnamed group,
// so this form is equivalent to that above.
assertion A: SeqGroup::A holds in the timed model

// Assertion groups can, themselves, be named.
// This assists with keeping related assertions together.
assertion group Grp
  assertion B: SeqGroup::B does not hold in the traces model
  assertion C: SeqGroup::C does not hold in the traces model

```

An **AssertionGroup** is an optionally-named Group^a containing zero or more **Assertions**.

1.5.2 Assertion and Property^a

```

assertion A: SeqGroup::A holds in the timed model

```

An **Assertion** is a single assertion. Each has a *name* and one of the following types of Property^a:

- **CoreProperty** (§ 1.5.3).
- **SequenceProperty** (§ 4.6);
- **CSPProperty** (§ 7.1);

1.5.3 CoreProperty

target of Group is deadlock-free
 target of Group is not timelock-free
 target of Group terminates
 target of Group does not terminate

A CoreProperty is a Property^a that represents a key property of the Target^a of the enclosing Cert-Package. The types of property form an enumeration CorePropertyType, of which *type* is a member. Table 1.3 describes this enumeration. As with any Property^a, core properties can be *negated*.

CorePropertyType value	Keyword		Checks
	<i>Positive</i>	<i>Negative</i>	
DEADLOCK_FREE	is deadlock free	is not deadlock free	timed deadlock freedom
TIMELOCK_FREE	is timelock free	is not timelock free	timelock freedom
DETERMINISM	is deterministic	is not deterministic	determinism
TERMINATION	terminates	does not terminate	termination

Table 1.3: Core properties available in *RoboCert* version 0.1.

1.5.4 SemanticModel

traces
 timed

SemanticModel is an enumeration of semantic models, presently used in the definition of SequenceProperty. In *RoboCert* version 0.1, there are two models:

- **TRACES** is a lightweight trace inclusion model, where no liveness reasoning is done and time is only considered as far as being a discrete recurring event in the trace.
- **TIMED** (the default) is a full timed liveness analysis model, where the target must both accept and refuse events according to the specification. This model distinguishes between provisional and mandatory behaviour as well as between successful termination and deadlock, and exposes target constraints on the world (for instance, deadlines).

2. Core: textual syntax

This chapter discusses the core textual syntax of *RoboCert*.

2.1 Conventions

We lay out the syntax in a variant of Backus-Naur form, where:

- nonterminals named after metamodel classes represent the syntactic representation of those classes;
- the suffix $*$ denotes zero or more instances of the preceding element;
- the suffix $+$ denotes one or more instances of the preceding element;
- the suffix $?$ denotes zero or one instances of the preceding element;
- parentheses group elements together for the purposes of the above;

2.1.1 Terminals

Because *RoboCert* uses significant whitespace, the symbols \rightarrow and \leftarrow denote special terminals inducing a mandatory increase (respectively, a decrease) in the indent level.

The following conventional terminals also exist:

- NAME is a single, unqualified Java-style identifier;
- QUALIFIED-NAME is a run of NAMEs conjoined by the namespacing operator ‘ $::$ ’.

2.2 Top-level

$\langle \text{CertPackage} \rangle ::= \langle \text{Group}^a \rangle^*$

2.2.1 Group^a

$\langle \text{Group}^a \rangle ::= \langle \text{CSPGroup} \rangle \mid \langle \text{SpecificationGroup} \rangle \mid \langle \text{AssertionGroup} \rangle$

$\langle \text{SpecificationGroup} \rangle ::= \text{‘specification’ ‘group’ ID} \rightarrow \langle \text{targetSpec} \rangle \langle \text{specElement} \rangle^* \leftarrow$

$\langle \text{targetSpec} \rangle ::= \text{‘target’ ‘=’} \langle \text{Target}^a \rangle \langle \text{instantiation} \rangle?$

$$\langle instantiation \rangle ::= \text{'with'} \rightarrow \langle ConstAssignment \rangle^+ \leftarrow$$

$$\langle specElement \rangle ::= \langle actorList \rangle \mid \langle Interaction \rangle \mid \langle NamedMessageSet \rangle$$

2.2.2 ConstAssignment

This rule set features a pattern common to lists of *RoboCert* elements: it allows the use of natural English-style list delimiters for the constant name list. For instance, x ; x and z ; x, y and z ; and $x, y, \text{ and } z$ are all valid.

$$\langle ConstAssignment \rangle ::= \langle constNames \rangle \langle assignWords \rangle \langle Expression^c \rangle$$

$$\langle constNames \rangle ::= \text{QUALIFIED_NAME} (', ' \text{QUALIFIED_NAME})^* (', '? \text{'and'} \text{QUALIFIED_NAME})?$$

$$\langle assignWords \rangle ::= \text{'set'} \mid \text{'to'} \mid \text{'assigned'}$$

2.3 Targets

$$\langle Target^a \rangle ::= \langle ComponentTarget^a \rangle \mid \text{'components of'} \langle CollectionTarget^a \rangle$$

$$\begin{aligned} \langle ComponentTarget^a \rangle ::= & \langle ModuleTarget \rangle \\ & \mid \langle ControllerTarget \rangle \\ & \mid \langle StateMachineTarget \rangle \\ & \mid \langle OperationTarget \rangle \end{aligned}$$

$$\langle ModuleTarget \rangle ::= \text{'module'} \text{QUALIFIED_NAME}$$

$$\langle ControllerTarget \rangle ::= \text{'controller'} \text{QUALIFIED_NAME}$$

$$\langle StateMachineTarget \rangle ::= \text{'state'} \text{'machine'} \text{QUALIFIED_NAME}$$

$$\langle ComponentTarget^a \rangle ::= \langle InModuleTarget \rangle \mid \langle InControllerTarget \rangle$$

$$\langle OperationTarget \rangle ::= \text{'operation'} \text{QUALIFIED_NAME}$$

$$\langle InModuleTarget \rangle ::= \text{'module'} \text{QUALIFIED_NAME}$$

$$\langle InControllerTarget \rangle ::= \text{'controller'} \text{QUALIFIED_NAME}$$

2.4 Values

$$\langle ValueSpecification^a \rangle ::= \langle ExpressionValueSpecification \rangle \mid \langle WildcardValueSpecification \rangle$$

$$\langle WildcardValueSpecification \rangle ::= \langle anyWord \rangle \langle wildcardCapture \rangle?$$

$$\langle anyWord \rangle ::= \text{'any'} \mid \text{'anything'}$$

$$\langle wildcardCapture \rangle ::= \text{'into'} \text{NAME}$$

2.5 Assertions

Note that the production for *CorePropertyType* also affects whether the core property is negated.

$\langle \text{AssertionGroup} \rangle ::= \text{'assertion'} (\langle \text{Assertion} \rangle \mid \langle \text{fullAssertionGroup} \rangle)$

$\langle \text{fullAssertionGroup} \rangle ::= \text{'group'} \text{NAME} \rightarrow (\text{'assertion'} \langle \text{Assertion} \rangle)^+ \leftarrow$

$\langle \text{Assertion} \rangle ::= \text{NAME} \text{' : ' } (\langle \text{Property}^\alpha \rangle \mid \rightarrow \langle \text{Property}^\alpha \rangle \leftarrow)$

$\langle \text{Property}^\alpha \rangle ::= \langle \text{CoreProperty} \rangle \mid \langle \text{CSPPProperty} \rangle \mid \langle \text{SequenceProperty} \rangle$

$\langle \text{CoreProperty} \rangle ::= \text{'target'} \text{' of ' } \text{QUALIFIED_NAME} \langle \text{CorePropertyType} \rangle$

$\langle \text{CorePropertyType} \rangle ::= \text{'does'} \text{' not'? } \langle \text{doesCorePropertyType} \rangle$

$\mid \text{'is'} \text{' not'? } \langle \text{isCorePropertyType} \rangle$

$\mid \langle \text{verbCorePropertyType} \rangle$

$\langle \text{doesCorePropertyType} \rangle ::= \text{'terminate'}$

$\langle \text{isCorePropertyType} \rangle ::= \text{'deadlock-free'} \mid \text{'deterministic'} \mid \text{'timelock-free'}$

$\langle \text{verbCorePropertyType} \rangle ::= \text{'terminates'}$

$\langle \text{SemanticModel} \rangle ::= \text{'traces'} \mid \text{'timed'}$

3. Core: well-formedness

This section gives well-formedness conditions for the metamodel in chapter 1.

3.1 Introduction

Here, we discuss the common features of this and other well-formedness chapters in the manual.


Each condition has:

- a error code such as **XYZ1** (also used in the tooling), which is hierarchical and based on the target subnotation, class, and feature;
- a textual description of the form ‘an *X* **must** *Y*’ or ‘an *X* **must not** *Y*’, where *X* is the object on which the condition is evaluated, and *Y* is a property that must (or must not) hold of *X*;
- a rationale (in a remark below the description).

3.2 General conditions

These conditions apply across all parts of a *RoboCert* script. We state them here to avoid repetition in the class-specific conditions later on.


- G1** A `NamedElementc` or `OptionallyNamedElementc` **must not** have the same fully-qualified name as another element in the same `CertPackage`.

 This prevents naming ambiguity and means the tooling need not prevent name clashes.


- G2** A `NamedElementc` or `OptionallyNamedElementc` **must not** have the same fully-qualified name as a *RoboChart* `NamedElementc` in any `RCPackagec` in the same resource set.

 As above, but this is primarily to prevent name clashes at the semantics level.

- G3** A feature with a multiplicity requirement **must** contain a quantity of values that conforms to the requirement.

-  This prevents unexpected situations where the semantics depends on a particular number of values for a feature.

G4 A cross-reference **must** refer to a well-formed *RoboCert* or *RoboChart* object; *RoboChart* objects are subject to the *RoboChart* well-formedness conditions.

-  This makes well-formedness a transitive closure, and lets us rely on the well-formedness conditions guaranteed by *RoboChart*.

3.3 Top-level

This section contains well-formedness conditions for the classes defined in § 1.2.

3.3.1 CertPackage (§ 1.2.1)

There are no well-formedness conditions for this class.


3.3.2 Group^a (§ 1.2.2): CG

There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses. In *RoboCert* version 0.1, there are only well-formedness conditions for *SpecificationGroups*.

3.3.3 SpecificationGroup: CGs

Feature assignments: CGsC

CGsC1 The *assignments* of a *SpecificationGroup* **must not** overlap in their sets of *constants*.


-  It is ambiguous as to which assignment should be chosen in the case of an overlap.

Feature messageSets


There are no well-formedness conditions for this feature.

Feature actors: CGsA


CGs1 A *SpecificationGroup* **must not** contain two duplicate *Actor*^as.¹

-  This would complicate the language for no gain in expressivity. Exercising this rule also forbids ill-formed situations such as a *ComponentTarget*^a group with two *TargetActors* or a *CollectionTarget*^a group with no *ComponentActors*.

CGs2 A *SpecificationGroup* for a *CollectionTarget*^{a2} **must not** contain a *TargetActor*.

-  Such sequence groups look *inside* the target, so we cannot reason opaquely about the target's behaviour.

CGs3 A *SpecificationGroup* for a *ComponentTarget*^{a3} **must not** contain a *ComponentActor*.

-  Such sequence groups look *outside* the target, so we cannot reason directly about behaviour of the target's components.

¹We consider two *Actor*^as to be duplicates if they are both the same type of *Actor*^a and, if *ComponentActors*, both reference the same model component.

²These include, for example, *InModuleTargets*.


³These include, for example, *ModuleTargets*.

Feature *interactions*


There are no well-formedness conditions for this feature.

3.3.4 ConstAssignment (§ 1.2.3): CC**Feature *constants*: CCC**


CCC1 The *constants* of a ConstAssignment **must** be unique.

 It is ambiguous as to which assignment should be chosen in the case of an overlap.

CCC2 The *constants* of a ConstAssignment **must** have the **CONST** variable modifier.

 Setting model variables through this mechanism makes no sense.

CCC3 The *constants* of a ConstAssignment **must** belong to the parameterisation of the relevant Target^a.


 Instantiating other constants through this mechanism makes no sense; said instantiations would not be used when determining the effective target of any assertions.

Feature *value*: CCV

CCV1 The *value* of a ConstAssignment **must** be type-compatible with all *constants*.

 Type safety.

CCV2 The *value* of a ConstAssignment **must not** reference any variables.

 The only variables in scope here would be other constants in the Target^a instantiation. Out of an abundance of caution, we forbid these to prevent recursive and mutually recursive instantiations.

3.4 Targets

In *RoboCert* version 0.1, there are no well-formedness conditions on targets (§ 1.3).

3.5 Values


This section contains well-formedness conditions for the classes defined in § 1.4.

3.5.1 ValueSpecification^a (§ 1.4.2): CV


There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses.

3.5.2 ExpressionValueSpecification: CVe**Feature *expr*: CVeE**

CVeE1 Variables with modifier **VAR** referenced in an ExpressionValueSpecification **must** belong to an enclosing Interaction.

 No other variables are visible to such expressions in *RoboCert* version 0.1. Reading values from model variables is not yet supported.


CVeE2 Variables with modifier **CONST** referenced in an ExpressionValueSpecification **must** belong to the parameterisation of the target of the enclosing SpecificationGroup.

 No other constants are visible to such expressions.

3.5.3 WildcardValueSpecification: CVw

Feature destination: CVeD

CVwD1 The destination of a WildcardValueSpecification **must** belong to an enclosing Interaction.

 Storing values to model variables is not yet supported, and storing values to constants makes no sense.

3.6 Assertions

This section contains well-formedness conditions for the classes defined in § 1.4.


3.6.1 Property^a

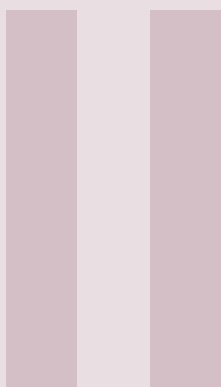
There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses. Note that there are no well-formedness conditions for SequenceProperty or CSPPProperty in *RoboCert* version 0.1.

3.6.2 CoreProperty (§ 1.5.3): CPc

Feature group: CPcG

CPcG1 The *group* of a CoreProperty **must** have a *target* that is a ComponentTarget^a.

 Core properties check high-level properties on single components; such checks have no meaning in CollectionTarget^as where the component is not tangible.



Sequence Notation

4	Sequences: metamodel	25
4.1	Introduction	
4.2	Interactions	
4.3	Interaction fragments	
4.4	Occurrences	
4.5	Messages	
4.6	Assertions	
5	Sequences: well-formedness	43
5.1	Sequences	
5.2	Fragments	
5.3	Occurrences	
5.4	Messages	
5.5	Assertions	
6	Sequences: textual syntax	51
6.1	Interactions	
6.2	Fragments	
6.3	Occurrences	
6.4	Messages	
6.5	Assertions	

The *sequence* notation of *RoboCert* provides a method of defining the expected interactions between actors in a robotic model, optionally with time constraints. For instance, sequences may specify how a *RoboChart* module uses services offered by a robotic platform. Sequences resemble UML sequence diagrams.

4. Sequences: metamodel

This chapter discusses the metamodel of *RoboCert* sequences. This metamodel has two concrete notations — *textual* (chapter 6) and *graphical* (not yet formally specified) —, and a semantics (chapter 8).

4.1 Introduction

For information on how to read this chapter, see the notes on the top-level metamodel (§ 1.1.2).

4.1.1 Naming and UML compatibility

Where possible, we name concepts after their UML counterparts, or the nearest UML concept. This is why, for instance, a sequence diagram is an instance of Interaction (§ 4.2.2).

RoboCert sequences are derived from, but not fully compatible with, UML sequence diagrams. Nonetheless, where we diverge from UML, we explain and justify the differences. General differences include:

- The UML concept of ‘named elements’ is referred to as an `OptionallyNamedElementa` here, to avoid clashing with the *RoboChart* `NamedElementc`.

4.1.2 Ordering and timing

Unless modified using fragments (§ 4.3), communications on *RoboCert* lifelines are strictly ordered with respect to each other (but not necessarily those on other lifelines) and permit no intervening communications. This reflects the view of sequence diagrams as representations of traces of the system under test, and is consistent with the canonical UML semantics [4, 8].

RoboCert sequences are, by default, *explicit* as to *which* actions occur, but *implicit* as to *when*. *RoboCert* adopts the *RoboChart* discrete-time model, where time is measured in *time units* and events, operation calls, and primitive data operations (assignments, communication, and so on) are instantaneous. In this light, `Occurrencea`s (§ 4.4) represent instants in time where something occurs, preceded by a time interval that defaults to being unbounded. To constrain the flow of time in a sequence, use `DeadlineFragments` (§ 4.3.3).

4.1.3 Temperature

When performing liveness reasoning, we must distinguish between parts of a sequence where the model has control over when things can occur, and parts where it must be ready to engage in any possibility it offers. For instance, an *AltFragment* (§ 4.3.9) could represent either a model-level choice of multiple possible implementations (a ‘provisional’ alternative), or a control flow where the environment can influence the model behaviour by providing different inputs (a ‘mandatory’ alternative). These two become distinct for liveness purposes as the former permits the model to refuse to engage in unimplemented alternatives.

UML does not inherently have a way to distinguish the two cases above, though variants of UML such as *STAIRS* [5] extend it with constructs such as mandatory alternatives. Our approach is to use the Live Sequence Charts [2] notion of a ‘temperature’ modality: certain parts of a sequence can be ‘hot’ or ‘cold’. In *RoboCert*, this modality affects liveness reasoning as follows:

- Cold elements permit the model to decide when, if ever, it is ready to engage in that element. For instance, cold *AltFragments* permit the element to leave certain branches unimplemented; cold *MessageOccurrences* can be refused indefinitely by the model; and so on.
- Hot elements require the model to be available, at any point in time, to engage in any aspect of that element. For instance, hot *AltFragments* insist that the model be ready for any of the branches to occur; hot *MessageOccurrences* specifying incoming *EventTopics* require that the model be ready to engage in the communication at any moment; and so on.

The ‘cold’ modality is the default, is a weaker obligation on the model, and is only different from the ‘hot’ modality when performing liveness reasoning. As such, sequences should usually start with all elements marked ‘cold’ with a gradual transition to ‘hot’ modalities where needed.

We use a temperature modality for various reasons. The main reason is its existing use in LSC. Another is that it captures liveness dichotomies effectively over various distinct parts of the *RoboCert* metamodel. A third is that the concept of a ‘hot’ element is a good intuition to the liveness behaviour: just as we reflexively drop hot items quickly, so must *RoboChart* models be ready to effect a hot sequence element at any time, including immediately.

4.2 Interactions

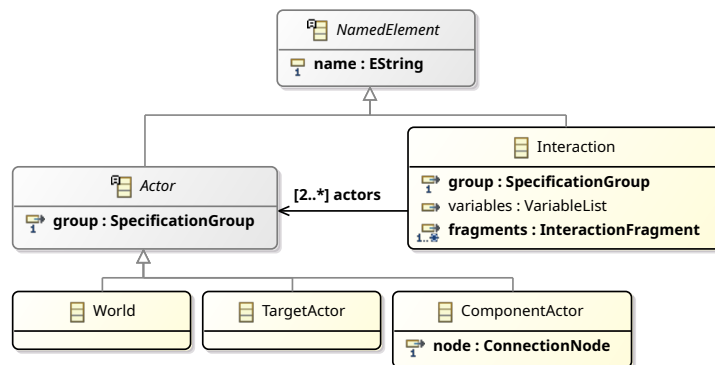


Figure 4.1: Class diagram for the part of the *RoboCert* metamodel dealing with sequences.

The sequence notation (fig. 4.1) consists of sequences (formally termed ‘interactions’) and related items (such as message sets and actors).

4.2.1 Actor^a

```
target      // TargetActor
component Stm1 // ComponentActor
world       // World
```

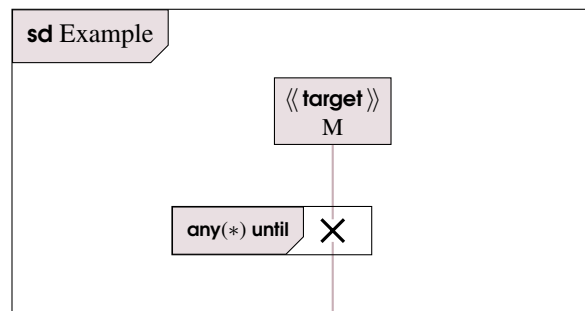
Actor^as capture sequence communication participants. A SpecificationGroup defines a set of Actor^as, and each Interaction (§ 4.2.2) maps a subset of those actors to lifelines. There are three types:

- a TargetActor is an Actor^a that represents a ComponentTarget^a in a sequence. It has no features;
- a ComponentActor is an Actor^a that represents a subcomponent (that is, a ConnectionNode^{rc}) of a CollectionTarget^a;
- a World is an Actor^a that represents the ‘world’ in a sequence: the context into which the target is placed. For instance, the world of a ModuleTarget is the world (in the *RoboWorld* sense), as viewed through the services provided by the robotic platform; the world of a ControllerTarget is the robotic platform plus any other controllers connected to the Controller^{rc}; and so on.

R Unlike most actors, a World does *not* semantically or notationally induce a lifeline. This is because the World is ‘outside’ the component constrained by the sequence, and our ability to reason about its behaviour is correspondingly limited. Instead, Worlds behave like UML formal gates: graphically, the world corresponds to the right edge of a diagram.

4.2.2 Interaction

```
sequence Example
actors M and W // lifelines
var x: int     // variables
anything until: deadlock on M // body
```



An Interaction represents a sequence diagram. It is a NamedElement^{rc} that contains:

- an unordered list *actors* of Actor^as, representing lifelines and gates in the diagram and forming a subset of *actors* from the parent SpecificationGroup;
- a VariableList^{rc} *variables*, which declares variables to be used by WildcardValueSpecifications and Expression^{rc}s in the diagram;
- the *body* of the interaction, an ordered list of InteractionFragment^as (§ 4.3).

The example above shows the most permissive Interaction possible: a diagram that allows the system to do anything (and accept any communications from the robotic platform) until termination.

Differences from UML

An Interaction interleaves Occurrence^as from all lifelines into one vertical, lexically ordered flow, with nesting from CombinedFragment^as (§ 4.3). This differs from UML, where the containment of OccurrenceFragments in an Interaction is unordered, and a separate ‘events’ relation orders OccurrenceFragments into individual lifelines.

We use the vertical flow to maximise legibility for Interactions in the textual notation, especially in the case where there is no potential for parallelism between lifelines. A consequence is that

this approach complicates the slicing of the diagram into lifelines later on, as well as expressing concepts such as message overtaking.

The textual ordering of Occurrence^as does not necessarily correspond to a total ordering of their effect; like UML, there is no implicit synchronisation between lifelines. Figure 4.3 is an example using where the textual ordering overapproximates the actual ordering.

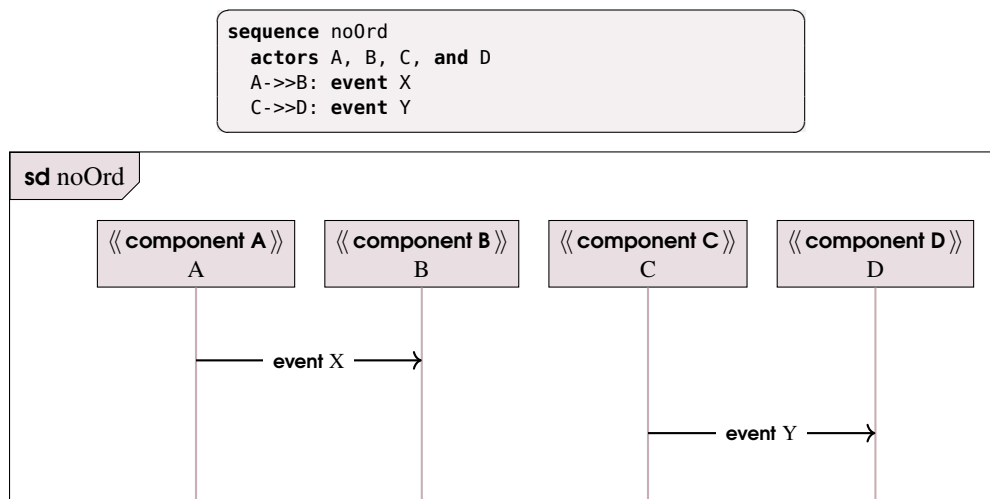


Figure 4.3: X and Y are unordered, as they are on disjoint lifelines with no synchronisation.

Another difference from UML is the inclusion of variable declarations at the top level of Interactions. These are necessary to allow capturing of dataflow properties, such as the value from one event being propagated to another event. Variable declaration and usage follows the conventions of *RoboChart*, to simplify variable use for existing *RoboChart* modellers.

4.3 Interaction fragments

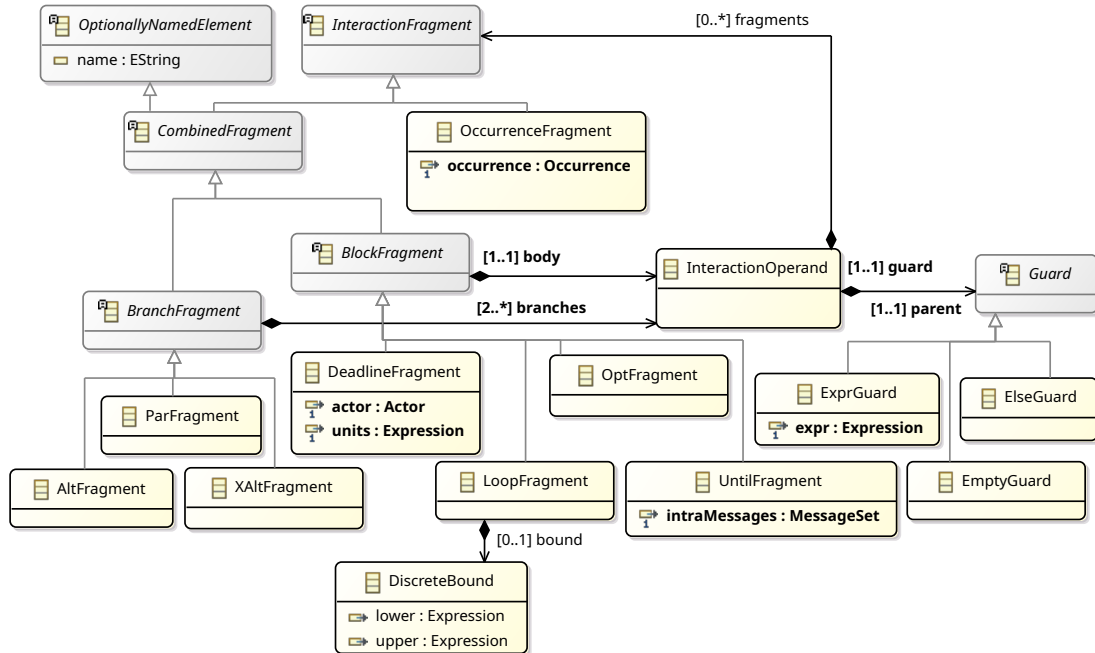


Figure 4.4: Class diagram for the part of the *RoboCert* metamodel dealing with fragments.

Interaction fragments (fig. 4.4), or ‘fragments’ for short, are elements of `InteractionFragmentQ`. They represent communications and control flow inside an Interaction.

Combined fragments

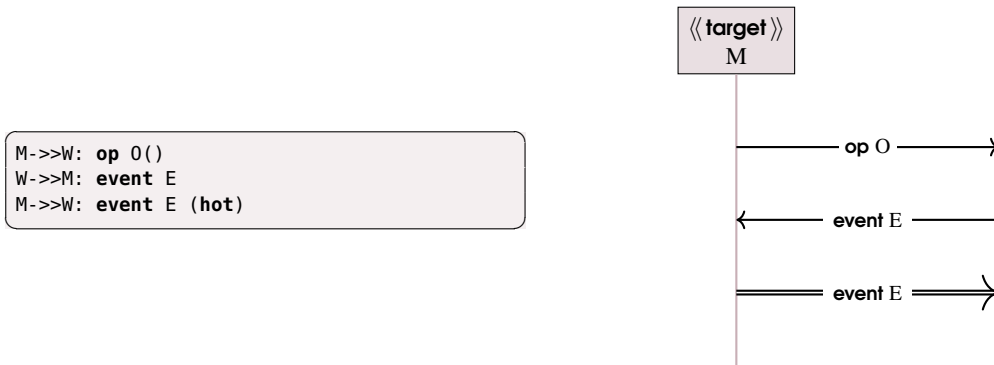
Some `InteractionFragmentQ`s contain sub-sequences of further `InteractionFragmentQ`s (both UML and *RoboCert* refer to these as `InteractionOperands`; see § 4.3.12). These form subclasses of `CombinedFragmentQ`, named by analogy to the UML concept of combined fragments.

Unlike UML, where combined fragments directly contain an ‘interaction operator’ followed by zero or more ‘interaction operands’, we organise `CombinedFragmentQ` into a subclass hierarchy based first on the number of expected operands and then on the operator. The first level is as follows:

- `BlockFragmentQ`s (§ 4.3.2) represent control flow over a single `InteractionOperand`;
- `BranchFragmentQ`s (§ 4.3.8) are combining operators over ≥ 2 `InteractionOperands`.

We do this to simplify both implementation (by making the metamodel follow commonalities in how we handle the fragments) and well-formedness (by ensuring the number of expected operands follows from the metamodel multiplicity).

4.3.1 OccurrenceFragment



An OccurrenceFragment lifts an Occurrence^o (§ 4.4) to a fragment.

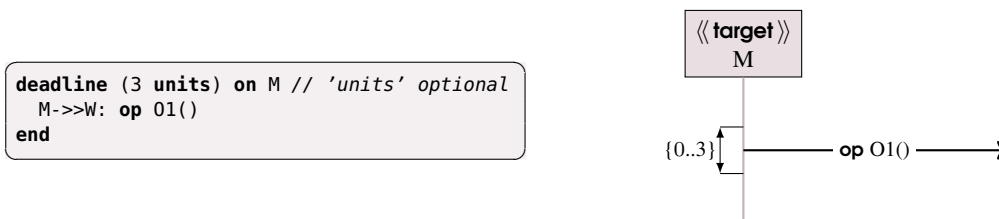
- R** By default, any amount of time (including no time at all) may pass between the start of a OccurrenceFragment and the effect of its enclosed Occurrence^o. Use DeadlineFragments (§ 4.3.3) to constrain this.

4.3.2 BlockFragment^a

A BlockFragment^a is a OccurrenceFragment that contains a InteractionOperand, *body*, and performs some form of control-flow lifting over it. There are four types of BlockFragment^a:

- DeadlineFragment (§ 4.3.3);
- LoopFragment (§ 4.3.4);
- OptFragment (§ 4.3.5);
- UntilFragment (§ 4.3.6).

4.3.3 DeadlineFragment



A DeadlineFragment places an upper bound on the amount of time that a InteractionOperand takes to complete *on a given Actor^a*. A required Expression^c, *units*, effects this constraint in terms of the number of time units passing from the perspective of the Actor^a *actor* inside the operand. Other Actor^as can participate in the operand, but the constraint does not directly affect them.

- R** To specify that the actions must occur immediately, set *units* to 0.

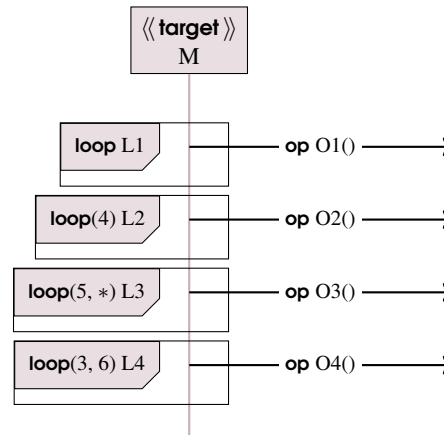
4.3.4 LoopFragment

```
// No bound
loop L1
  [always] // this can be omitted
  M->>W: op 01()
end

// exact bound
loop (4 times) L2
  M->>W: op 02()
end

// lower bound
loop (at least 5 times) L3
  M->>W: op 03()
end

// range bound
loop (between 3 and 6 times) L4
  M->>W: op 04()
end
```



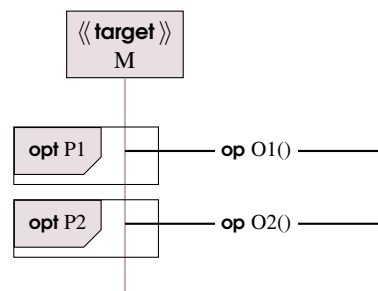
A LoopFragment is a loop over a InteractionOperand. The number of times a LoopFragment will iterate (unless deadlocked¹) depends on its attached DiscreteBound (§ 4.3.7), if any; if the bound is absent, the loop is infinite.

R Expression^{rc}s inside any loop DiscreteBound are evaluated *once*, before executing the body.

4.3.5 OptFragment

```
// target can call 01, but doesn't have to
opt P1
  M->>W: op 01()
end

opt P2: M->>W: op 02() // one-liner syntax
```



An OptFragment marks its *body* as *optional*: either the whole *body* occurs, or it does not. The target is allowed to refuse to perform the body; as in UML, an OptFragment is semantically equivalent to an AltFragment with two empty-guarded branches (one containing *body* and one containing nothing).

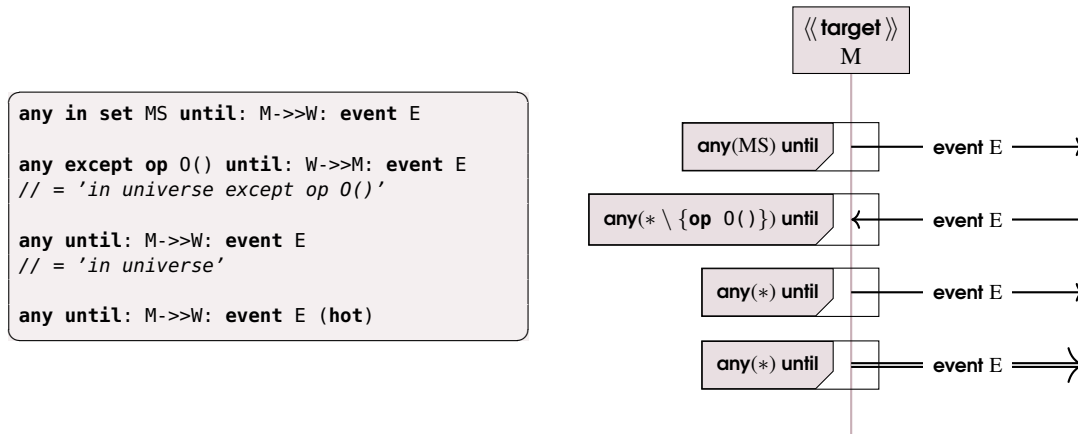
Despite the equivalence above, we justify retaining OptFragment in *RoboCert* (when languages such as Property Sequence Charts [1] remove it) as it more clearly and succinctly captures the notion of optionality than the AltFragment construction; also, removing it would diverge from UML.

Differences from UML

There are no differences between OptFragment and UML opt.

¹Future work will add the ability to break out of loops prematurely.

4.3.6 UntilFragment



An UntilFragment is a subclass of OccurrenceFragment that suspends ordinary Occurrence^αs on *all* lifelines. Instead, the diagram permits the exchange of arbitrary messages from a specified set *until* an Occurrence^α within the enclosed InteractionOperand takes effect.

Any InteractionOperand may be nested inside the UntilFragment, *except* if it begins with another UntilFragment². The sequence behaves as a loop over the messages in the MessageSet^α *intraMessages*, *except* for messages within the set of *potential* initial Messages reachable from the first InteractionFragment^α (if any) of the InteractionOperand. These are:

- for OccurrenceFragments carrying a MessageOccurrence, the Message;
- for BlockFragment^αs, the initial Messages of the subsequence of the block;
- for BranchFragment^αs, the union of the initial Messages of the subsequences of each branch (regardless of their Guard^αs);
- for everything else, the empty set.

UntilFragments induce a synchronisation on lifelines entering the fragment (the exchange of *intraMessages* does not start until all lifelines reach the top of the UntilFragment) and another synchronisation as soon as the Occurrence^α takes effect. The rest of the InteractionOperand proceeds with the usual ordering across lifelines.

Differences from UML

UntilFragments *do not* correspond to any UML combined fragment. Instead, they are a UML extension based on both the *intraMSG* constraints of Property Sequence Chart [1] and the **UNTIL** stereotype of Lindoso et al. [7].

Example: prefix sequence

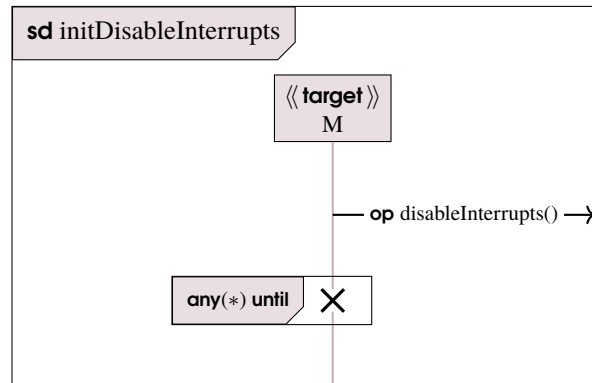
Despite being an extension to UML, UntilFragments lets us succinctly capture various common specification patterns. We outline these in several examples.

Sequences in *RoboCert* are *total*: they capture the behaviour of a robotic component from start to finish. This is in accordance with the standard UML semantics. However, it is common to want to only specify some *prefix* of the intended behaviour. This can be captured by following the intended prefix with an UntilFragment enclosing either deadlock (which forbids termination) or an indefinite wait (which permits it)³.

²This restriction serves only to make the set of initial messages enumerable, which is an implementation concern and may be relaxed in future revisions.

³This encoding does not work with ‘is observed’ assertions, as the semantics is that *every* possible selection of actions that can complete the sequence must be observable.

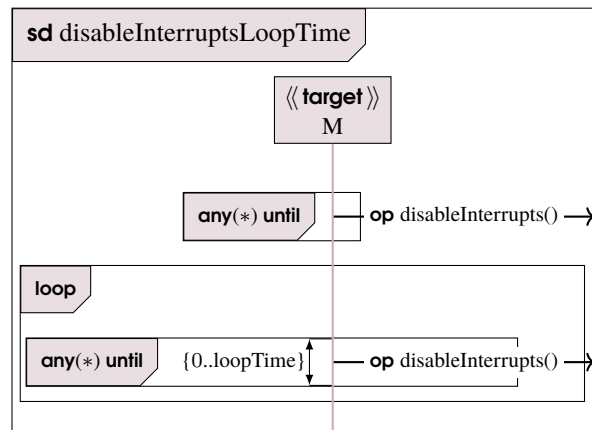
```
// The first thing the Segway does is
// call disableInterrupts()
sequence initDisableInterrupts
actors M and W
M->>W: op disableInterrupts()
any until: deadlock on M
```



Example: suffix sequences

Similarly, an UntilFragment at the *start* of the sequence lets us encode certain types of suffix sequence. This is similar to the concept of *pre-charts* in Live Sequence Charts [3]: preceding the main sequence by a period where the sequence is ‘inactive’ and activated by an Occurrence^a.

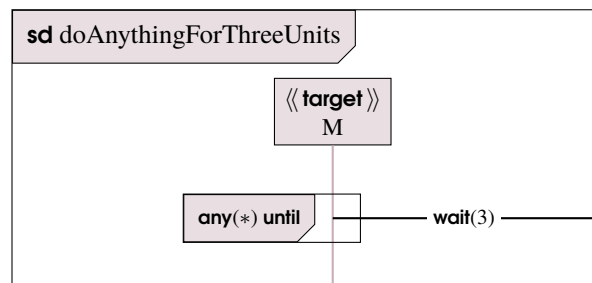
```
// After each call of disableInterrupts(),
// loopTime time units pass before the next
sequence disableInterruptsLoopTime
actors M and W
any until: M->>W: op disableInterrupts()
loop
duration (between 0 and loopTime) on M
any until
M->>W: op disableInterrupts()
end
end
end
```



Do anything for a certain amount of time

The combination of UntilFragments and WaitOccurrences (§ 4.4.2) lets us specify the act of blocking all lifelines for an amount of time units, but simultaneously permitting *intraMessages*.

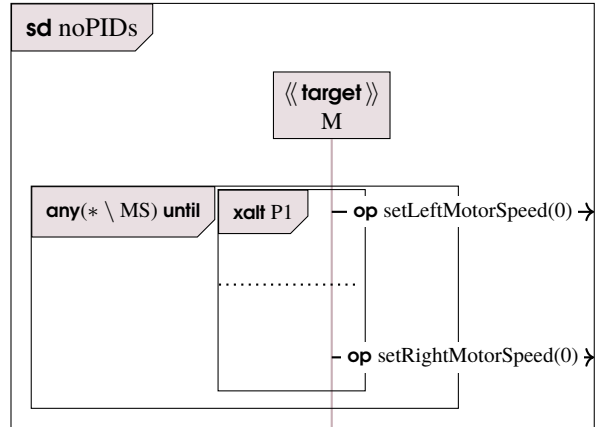
```
// We don't have an example for the
// Segway here, yet
sequence doAnythingForThreeUnits
actors M and W
any until: after 3 units on M
```



Use with AltFragment

By nesting an AltFragment (§ 4.3.9) inside an UntilFragment, we can capture the concept of waiting for one of multiple different Occurrence^as. This example constrains the Segway such that, every time a setLeftMotorSpeed OR setRightMotorSpeed operation occurs, its argument will be 0:

```
// When the PID constants are set to 0,
// the values set by setLeftMotorSpeed and
// setRightMotorSpeed() are 0.
message set MS =
{ M->>W: op setLeftMotorSpeed(any)
, M->>W: op setRightMotorSpeed(any) }
sequence noPIDs
actors M and W
loop
  any except MS until
    xalt: M->>W: op setLeftMotorSpeed(0)
  else: M->>W: op setRightMotorSpeed(0)
  end
end
end
```



4.3.7 DiscreteBound

```
6          // exactly 6
at least 5 // lower bound is 5
[2, 4]     // 2 to 4 inclusive
(3, 5)     // 3 to 5 exclusive (so, 4)
```

A *DiscreteBound* defines a potentially-open set of natural numbers, representing a constraint on the number of something (such as time loop iterations in *LoopFragments*). Each *DiscreteBound* contains a *Expression^{rc} count*, which is either a natural or a range expression over naturals. In the former case, a Boolean *lower* determines whether the resulting count is an exact bound or a lower bound.

R If setting an upper bound only, use a range expression with a lower bound of 0.

Differences from UML

We adopt the *RoboChart* concept of range expressions (with an extension to handle lower bounds) as the uniform way of handling ranges throughout *RoboCert*. In doing so, we break cosmetically from UML in loop bounds, where the lower and upper bounds of a doubly bound loop are separate expressions.

4.3.8 BranchFragment^a

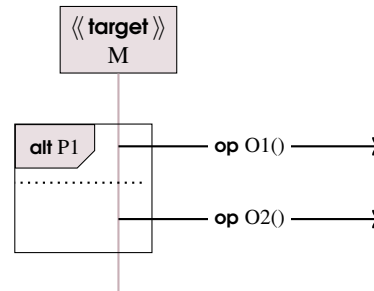
A *BranchFragment^a* contains two or more *InteractionOperands*, and represents a form of combining operator upon them. There are three types of *BranchFragment^a*:

- *AltFragment* (§ 4.3.9);
- *XAltFragment* (§ 4.3.10);
- *ParFragment* (§ 4.3.11).

There are several well-formedness conditions that govern which combinations of guards are allowed in a *BranchFragment^a* (see § 5.2.10).

4.3.9 AltFragment

```
// target may be capable of one or both ops
alt: M->>W: op O1()
else: M->>W: op O2()
end
```



An AltFragment, or *provisional alternative*, is a BranchFragment^a representing a decision point where the model can behave as *precisely one* of the branches provided. In a provisional alternative, the model **may** arbitrarily choose which branch to take, and refuse to accommodate other branches. This means that the sequence lets the model choose which parts of the alternative to implement.

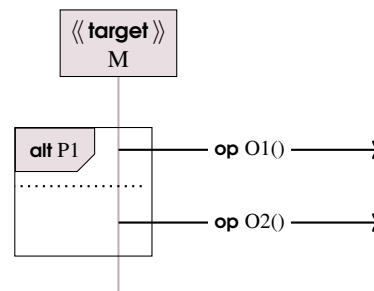
If there are no branches available with viable guards, the semantics is deadlock.

Differences from UML

This fragment corresponds to its UML counterpart. More specifically, it corresponds to the *STAIRS* view of provisional alternatives, with XAltFragment representing mandatory alternatives.

4.3.10 XAltFragment

```
// target must be capable of both of these
xalt: M->>W: op O1()
else: M->>W: op O2()
end
```



An XAltFragment, or *mandatory alternative*, is a BranchFragment^a representing a decision point where the model **must** be able to behave as any of the given branches, provided that their guards evaluate to true.

XAltFragment is a stronger form of AltFragment where the choice between branches may be taken by the environment rather than the model. This distinction only arises when using the **TIMED** semantic model; otherwise, AltFragment is equivalent and sufficient.

As with AltFragments, the semantics of an XAltFragment where no branches have guards evaluating to true is deadlock.

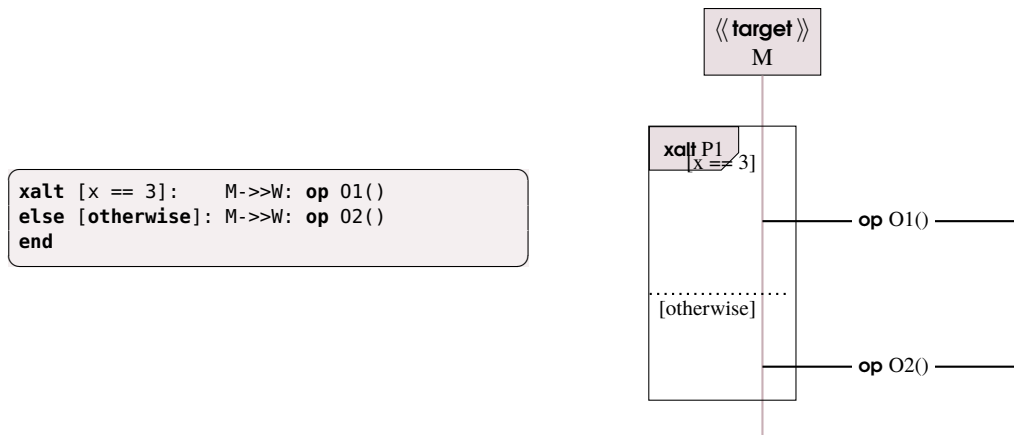
Differences from UML

This fragment corresponds to the similarly-named fragment in the *STAIRS* extension of UML sequence diagrams.

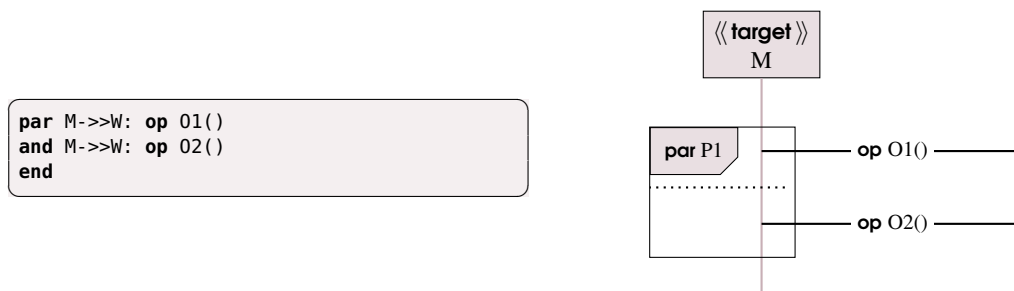
Encoding conditionals

A key purpose of XAltFragments is to encode conditionality. As with UML, XAltFragments are a more general (and low-level) construct than if-then-else conditionals found in programming languages;

encoding such conditionals requires the combination of a hot⁴ XAltFragment and Guard^as (4.3.8):



4.3.11 ParFragment



An ParFragment is a BranchFragment^a that represents an interleaving parallel composition of the branches offered, with MessageOccurrences being the atomic actions.

The motivation for this fragment is the same as in UML. While lifelines implicitly run in parallel with each other (horizontally), there exists a (vertical) ordering over actions within lifelines, as well as an implicit ordering effect when communications synchronise lifelines. ParFragment relaxes this default ordering by allowing InteractionOperands to interleave irrespective of lifelines involved.

Differences from UML

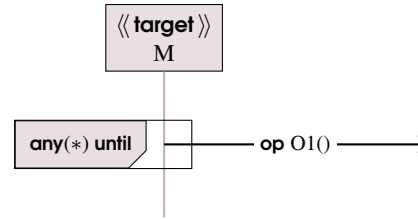
There are no differences from the UML concept of `par`.

- R Implementing the `strict` operator, which has the converse effect of implementing a total operator, is left to future work. The synchronising nature of UntilFragment may be used to perform some of the same effect.

⁴The use of *mandatory* alternative here serves to ensure that both branches are live; otherwise, the model could implement only one and deadlock on reaching the other.

4.3.12 InteractionOperand

```
// these are equivalent;
// here, everything after 'until' is
// an InteractionOperand
anything until
  [always] M->>W: op O1()
end
anything until
  M->>W: op O1() // elided [always]
end
// one-liner syntax
anything until: [always] M->>W: op O1()
anything until: M->>W: op O1()
```



An InteractionOperand is a slice of an Interaction that forms an operand to a CombinedFragment^a (§ 4.3). Like an Interaction, a InteractionOperand contains an ordered list of InteractionFragment^as. As in UML, InteractionOperands also contain Guard^as (§ 4.3.13), which guard entry to the operand according to the semantics of the enclosing CombinedFragment^a.

4.3.13 Guard^a

```
[always] // EmptyGuard (does not appear in graphical notation)
[a > b] // ExprGuard
[otherwise] // ElseGuard
```

A Guard^a places a constraint on the execution of an InteractionOperand. While the precise semantics of a Guard^a depends on its parent CombinedFragment^a (§ 4.3), the usual reading is that any Actor^a reaching a Guard^a that evaluates to false deadlocks. There are three types of Guard^a:

- EmptyGuard, which is always true;
- ExprGuard, which is true iff. its expression evaluates to true;
- ElseGuard, which is true iff. all other guards in the same BranchFragment^a evaluate to false.

4.4 Occurrences

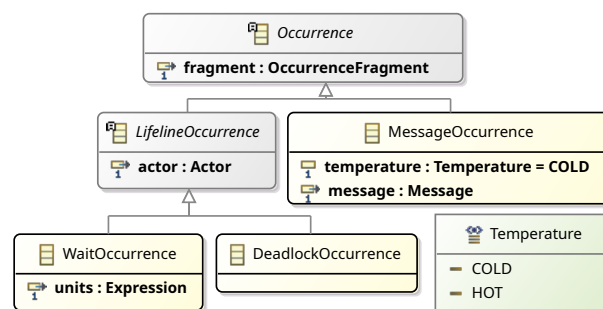


Figure 4.19: Class diagram for the part of the *RoboCert* metamodel dealing with occurrences.

Occurrences (fig. 4.19) represent points in time on a sequence when something is expected to occur. There are three types of occurrence, each of which is a subclass of Occurrence^a:

- MessageOccurrence (§ 4.4.1): a message is expected to occur;
- WaitOccurrence (§ 4.4.2): a delay is expected to occur;

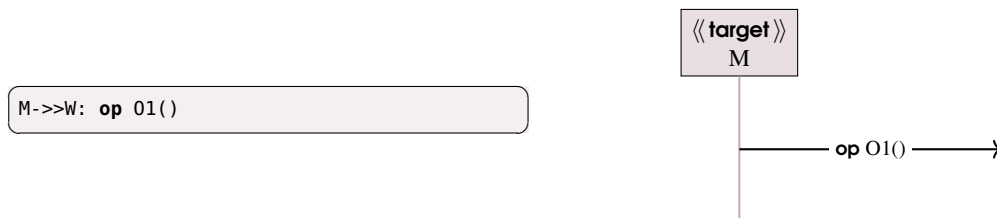
- **DeadlockOccurrence** (§ 4.4.3): a deadlock is expected to occur.

Of these, **WaitOccurrence** and **DeadlockOccurrence** are subclasses of **LifelineOccurrence**. This is an abstract class representing occurrences that take effect on precisely one lifeline.

Differences from UML

Unlike UML, there is no concept of an execution specification. This is because lifelines represent *RoboChart* components that begin execution simultaneously and remain executing unless terminated, and Messages do not themselves start or end processes or functions on the lifelines.

4.4.1 MessageOccurrence



An **MessageOccurrence** lifts a **Message** (§ 4.5) to an **Occurrence**^a.

Temperature

By default, a **MessageOccurrence** is ‘cold’: the model can refuse to engage in the occurrence for an arbitrary amount of time, but, once it does, the sequence will progress downwards. For instance, the occurrence $W \rightarrow T: \text{event } E$ is an **MessageOccurrence** (§ 4.4.1) specifying that *when* the Target^a accepts event E from the World, the sequence will progress; the target may refuse to accept E for an arbitrary amount of time. Similarly, $W \rightarrow T: \text{any until: event } E$ is an **UntilFragment** (§ 5.2.8) that allows the Target^a to engage in any communications with the World until it accepts E , but does not require that said acceptance is readily available.

Setting the *temperature* of a **MessageOccurrence** to **HOT** specifies instead that the model *must* be ready to participate in the communication as soon as it becomes available (or at any time afterwards, if not otherwise constrained). For instance, $W \rightarrow T: \text{event } E \text{ (hot)}$ specifies that the target must be willing to accept E immediately; $\text{any until: } W \rightarrow T: \text{event } E \text{ (hot)}$ allows the Target^a to engage in any communications with the World but requires that it is *always* ready to accept E ; this is a strong requirement, hence why it is not the default meaning.

Differences from UML

Unlike UML, where the send and receive ends of a message each form an occurrence on their respective lifelines, the whole message here forms *one* Occurrence^a. This is related to the flat structuring of **InteractionFragment**^as across all lifelines discussed in § 4.2.2. Separating the message ends would spread the message into three parts, at least two of which would still inhabit the Interaction, and induce undue complexity in the textual notation.

4.4.2 WaitOccurrence



A WaitOccurrence captures the *RoboChart* concept of wait statements, which induce a delay for an amount of time. In *RoboCert*, this amount is a ValueSpecification^a *units*, applied to one Actor^a. A key use for WaitOccurrences is to allow the specification of time budgets.

As in *RoboChart*, if *units* is an ExpressionValueSpecification over a RangeExp^c, the wait will be nondeterministic over the range of possible time units supplied; otherwise, it will be deterministic over the value of the expression.

If *units* is an WildcardValueSpecification, the model is expected to wait an indeterminate amount. This case is useful when using UntilFragment (§ 4.3.6), as it captures the concept of accepting messages from an *intraMessages* set indefinitely without deadlocking. There is presently no way to capture the amount of time waited into a Variable^c.

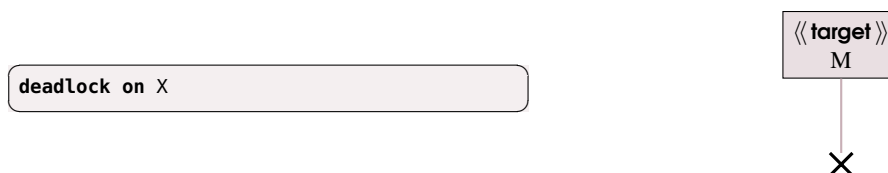
As a LifelineOccurrence, a WaitOccurrence has a *actor* on which we expect the wait to occur. This **must** be provided if, and only if, the occurrence is not inside a UntilFragment.

- R** The instant in time at which the occurrence takes effect is *after* the number of units. This has a subtle effect on how UntilFragments affect a WaitOccurrence: the fragment remains active until the wait has concluded. For instance, a wait for 3 units allows any communication allowed by the fragment to occur until the named actor has been inside the fragment for 3 units.

Differences from UML

WaitOccurrence does not directly correspond to a UML feature. We consider the resulting UML extension to be justified, as waits are a key *RoboChart* timing concept. To compensate, we encode WaitOccurrence in the graphical notation as if it were a gate message sending ‘wait’ to the World.

4.4.3 DeadlockOccurrence



A DeadlockOccurrence captures a deadlock scenario in a lifeline or UntilFragment. This is where part of the model no longer makes any progress, but has not successfully terminated.

Outside an UntilFragment, the DeadlockOccurrence takes a *actor* on which the deadlock is expected to happen. Inside an UntilFragment, there is no distinction between the actions of individual Actor^as, and so the DeadlockOccurrence may omit its *actor*.

Differences from UML

DeadlockOccurrences are an extension of UML to handle part of the semantics of *RoboChart*. The closest concept in UML, and the source of the graphical notation, is UML destruction messages; these represent the end of the life of an object.

4.5 Messages

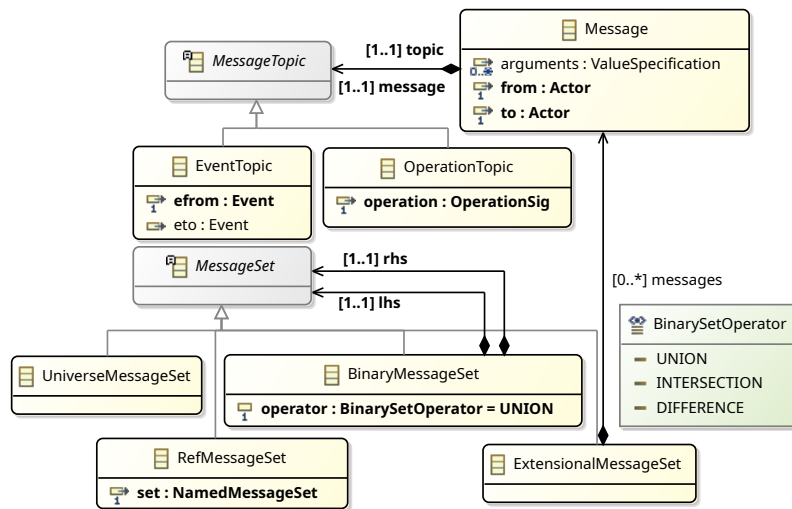


Figure 4.23: Class diagram for the part of the *RoboCert* metamodel dealing with messages.

Messages (fig. 4.23 capture *RoboChart* event and operation communications. In sequences, they appear within MessageSet^a s and $\text{MessageOccurrences}$.

4.5.1 MessageSet^a

```

universe // UniverseMessageSet
{ op 01(), op 02() } // ExtensionalMessageSet
op 01() // singleton ExtensionalMessageSet
set S // RefMessageSet
set X or set Y // UNION BinaryMessageSet
set X and set Y // INTER BinaryMessageSet
set X except set Y // DIFF BinaryMessageSet
  
```

A MessageSet^a expresses a set of Messages. There are four types:

- a $\text{UniverseMessageSet}$ represents the universal set containing all possible messages;
- an $\text{ExtensionalMessageSet}$ is a set (expressed as an unordered list) of zero or more Messages (§ 4.5.3);
- a RefMessageSet refers to a NamedMessageSet attached to the sequence group;
- a BinaryMessageSet is a binary operator over two other MessageSet^a s (operators are union, intersection, and difference).

4.5.2 NamedMessageSet

```

message set S = { op 01(), op 02 }
  
```

A `NamedMessageSet` attaches a name to a `MessageSet`^a, so that it can be placed inside a `SpecificationGroup` and referenced using `RefMessageSets`.

4.5.3 Message



A `Message` is a specification on the types of communication that can happen before or during an action. Each `Message` contains:

- the `MessageTopic`^a (§ 4.5.4) giving the type of communication that the spec is capturing;
- the `ValueSpecification`^as (§ 1.4.2) supplied to the communication;
- two `Actor`^as representing the endpoints of the connection: *from* and *to*.

Differences from UML

Messages broadly reflect the analogous concept in UML, but with adaptations to introduce the specific taxonomy of communications available in *RoboChart* (eg, `MessageTopic`^as). We do this to enable formal analysis of the communications of a model with respect to the Messages in its sequence diagrams.

We do not have reply Messages. This is because they do not fit in the *RoboChart* paradigm, except for in the case of calling controller operations (in which case, the *RoboCert* encoding is to inline the communications of the operation into the caller lifeline).

We do not have a distinction between synchronous and asynchronous Messages. This is a point that will be revisited in future work, as *RoboChart* models *do* have this distinction.

4.5.4 MessageTopic^a

```
operation 01() // OperationTopic
op 01()       // shorthand

event E      // EventTopic, eFrom=E, eTo undefined (effectively E)
event E1/E2  // EventTopic, eFrom=E1, eTo=E2
```

`MessageTopic`^as identify the form of communication in a `Message`. They do not directly identify the concrete *RoboChart* model part responsible for the communication, but specify it in a way that maps onto the existing UML concept of a message, and which is understandable to the user without in-depth knowledge of the naming conventions of the target model. Resolving `MessageTopic`^as into model elements therefore requires some analysis of the model. There are two types of topic:

- an `OperationTopic` identifies a *RoboChart* operation by its `OperationSig`^c;
- an `EventTopic` identifies a `Connection`^c by its endpoint `Event`^cs.

As `Connection`^cs do not have a name in *RoboChart*, an `EventTopic` is an indirect specification of its intended `Connection`^c. Each contains a mandatory event *efrom* (the event on the ‘from’ `Actor`^a)

and an optional event *eto* (the event on the ‘to’ Actor^Q). Each well-formed EventTopic corresponds to a Connection^{rc} as follows:⁵

- the EventTopic’s parent Message connects two Actor^Qs whose elements correspond to the ConnectionNode^{rc}s connected by the Connection^{rc};
- the *efrom* of the EventTopic corresponds to the *efrom* of the Connection^{rc} if it is unidirectional, or either the *efrom* or *eto* if bidirectional;
- the *eto* of the EventTopic, if specified, corresponds to the other Connection^{rc} event;
- the *eto* can be left unspecified, but only if both Connection^{rc} events share the same name.

R For the purpose of supplying ValueSpecification^Qs to a Message, an EventTopic has 0 parameters if its underlying Event^{rc}s are untyped, and 1 parameter otherwise (with the parameter taking that type).

4.6 Assertions

The *RoboCert* sequence language introduces properties into the *RoboCert* assertions language that allow verification over sequences.

4.6.1 SequenceProperty

```
assertion A: SequenceName holds           // positive 'holds' SequenceAssertion
assertion B: SequenceName does not hold   // negative 'holds' SequenceAssertion
assertion C: SequenceName is observed     // positive 'is observed' SequenceAssertion
assertion D: SequenceName is not observed // negative 'is observed' SequenceAssertion
```

A SequenceProperty is a Property^Q about a particular Interaction with respect to its Target^Q.

The specific sequence assertion type comes from the SequenceAssertionType: either ‘sequence holds on target’ (refinement), or ‘sequence is observed on target’ (reverse refinement). The assertion can be negated. The choice of SemanticModel (§ 1.5.4) affects how we check the assertion.

⁵See § 5.4.4 for the formal condition definitions.

5. Sequences: well-formedness

This section gives well-formedness conditions for the metamodel in chapter 4. See § 3.1 for an introduction to the conventions used in well-formedness condition chapters.

5.1 Sequences

This section contains well-formedness conditions for the classes defined in § 4.2.

5.1.1 Actor^a (§ 4.2.1)

There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses.

5.1.2 TargetActor

There are no well-formedness conditions for this class.

5.1.3 ComponentActor: SAc

Feature *component*: SAcC

SAcC1 The *component* of a ComponentActor **must** name a model component contained within the subcomponents of the Target^a its enclosing CertPackage.



With **SG1** ensuring that the Target^a does indeed have subcomponents, this ensures that actors specify the behaviour of subcomponents *inside* the element of the Target^a.

5.1.4 World

There are no well-formedness conditions for this class.


5.1.5 Interaction (§ 4.2.2)

Feature *group*

There are no well-formedness conditions for this feature.


Feature *variables*: SIV

SIV1 The *variables* of an Interaction **must** have the **VAR** modifier.


-  Constants make no sense; these variables exist to produce a memory for the interaction.

Feature *actors*: SIA

SIA1 The *actors* of an Interaction **must** be distinct.

-  Having more than one copy of each actor introduces needless redundancy and ambiguity.

SIA2 The *actors* of an Interaction **must** each be present on the enclosing SpecificationGroup.

-  This prevents message sets and other group-defined elements from disagreeing with the sequence on the definition of Actor^as.

Feature *fragments*

There are no well-formedness conditions for this feature.

5.2 Fragments

This section contains well-formedness conditions for the classes defined in § 4.3.

5.2.1 InteractionFragment^a (§ 4.3)

There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses.

5.2.2 OccurrenceFragment (§ 4.3.1)


There are no well-formedness conditions for this class.

5.2.3 CombinedFragment^a

There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses.


5.2.4 BlockFragment^a (§ 4.3.2): SBL**Feature *body*: SBLB**

SBLB1 The *body* of a BlockFragment^a **must not** have an ElseGuard.

-  ElseGuards are the negation of the disjunction of all other guards, so the guard would always be false.

5.2.5 DeadlineFragment (§ 4.3.3): SD**Feature *actor*: SDA**

SDA1 The *actor* of a DeadlineFragment **must not** be a World.

-  The semantics of the fragment is a constraint over the actions on the Actor^a's lifeline; a World has no lifeline in the semantics, and so such a constraint would have no meaning.

Feature *units*

There are no well-formedness conditions for this feature.

5.2.6 LoopFragment (§ 4.3.4)

There are no well-formedness conditions for this class.

5.2.7 OptFragment (§ 4.3.5)

There are no well-formedness conditions for this class.


5.2.8 UntilFragment (§ 4.3.6: SU)

Feature *intraMessages*

There are no well-formedness conditions for this feature.

Feature *body*: SUB


SDbL1 The *body* of a UntilFragment **must not** contain another UntilFragment.

-  The body of such a fragment is intended to be the trigger condition for exiting the ‘any in set’ behaviour of the fragment. Having another UntilFragment in such a trigger does not make sense.


5.2.9 DiscreteBound (§ 4.3.7): SDb

Feature *lower*: SDbL


SDbL1 The *lower* of a DiscreteBound **must** be a natural, if present.

-  Type safety. Other types make no sense here, as this construct represents the number of times something may occur.


SDbL2 Variables with modifier *VAR* referenced in the *lower* of a DiscreteBound **must** belong to an enclosing Interaction.

-  No other variables are visible to such expressions in *RoboCert* version 0.1. Reading values from model variables is not yet supported.

SDbL3 Variables with modifier *CONST* referenced in the *lower* of a DiscreteBound **must** belong to the parameterisation of the Target^a of an enclosing SpecificationGroup.


-  No other constants are visible to such expressions.

SDbL4 The *lower* of a DiscreteBound **must** be present if the *upper* is absent.


-  A bound with no ends specified is meaningless. Any constructs using DiscreteBounds that support unboundedness do so by making the DiscreteBound itself optional.

Feature *upper*: SDbU


SDbU1 The *upper* of a DiscreteBound **must** be a natural, if present.

-  Type safety. Other types make no sense here, as this construct represents the number of times something may occur.


SDbU2 Variables with modifier *VAR* referenced in the *upper* of a DiscreteBound **must** belong to an enclosing Interaction.

-  No other variables are visible to such expressions in *RoboCert* version 0.1. Reading values from model variables is not yet supported.

SdbU3 Variables with modifier *CONST* referenced in the *upper* of a *DiscreteBound* **must** belong to the parameterisation of the *Target*^a of an enclosing *SpecificationGroup*.

 No other constants are visible to such expressions.


SdbU4 The *upper* of a *DiscreteBound* **must** be present if the *lower* is absent.

 A bound with no ends specified is meaningless. Any constructs using *DiscreteBounds* that support unboundedness do so by making the *DiscreteBound* itself optional.


5.2.10 *BranchFragment*^a (§ 4.3.8): *SBr*

Feature *branches*: *SBrB*

SBrB1 The *branches* of a *BranchFragment*^a **must not** directly contain more than one *ElseGuard*.

 *ElseGuards* are the negation of the disjunction of all other guards, so the presence of another *ElseGuard* would create a recursive situation whose meaning would be unclear.

SBrB2 The *branches* of a *BranchFragment*^a **must not** directly contain both an *ElseGuard* and an *EmptyGuard*.

 The *ElseGuard* would always evaluate to false, making the branch purposeless.

5.2.11 *AltFragment* (§ 4.3.9)

There are no well-formedness conditions for this class.

5.2.12 *XAltFragment* (§ 4.3.10)

There are no well-formedness conditions for this class.

5.2.13 *ParFragment* (§ 4.3.11)

There are no well-formedness conditions for this class.

5.2.14 *InteractionOperand* (§ 4.3.12)

There are no well-formedness conditions for this class.

5.2.15 *Guard*^a (§ 4.3.13)

Well-formedness conditions for guards proceed per subclass.

5.2.16 *EmptyGuard*

There are no well-formedness conditions for this class.

5.2.17 *ExprGuard*: *SGe*


An *ExprGuard* must have an expression that is of boolean type.

Feature *expr*: *SGeE*


SGeE1 The *expr* of an *ExprGuard* **must** be of Boolean type.

 Type safety.

SGeE2 Variables with modifier *VAR* referenced in the *expr* of a ExprGuard **must** belong to an enclosing Interaction.

 No other variables are visible to such expressions in *RoboCert* version 0.1. Reading values from model variables is not yet supported.

SGeE3 Variables with modifier *CONST* referenced in the *expr* of a ExprGuard **must** belong to the parameterisation of the Target^a of an enclosing SpecificationGroup.

 No other constants are visible to such expressions.

5.2.18 ElseGuard

There are no well-formedness conditions for this class.

5.3 Occurrences

This section contains well-formedness conditions for the classes defined in § 4.4.

5.3.1 Occurrence^a (§ 4.4)

There are no well-formedness conditions for this class, but there may be well-formedness conditions on its subclasses.


5.3.2 MessageOccurrence (§ 4.4.1)

There are no well-formedness conditions for this class.

5.3.3 LifelineOccurrence: SLo

Feature *actor*: SLoA


SLoA1 The *actor* of a LifelineOccurrence **must not** be a World.

 The semantics of the fragment is an action on the Actor^a's lifeline; a World has no lifeline in the semantics, and so such a constraint would have no meaning.

5.3.4 WaitOccurrence (§ 4.4.2): SW

Feature *units*: SWU

SWU1 The *units* of a WaitOccurrence **must** be a natural number.

 A negative *units* amount would require the implementation to travel backwards in time; further, the *RoboChart* model of time is discrete.

5.3.5 DeadlockOccurrence (§ 4.4.3)

There are no well-formedness conditions for this class.

5.4 Messages

This section contains well-formedness conditions for the classes defined in § 4.5.


5.4.1 MessageSet^a (§ 4.5.1)

There are no well-formedness conditions for this class or its subclasses.

5.4.2 NamedMessageSet (§ 4.5.2)

Feature set: SSnS


SMTp1 The *set* of a NamedMessageSet **must not** reference the same NamedMessageSet, either directly or through expansion of other references.¹

 This would cause an infinitely recursive expansion of the message set contents.


5.4.3 Message (§ 4.5.3)

Feature topic: SMTp


SMTp1 A Message with an OperationTopic *topic* **must** reference an operation defined inside the robotic platform.

 This aids in preventing currently-undefined situations where state machine lifelines call into operations defined on the controller. Future revisions of *RoboCert* may weaken or remove this condition as its validation requires non-modular knowledge of the context of the Target^a.

SMTp2 A Message with an EventTopic *topic* **must** correspond to a Connection^{rc} from *from* to *to*, and with the same *efrom* and (if present) *eto*. If the Connection^{rc} is bidirectional, the Message **may** match the inverse of the connection.


 This allows us to resolve a loose specification of what the intended event communication is to a specific but unnamed part of the *RoboChart* model. This also pulls in the well-formedness conditions of *RoboChart* connections.

SMTp3 A Message with an EventTopic *topic* **must** have an *eto* if the corresponding event of its Connection^{rc} has a different name from that of the event corresponding to the *efrom*.


 This makes users be explicit about the events where there is a difference in name, while leaving the one-event option as a convenience for the typical *RoboChart* idiom of both events being named in the same way.

Feature arguments: SMA

SMA1 The *arguments* of a Message **must** have exactly as many elements as its *topic* has parameters.²


 This ensures all arguments are well-specified in a message, even if by supplying WildcardValueSpecifications.

SMA2 The *arguments* of a Message **must** be type-compatible with their corresponding parameters.

 Type safety. Note that the type of an ExpressionValueSpecification is the type of its underlying expression, the type of a bound WildcardValueSpecification is that of its underlying variable, and unbound WildcardValueSpecifications have no fixed type.

Feature from: SMF


SMF1 The *from* of a Message **must not** be equal to the *to* of the Message.

 There is no situation in *RoboChart* where elements can send messages to themselves.

¹In other words, there must be no cycles in NamedMessageSet definitions.


²Here, an EventTopic is considered to have 1 parameter if typed, and 0 otherwise; said parameter has the type of the event and an arbitrary name.

SMF2 The *from* of a Message with an OperationTopic *topic* **must** correspond to a model element that requires the *topic*'s *operation*.

-  This establishes that the operation can be called from its callee; it also binds the operation to the well-formedness conditions of *RoboChart* operations by stating that a definition for the operation must exist.

Feature to: SMT

SMT1 The *to* of a Message with an OperationTopic **must** be a World.

-  In *RoboCert* version 0.1, operation calls that do not go from inner components to outer components are ill-formed. As *RoboChart* permits operation calls from state machines of controllers to operations defined in controllers, any extension to *RoboCert* that allows the specification of operations as lifelines will relax or remove this condition.

5.4.4 MessageTopic^a (§ 4.5.4)

There are no well-formedness conditions for this class.³

5.4.5 EventTopic

There are no well-formedness conditions for this class.

5.4.6 OperationTopic

There are no well-formedness conditions for this class.

5.5 Assertions

There are no rules for the definitions in § 4.6.

³All conditions over topics also take the rest of the message as context, and are therefore defined over messages.

6. Sequences: textual syntax

This chapter describes the textual syntax of *RoboCert* sequence diagrams. See chapter 2 for discussion on the notation and conventions used here.

6.1 Interactions

6.1.1 Actor^a

$\langle Actor^a \rangle ::= \langle World \rangle \mid \langle TargetActor \rangle \mid \langle ComponentActor \rangle$

$\langle World \rangle ::= \text{'world'}$

$\langle TargetActor \rangle ::= \text{'target'}$

$\langle ComponentActor \rangle ::= \text{'component' QUALIFIED-NAME}$

6.1.2 Interaction

The syntax for actors is an extension of that permitted by Mermaid.

$\langle Interaction \rangle ::= \text{'sequence' NAME} \rightarrow \langle intVars \rangle? \langle intActors \rangle + \langle InteractionFragment^a \rangle + \leftarrow$

$\langle intVars \rangle ::= \text{'var' } \langle Variable^{rc} \rangle (', ' \langle Variable^{rc} \rangle)^* (', '? \text{'and' } \langle Variable^{rc} \rangle)?$

$\langle intActors \rangle ::= (\text{'actor' } \mid \text{'actors'}) \text{NAME} (', ' \text{NAME})^* (', '? \text{'and' } \text{NAME})?$

6.2 Fragments

$\langle InteractionFragment^a \rangle ::= \langle BlockFragment^a \rangle \mid \langle BranchFragment^a \rangle \mid \langle Occurrence^a \rangle$

6.2.1 BlockFragment^a

Here, the distinction between types of fragment occurs in the *header* of the block, while a common rule collects the body of the fragment.

A block operand is either an inline singleton fragment (denoted by the presence of a colon), or a whitespace-delimited fragment list.

$$\langle \text{BlockFragment}^a \rangle ::= \langle \text{blockHeader} \rangle \text{NAME? } \langle \text{blockOperand} \rangle$$

$$\begin{aligned} \langle \text{blockHeader} \rangle &::= \langle \text{DeadlineFragment} \rangle \\ &| \langle \text{LoopFragment} \rangle \\ &| \langle \text{OptFragment} \rangle \\ &| \langle \text{UntilFragment} \rangle \end{aligned}$$

6.2.2 DeadlineFragment

$$\langle \text{DeadlineFragment} \rangle ::= \text{'deadline' } ((\langle \text{Expression}^c \rangle (\text{'unit' } | \text{'units' })?)) \text{'on' NAME}$$

6.2.3 LoopFragment

$$\langle \text{LoopFragment} \rangle ::= \text{'loop' } ((\langle \text{DiscreteBound} \rangle (\text{'time' } | \text{'times' })?))?$$

6.2.4 OptFragment

$$\langle \text{AltFragment} \rangle ::= \text{'opt'}$$

6.2.5 UntilFragment

When an intra-message set begins with ‘except’, read it as if there was an implicit ‘in universe’ preceding the ‘except’. ‘block until’ is sugar for ‘any in {} until’.

$$\langle \text{UntilFragment} \rangle ::= (\text{'block' } | (\text{'any' } | \text{'anything' }) \langle \text{intraMessageSet} \rangle) \text{'until'}$$

$$\langle \text{intraMessageSet} \rangle ::= \text{'in' } \langle \text{MessageSet}^a \rangle | \text{'except' } \langle \text{MessageSet}^a \rangle$$

6.2.6 BranchFragment^a

Unlike BlockFragment^as, the syntax of BranchFragment^as cannot be split cleanly into varying headers and common bodies. There are two general types of syntax: one for alternatives, and one for parallel composition.

$$\langle \text{BranchFragment}^a \rangle := \langle \text{altOrXAltFragment} \rangle | \langle \text{ParFragment} \rangle$$

$$\langle \text{altOrXAltFragment} \rangle ::= \langle \text{altOrXAlt} \rangle \text{NAME? } \langle \text{altOrXAltBranches} \rangle \text{'end'}$$

$$\langle \text{altOrXAlt} \rangle := \langle \text{AltFragment} \rangle | \langle \text{XAltFragment} \rangle$$

$$\langle \text{altOrXAltBranches} \rangle ::= \langle \text{branchOperand} \rangle (\text{'else' } \langle \text{branchOperand} \rangle)^+$$

6.2.7 AltFragment

This is a header; see $\langle \text{altOrXAltFragment} \rangle$ above for the full syntax.

$$\langle \text{AltFragment} \rangle ::= \text{'alt' } | \text{'provisional' }? \text{'alternative'}$$

6.2.8 XAltFragment

Similar to AltFragment.

$$\langle \text{AltFragment} \rangle ::= \text{'xalt' } | \text{'mandatory' } \text{'alternative'}$$

6.2.9 ParFragment

$$\langle \text{ParFragment} \rangle ::= (\text{'par' } | \text{'parallel'}) \text{ NAME? } \langle \text{parBranches} \rangle \text{'end'}$$

$$\langle \text{parBranches} \rangle ::= \langle \text{branchOperand} \rangle (\text{'and' } \langle \text{branchOperand} \rangle)^+$$
6.2.10 InteractionOperand

There is no single rule for interaction operands, as their syntax depends on whether they arise in branch or block position. The only difference is that the latter includes a terminating ‘end’.

An elided guard desugars to ‘[always]’.

$$\begin{aligned} \langle \text{blockOperand} \rangle &::= \text{' : ' } \langle \text{Guard}^a \rangle? (\langle \text{InteractionFragment}^a \rangle | \text{'nothing'}) \\ &| \langle \text{Guard}^a \rangle (\rightarrow \langle \text{InteractionFragment}^a \rangle \leftarrow \text{'end' } | \text{'nothing'}) \end{aligned}$$

$$\begin{aligned} \langle \text{branchOperand} \rangle &::= \text{' : ' } \langle \text{Guard}^a \rangle? (\langle \text{InteractionFragment}^a \rangle | \text{'nothing'}) \\ &| \langle \text{Guard}^a \rangle (\rightarrow \langle \text{InteractionFragment}^a \rangle \leftarrow | \text{'nothing'}) \end{aligned}$$
6.2.11 Guard^a

$$\langle \text{Guard}^a \rangle ::= \text{'[' } (\langle \text{EmptyGuard} \rangle | \langle \text{ExprGuard} \rangle | \langle \text{ElseGuard} \rangle) \text{']'}$$

$$\langle \text{EmptyGuard} \rangle ::= \text{'always'}$$

$$\langle \text{ExprGuard} \rangle ::= \langle \text{Expression}^{\text{C}} \rangle$$

$$\langle \text{ElseGuard} \rangle ::= \text{'otherwise'}$$
6.3 Occurrences

LifelineOccurrences take the name of their bound Actor^a.

$$\langle \text{Occurrence}^a \rangle ::= \langle \text{MessageOccurrence} \rangle | \langle \text{LifelineOccurrence} \rangle$$

$$\langle \text{LifelineOccurrence} \rangle ::= (\langle \text{WaitOccurrence} \rangle | \langle \text{DeadlockOccurrence} \rangle) \text{'on' NAME}$$
6.3.1 MessageOccurrence

A missing *temperature* desugars to ‘cold’.

$$\langle \text{MessageOccurrence} \rangle ::= \langle \text{Message} \rangle \langle \text{Temperature} \rangle?$$

$$\langle \text{Temperature} \rangle ::= \text{'hot' } | \text{'cold'}$$
6.3.2 WaitOccurrence

We capture the actor name in the rule for LifelineOccurrence.

$$\langle \text{WaitOccurrence} \rangle ::= \text{'wait' } \text{'(' } \langle \text{Expression}^{\text{C}} \rangle (\text{'unit' } | \text{'units'})? \text{'('})$$
6.3.3 DeadlockOccurrence

As above.

$$\langle \text{DeadlockOccurrence} \rangle ::= \text{'deadlock'}$$

6.4 Messages

6.4.1 MessageSet^a

The rules for message sets are factored for precedence and associativity. The first two rules produce BinaryMessageSets in their second productions; the first handles a **DIFFERENCE**; the next two handle **UNION** and **INTERSECTION** respectively.

```

<MessageSeta> ::= <unionOrInterSet>
  | <MessageSeta> 'except' <unionOrInterSet>

<unionOrInterSet> ::= <primitiveSet>
  | <unionOrInterSet> <unionOrInterOperator> <primitiveSet>

<unionOrInterOperator> ::= 'or' | 'and'

<primitiveSet> ::= '(' <MessageSeta> ')'
  | <UniverseMessageSet>
  | <ExtensionalMessageSet>
  | <RefMessageSet>

<UniverseMessageSet> ::= 'universe'

<ExtensionalMessageSet> ::= '{' (<Message> (',' <Message>))* '}'

<RefMessageSet> ::= 'message'? 'set' NAME

```

6.4.2 NamedMessageSet

The requirement to specify 'message' here, which is optional above, is intentional. Named message sets appear in specification groups, where it is important to distinguish exactly what the type of

```

<NamedMessageSet> ::= 'message' 'set' NAME '=' <MessageSeta>

```

6.4.3 Message

In the edge, the first NAME names a from-Actor^a; the second names a to-Actor^a.

Argument lists can be populated, empty, or elided (the latter two cases are equivalent).

```

<Message> ::= <edge> ':' <MessageTopica> <messageArguments>?

<edge> ::= NAME '->' NAME

<messageArguments> ::= '(' (<ValueSpecificationa> (',' <ValueSpecificationa>))* ')'

```

6.4.4 MessageTopic^a

```

<MessageTopica> ::= <EventTopic> | <OperationTopic>

<EventTopic> ::= 'event' NAME ('/' NAME)?

<OperationTopic> ::= ('op' | 'operation') NAME

```

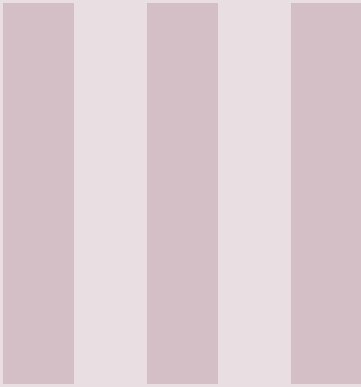

6.5 Assertions

The production for `SequenceAssertionType` also governs whether the `SequenceProperty` is negated.

$\langle \text{SequenceProperty} \rangle ::= \langle \text{QualifiedName} \rangle \langle \text{SequencePropertyType} \rangle \langle \text{modelStanza} \rangle ?$

$\langle \text{SequenceAssertionType} \rangle ::= \text{'holds' | 'does' 'not' 'hold' | 'is' 'not'? 'observed'}$

$\langle \text{modelStanza} \rangle ::= \text{'in' 'the' } \langle \text{SemanticModel} \rangle \text{'model'}$



Low-Level Language Interoperability

7	CSP_M	61
7.1	Metamodel	
7.2	Well-formedness conditions	

RoboCert contains features for directly including ‘fragments’ of low-level assertion code (for instance, CSP-M) into textual scripts. These notations exist for testing, debugging, and expert use, and do not have strong guarantees of stability or forward-compatibility.

RoboCert version 0.1 contains CSP-M support; support for other languages will follow later on.

7. CSP_M

This chapter discusses the metamodel and notation for including CSP_M fragments into CertPackages.

In CSP_M, the **TRACES** semantic model corresponds to trace refinement with maximal progress (τ prioritised over *tock*), and the **TIMED** semantic model corresponds to *tick-tock* refinement ([6]).

7.1 Metamodel

Here, we define the *RoboCert* metamodel for CSP_M support.

7.1.1 CSPGroup

```
csp <$ print "Hello, there" $>
```

A CSPGroup is a Group^a enclosing an unprocessed fragment of CSP_M, *csp*. This fragment is inserted verbatim into any CSP_M generated from the CertPackage.

R CSPGroups are opaque; nothing defined inside will be available for use at the *RoboCert* level.

7.1.2 CSPProperty

```
assertion Example1: csp <$ Proc1 [T= Proc2 $>  
assertion Example2: csp negated <$ Proc1 [T= Proc2 $>
```

A CSPProperty lifts a CSP fragment into a Property^a. The CSP_M within becomes the body of an ‘assert’ or ‘assert not’ CSP_M directive, depending on whether the property is negated.

R CSPProperty instances, like CSPGroups, are also opaque.

7.2 Well-formedness conditions

This section gives well-formedness conditions for the metamodel above. See § 3.1 for an introduction to the conventions used in well-formedness condition chapters.

7.2.1 CSPGroup (§ 7.1.1)

There are no well-formedness conditions for this class.

7.2.2 CSPProperty (§ 7.1.2)

There are no well-formedness conditions for this class.

IV

Semantics

8	Introduction	65
8.1	How to read these chapters	
9	General Definitions	67
9.1	Core language	
9.2	Sequence notation	
10	Timed Semantics: <i>tock</i>-CSP	71
10.1	Relationship to the generator	
10.2	Note to the reader	
10.3	Dependencies on the <i>RoboChart</i> semantics	
10.4	Core language	
10.5	Sequence notation	
A	Language changelog	85
A.1	This draft	
A.2	Version 0.1 (2022-05-20)	

8. Introduction

This part of the manual formally captures the semantics of *RoboCert* in terms of its target languages. In *RoboCert* version 0.1, this is *tock*-CSP (chapter 10); semantics for other target languages will come in future work.

Each semantics captures Assertions as the top-level definition, with all objects reachable from the assertions translated in-line. As a consequence, we do not capture organisational details such as CertPackages, or any distinction between references to objects and their definitions.

8.1 How to read these chapters

The semantics treatments in this part take the form of rewrite rules from the *RoboCert* metamodels to some object language (for instance, *tock*-CSP). For conciseness, we use a meta-language based on the Z notation. We also use the following notational conventions:

- $\llbracket - \rrbracket^{\text{name}}$ denotes a main semantic rule;
- `name()` denotes auxiliary semantic functions;
- `x.name` denotes a field of the metamodel object `x`;
- underlined grey text denotes a construct from the meta-language; normal mathematical text constitutes the object language;
- in function types, the suffix `?` denotes an object that may be absent (for instance, the value of an optional feature). Let \perp represent the absence of a value of such a type. As a notational convenience, we assume that any semantic function on type T lifts to type $T?$ such that, unless otherwise defined, the application of the function on \perp returns \perp .

9. General Definitions

This chapter contains functions and rewrite rules that are agnostic to the particular target language.

9.1 Core language

This section provides helper definitions for capturing the core semantics of *RoboCert*.

9.1.1 Values (§ 1.4)

These definitions provide support for transforming or extracting information from expressions and value specifications.

Definition 9.1.1 — Expression variables. Function `eVars` is defined as the depth-first traversal of all *RoboCert*-level non-constant variables in an expression.

$$\text{eVars} : \text{Expression}^{\text{rc}} \rightarrow \text{seq Variable}^{\text{rc}}$$

This function is not concretely defined here; this may change in subsequent manual revisions.

Definition 9.1.2 — Bound variables of a ValueSpecification^a. Function `vBind` collects the variables bound in a `ValueSpecificationa`.

$$\text{vBind} : \text{ValueSpecification}^{\text{a}} \rightarrow \text{seq Variable}^{\text{rc}}$$

Case analysis of the value specification.

$$\begin{aligned} \underline{\text{vBind}(x : \text{WildcardValueSpecification})} &\stackrel{\text{def}}{=} \underline{\langle x : \langle x.\text{destination} \rangle \bullet x \neq \perp \rangle} \\ \underline{\text{vBind}(x : \text{ExpressionValueSpecification})} &\stackrel{\text{def}}{=} \langle \rangle \end{aligned}$$

9.2 Sequence notation

This section provides helper definitions for capturing the semantics of *RoboCert* sequence diagrams.

9.2.1 Interactions (§ 4.2.2)

These definitions support the semantics of Interactions.

Definition 9.2.1 — Namespace of an actor. Function actNs resolves a Actor^a to a CSP namespace^a under which the connection endpoints and callable operations of the Actor^a appear as channels.

$$\text{actNs} : \text{Actor}^a \rightarrow \text{Namespace}$$

This function is not concretely defined here; this may change in subsequent manual revisions.

^aStrictly speaking, namespaces are a CSP_M extension to CSP; however, we adopt them in this semantics as the formal semantics of *RoboChart* elements uses them. We can model them as a prefix on CSP channel names.

9.2.2 Fragments (§ 4.3)

These definitions support the semantics of $\text{InteractionFragment}^a$ s.

Definition 9.2.2 — Fragment list. Function flist yields a set of pairs (i, f) such that every fragment of the given metaclass maps to precisely one f and each i is unique (typically this would reflect the order of depth-first traversal of the object graph).

$$\text{flist} : \text{Interaction} \rightarrow \text{Class} \rightarrow \mathbb{P}(\mathbb{N} \times \text{InteractionFragment}^a)$$

This function is not concretely defined here; this may change in subsequent manual revisions.

Definition 9.2.3 — Fragment index. Function findex is defined such that $\text{findex}(f, c) = i$ where, for the parent Interaction s , $(i, f) \in \text{flist}(s, c)$.

$$\text{findex} : \text{InteractionFragment}^a \rightarrow \text{Class} \rightarrow \mathbb{N}$$

This function is not concretely defined here; this may change in subsequent manual revisions.

Definition 9.2.4 — Fragment expressions. Function fexprs collects expressions *directly*^a contained within a $\text{InteractionFragment}^a$.

$$\text{fexprs} : \text{InteractionFragment}^a \rightarrow \text{seq Expression}^c$$

We define exprs as a combination of the guard expressions of any operands (fops) within the fragment, as well as any expressions unique to the type of fragment being considered; we delegate these to an auxiliary function fdexprs .

$$\text{fexprs}(x) \stackrel{\text{def}}{=} \text{fdexprs}(x) \cap ((o : \text{fops}(x) \mid o.\text{guard} \in \text{ExprGuard} \bullet o.\text{guard}.\text{expr}))$$

We define fdexprs below.

$$\text{fdexprs} : \text{InteractionFragment}^a \rightarrow \text{seq Expression}^c$$

For occurrences, we delegate to oexprs ; for loops, we take the expressions contained within the

loop bound (bexprs; def. 9.2.6); for everything else, this is the empty sequence.

$$\begin{aligned} \underline{\text{fdexprs}(x : \text{OccurrenceFragment})} &\stackrel{\text{def}}{=} \underline{\text{oexprs}(x.\text{occurrence})} \\ \underline{\text{fdexprs}(x : \text{LoopFragment})} &\stackrel{\text{def}}{=} \underline{\text{bexprs}(x.\text{bound})} \\ \underline{\text{fdexprs}(x : \text{InteractionFragment}^a)} &\stackrel{\text{def}}{=} \underline{\langle \rangle} \quad (\text{default}) \end{aligned}$$

^aIn other words, fexprs does not recurse.

Definition 9.2.5 — Fragment operands. Function fops collects InteractionOperands contained within a InteractionFragment^a.

$$\text{fops} : \text{InteractionFragment}^a \rightarrow \text{seq InteractionOperand}$$

Case analysis on the type of fragment:

$$\begin{aligned} \underline{\text{fops}(x : \text{OccurrenceFragment})} &\stackrel{\text{def}}{=} \underline{\langle \rangle} \\ \underline{\text{fops}(x : \text{BranchFragment}^a)} &\stackrel{\text{def}}{=} \underline{x.\text{branches}} \\ \underline{\text{fops}(x : \text{BlockFragment}^a)} &\stackrel{\text{def}}{=} \underline{\langle x.\text{body} \rangle} \end{aligned}$$

Definition 9.2.6 — Bound expressions. The expressions of a DiscreteBound are just the upper and lower bound expressions, excluding any undefined expressions.

$$\begin{aligned} \text{bexprs} : \text{DiscreteBound} &\rightarrow \text{seq Expression}^{\text{rc}} \\ \text{bexprs}(b) &\stackrel{\text{def}}{=} \langle x : \langle b.\text{lower}, b.\text{upper} \rangle \mid x \neq \perp \rangle \end{aligned}$$

9.2.3 Occurrences (§ 4.4)

These definitions support the semantics of Occurrence^as.

Definition 9.2.7 — Relevant actors. The *relevant actors* oactors(*o*) of a Occurrence^a *o* is the set of Actor^as for which the Occurrence^a should emit an effect. When considering the semantics of a lifeline, any Occurrence^as not including that lifeline in the relevant actors become *Skip*.

$$\text{oactors} : \text{Occurrence}^a \rightarrow \mathbb{P}(\text{Actor}^a)$$

Case analysis on the type of occurrence (capturing all lifeline-bound occurrences with one rule):

$$\begin{aligned} \underline{\text{oactors}(x : \text{LifelineOccurrence})} &\stackrel{\text{def}}{=} \underline{\{x.\text{actor}\}} \\ \underline{\text{oactors}(x : \text{MessageOccurrence})} &\stackrel{\text{def}}{=} \underline{\{a : \{x.\text{message.from}, x.\text{message.to}\} \mid a \notin \text{World}\}} \end{aligned}$$

Definition 9.2.8 — Occurrence expressions. Function oexprs finds expressions in an Occurrence^a.

$$\text{oexprs} : \text{Occurrence}^a \rightarrow \text{seq Expression}^{\text{rc}}$$

Case analysis on the type of occurrence. Deadlock occurrences have no expressions. Wait occurrences yield their delay units. Message occurrences yield every expression that is *directly*

contained inside an argument.

$$\begin{aligned}
 \underline{\text{oexprs}(x : \text{DeadlockOccurrence})} &\stackrel{\text{def}}{=} \langle \rangle \\
 \underline{\text{oexprs}(x : \text{WaitOccurrence})} &\stackrel{\text{def}}{=} \langle x.\text{units} \rangle \\
 \underline{\text{oexprs}(x : \text{MessageOccurrence})} &\stackrel{\text{def}}{=} \bigwedge / \langle a : x.\text{message.arguments} \\
 &\quad | a \in \text{ExpressionValueSpecification} \bullet a.\text{expr} \rangle
 \end{aligned}$$

9.2.4 Messages (§ 4.5)

These definitions support the semantics of Messages.

Definition 9.2.9 — Connection of an event. Function eConn takes an EventTopic and a pair of Actor^as capturing the ends of its encompassing Message, and yields the underlying Connection^c. This function is total provided that neither Actor^a is a World and that every inter-component event Message has an underlying Connection^c; the latter is a well-formedness condition (see § 5.4).

$$\text{eConn} : \text{EventTopic} \rightarrow (\text{Actor}^a \times \text{Actor}^a) \rightarrow \text{Connection}^c$$

This function is not concretely defined here; this may change in subsequent manual revisions.

10. Timed Semantics: *tock*-CSP

This chapter introduces a *tock*-CSP semantics for *RoboCert*. As the original target notation, it serves as the most fully-realised form of the *RoboCert* semantics.

We assume the rules of chapter 9 are in scope, including imports from the *RoboChart* semantics.

10.1 Relationship to the generator

The behaviour of the *RoboCert* generator shall conform to this semantics, except that the generator:

- **may**, where CSP constructs are missing or not idiomatic, substitute semantically equivalent CSP_M constructs;
- **must** wrap processes in timed sections and prioritisations to achieve the appropriate *tock*-CSP behaviours of CSP operators;
- **may** perform semantics-preserving optimisations, such as substituting \underline{P} for $\text{Stop} \triangle \underline{P}$.

There is, in principle, a 1:1 mapping from rules in this chapter to rules in the generator. The generator is, however, a Java library, and so its native object-oriented paradigm does not map directly to the functional, mathematical style used throughout this chapter.

10.2 Note to the reader

As of time of writing, this section is still being written. Some semantic functions are missing or incomplete. Where parts of the semantics are missing, the authority on the semantics of *RoboCert* remains its implementation in the generator.

While semantic rules and functions are missing in this report, a snapshot of the generator source code will be available at <https://github.com/UoY-RoboStar/robocert-evaluation/raw/main/generator-src.zip>. Discrepancies between the behaviour of this generator and the rules in this report should be considered to be bugs.

10.3 Dependencies on the *RoboChart* semantics

Each semantics treatment in this section assumes the existence of a compatible semantics for *RoboChart*. Specifically, we assume the following rules and functions are available, or can be derived, from such a semantics:

- let $\text{targetProcess}(-)$ map a target to the parametric process exposed by the relevant *tock*-CSP semantics (for instance, we delegate to the *RoboChart* semantics for the underlying $\text{RCModule}^{\text{rc}}$ of a ModuleTarget);
- let $\text{targetParams}(-)$ map a target to the sequence of constants in its parameterisation;
- let $\text{constName}(-)$ map a constant to its name in the *RoboChart* instantiations file;
- let $\llbracket - \rrbracket^{\text{type}}$ map a *RoboChart* type to the set of CSP values that inhabit it.

10.4 Core language

These rules implement the semantics for the core language of *RoboCert* in *tock*-CSP.

10.4.1 Top-level (§ 1.2)

The top level definition for the *RoboCert* semantics as a whole is **Assertion** (§ 10.4.4). There is no semantics for groups and packages, as they are purely organisational.

10.4.2 Targets (§ 1.3)

We appeal to the semantics of the underlying *RoboChart* model components of targets, and therefore do not give them a separate semantics.¹

10.4.3 Values (§ 1.4)

These rules concern values and expressions.

Expressions

These rules concern *RoboChart* expressions.

Definition 10.4.1 — Expression context. Let ExprContext^a be a pseudo-metaclass capturing anything that can serve as a context for expression evaluation. In *RoboCert* version 0.1, this is *Interaction* or anything nested inside an *Interaction*, but future versions will add more contexts.

Rule 1 — Expression (§ 1.4.1). Rule $\llbracket x \rrbracket_c^{\text{expr}}$ is the expression semantics of x in the scope of context c . The expression semantics is largely that of *RoboChart* which, in turn, is largely that of Z . One distinction is that we resolve non-constant variables at the *RoboCert* level; for instance, inside a *Interaction* such variables resolve to those attached to the *Interaction*.

$$\llbracket - \rrbracket_-^{\text{expr}} : \text{Expression}^{\text{rc}} \rightarrow \text{ExprContext}^a \rightarrow \underline{\text{Value}}$$

This rule is not concretely defined here; this may change in subsequent manual revisions.

We can interpret $\text{ValueSpecification}^a$ s in three ways. The first ($\llbracket - \rrbracket_-^{\text{val}}$, rule 2), used in *MessageOccurrences*, is the definitive semantics of a value specification as a CSP channel extension. The second ($\llbracket - \rrbracket_-^{\text{svol}}$, rule 3), used in *MessageSet*^as, is identical to $\llbracket - \rrbracket_-^{\text{val}}$ except that it captures wildcards as bindings to an external set comprehension. and involves rule $\llbracket - \rrbracket_-^{\text{val}}$. The third, also used in *MessageSet*^as, captures all possible values of a value specification as a set; it involves function expVal (def. 10.4.2), and we use it to supply the sets for said comprehensions.

¹Collection targets do require a degree of composition of *RoboChart* components, but this composition follows from the inner definitions of the parent components of the collections.

Value specifications

These rules concern $\text{ValueSpecification}^a$ s (‘arguments’).

Rule 2 — Value specifications (§ 1.4.2). This rule captures the semantics of a value specification as a channel transformer—an input (?) or output (!). This transformer will be used, for instance, to suffix the channel coming from a MessageTopic^a (see rule 17). The semantics requires an additional context to pass to the expression rule.

$$\llbracket - \rrbracket_-^{\text{val}} : \text{ValueSpecification}^a \rightarrow \text{Channel} \times \text{ExprContext}^a \rightarrow (\text{Channel} \rightarrow \text{Channel})$$

Case analysis on specification type. Wildcards become inputs (using the binding if it exists^a); expression specifications become outputs of the value resulting from the expression.

$$\begin{aligned} \llbracket x : \text{WildcardValueSpecification} \rrbracket_c^{\text{val}} &\stackrel{\text{def}}{=} ?(\text{if } x.\text{destination} = \perp \text{ then } _ \text{ else } x.\text{destination}) \\ \llbracket x : \text{ExpressionValueSpecification} \rrbracket_c^{\text{val}} &\stackrel{\text{def}}{=} !\llbracket x.\text{expression} \rrbracket_c^{\text{expr}} \end{aligned}$$

^aPer def. 10.5.15, implementations **may** rename the binding to prevent shadowing, so long as there is consistency.

Rule 3 — Value specifications in set position. As with rule 2, this rule captures the semantics of a value specification. However, instead of binding wildcards to CSP outputs, it assumes that a set-comprehension binding of a given name is in scope, and uses that. This means that the rule is useful alongside def. 10.4.2 for defining message sets.

The semantics requires an additional context to pass to the expression rule. Unlike rule 2, it does not take the destination channel; instead, it takes the parameter corresponding to the $\text{ValueSpecification}^a$, from which we ascertain the name of the set-comprehension binding.^a

$$\llbracket - \rrbracket_-^{\text{sval}} : \text{ValueSpecification}^a \rightarrow \text{Parameter}^{\text{rc}} \times \text{ExprContext}^a \rightarrow \text{Value}$$

Case analysis on specification type. Wildcards become the value bound to a comprehension variable with the same name as the input parameter. Expression specifications become the value resulting from the expression.

$$\begin{aligned} \llbracket x : \text{WildcardValueSpecification} \rrbracket_{(p,c)}^{\text{sval}} &\stackrel{\text{def}}{=} p.\text{name} \\ \llbracket x : \text{ExpressionValueSpecification} \rrbracket_{(p,c)}^{\text{sval}} &\stackrel{\text{def}}{=} \llbracket x.\text{expression} \rrbracket_c^{\text{expr}} \end{aligned}$$

^aNote that the typing of the result as Value implies that the set-comprehension binding has been resolved.

Definition 10.4.2 — Support sets of value specifications. Function expVal captures the set of values that satisfy a specification, given the associated parameter and an expression evaluation context.

$$\text{expVal} : \text{ValueSpecification}^a \rightarrow \text{Parameter}^{\text{rc}} \rightarrow \text{ExprContext}^a \rightarrow \mathbb{P}(\text{Value})$$

Case analysis on specification type. Wildcards permit any value of the parameter type; expressions permit only the value of the included expression.

$$\begin{aligned} \text{expVal}(x : \text{WildcardValueSpecification}, p, c) &\stackrel{\text{def}}{=} \llbracket p.\text{type} \rrbracket^{\text{type}} \\ \text{expVal}(x : \text{ExpressionValueSpecification}, p, c) &\stackrel{\text{def}}{=} \{ \llbracket x.\text{expression} \rrbracket_c^{\text{expr}} \} \end{aligned}$$

10.4.4 Assertions (§ 1.5)

These rules implement the semantics of assertions, which forms the top level of *RoboCert* semantics.

Rule 4 — Assertion (§ 1.5.2). $\llbracket x \rrbracket^{\text{asst}}$ is the semantics of Assertion x . There is no semantic distinction between assertions and properties.

$$\llbracket - \rrbracket^{\text{asst}} : \text{Assertion} \rightarrow \text{Prop} \qquad \llbracket x \rrbracket^{\text{asst}} \stackrel{\text{def}}{=} \llbracket x.\text{property} \rrbracket^{\text{prop}}$$

Rule 5 — Property^a (§ 1.5.2). $\llbracket x \rrbracket^{\text{prop}}$ is the semantics of Property^a x .

$$\llbracket - \rrbracket^{\text{prop}} : \text{Property}^a \rightarrow \text{Prop}$$

Case analysis on subtype, applying negation to CSP and sequence properties. Any CSPProperty instances lift directly into *tock*-CSP, with *negated* applied if necessary. Other instances delegate to further semantic rules.

$$\begin{aligned} \llbracket x : \text{CSPProperty} \rrbracket^{\text{prop}} &\stackrel{\text{def}}{=} \text{pneg}(x.\text{csp}) & \llbracket x : \text{CoreProperty} \rrbracket^{\text{prop}} &\stackrel{\text{def}}{=} \llbracket x \rrbracket^{\text{cprop}} \\ \llbracket x : \text{SequenceProperty} \rrbracket^{\text{prop}} &\stackrel{\text{def}}{=} \text{pneg}(\llbracket x \rrbracket^{\text{sprop}}) \end{aligned}$$

Rule 6 — CoreProperty (§ 1.5.3). $\llbracket x \rrbracket^{\text{cprop}}$ is the semantics of CoreProperty x .

$$\llbracket - \rrbracket^{\text{cprop}} : \text{CoreProperty} \rightarrow \text{Prop}$$

First, delegate to an auxiliary function over type, negation, and target semantics:

$$\llbracket x \rrbracket^{\text{cprop}} \stackrel{\text{def}}{=} \text{cp}(x.\text{type}, x.\text{negated}, \llbracket x.\text{group.target} \rrbracket^{\text{tgt}})$$

Then, case analysis on the negation and core property type. Aside from termination, negated properties are the *pneg* of their positive counterparts.

In the rules below, we assume that τ has priority over *tock* (the maximal progress principle). In CSP-M encodings, we make this explicit. We also assume an arbitrary channel r .

$$\begin{aligned} \text{cp}(\text{TERMINATION} \quad , \text{true} , t) &\stackrel{\text{def}}{=} \neg(\text{deadlock free}_{FD} \ t) \wedge \text{deadlock free}_{FD} (t ; \text{Run}(\{r\})) \\ \text{cp}(\text{TERMINATION} \quad , \text{false} , t) &\stackrel{\text{def}}{=} \text{Stop} \sqsubseteq_T (t ; r \rightarrow \text{Skip}) \\ \text{cp}(\text{TIMELOCK_FREE} \quad , \text{true} , t) &\stackrel{\text{def}}{=} \text{Run}(\{\text{tock}\}) \parallel \text{Chaos}(\text{Events} \setminus \{\text{tock}\}) \sqsubseteq_F t \\ \text{cp}(\text{DEADLOCK_FREE} , \text{true} , t) &\stackrel{\text{def}}{=} \left(\begin{array}{l} \text{divergence free} \\ \text{prioritise} (\\ \quad t[\text{tock} \leftarrow \text{tock}, \text{tock} \leftarrow \text{tock}'], \langle \text{Events} \setminus \{\text{tock}\} \rangle \\ \quad) \setminus \{\text{tock}\} \end{array} \right) \\ \text{cp}(\text{DETERMINISM} \quad , \text{true} , t) &\stackrel{\text{def}}{=} \text{deterministic}_{FD} \ t \\ \text{cp}(\text{TIMELOCK_FREE} \quad , \text{false} , t) &\stackrel{\text{def}}{=} \text{pneg}(\text{cp}(\text{TIMELOCK_FREE}, \text{true}, t)) \\ \text{cp}(\text{DEADLOCK_FREE} , \text{false} , t) &\stackrel{\text{def}}{=} \text{pneg}(\text{cp}(\text{DEADLOCK_FREE}, \text{true}, t)) \\ \text{cp}(\text{DETERMINISM} \quad , \text{false} , t) &\stackrel{\text{def}}{=} \text{pneg}(\text{cp}(\text{DETERMINISM}, \text{true}, t)) \end{aligned}$$

Rule 7 — SemanticModel. $\llbracket x \rrbracket^{\text{mdl}}$ is the semantics for SemanticModel x .

$$\llbracket - \rrbracket^{\text{mdl}} : \text{SemanticModel} \rightarrow \text{Model}$$

Case analysis on m . The semantics of a SemanticModel maps the traces *RoboCert* model to the CSP traces model, and the timed model to \checkmark -tock.

$$\llbracket \text{TRACES} \rrbracket^{\text{mdl}} \stackrel{\text{def}}{=} \top \qquad \llbracket \text{TIMED} \rrbracket^{\text{mdl}} \stackrel{\text{def}}{=} \top$$

Definition 10.4.3 — Property negation. Function pneg lifts the potential negation of property p into the level of CSP propositions, taking a proposition c to which the negation should be applied.

$$\text{pneg} : \text{Property}^a \rightarrow \text{Prop} \rightarrow \text{Prop} \qquad \text{pneg}(p, c) \stackrel{\text{def}}{=} \text{if } p.\text{negated} \text{ then } \neg c \text{ else } c$$

10.5 Sequence notation

These rules implement the *tock*-CSP semantics of the sequence notation. See § 9.2 for general definitions used in the rules below.

10.5.1 Assertions (§ 4.6)

These rules implement the semantics of sequence properties.

Rule 8 — SequenceProperty (§ 4.6.1). The semantics of sequence properties is a refinement between the Interaction and the Target^a of its specification group. When the property is a **HOLDS** property, the Target^a is in implementation position. In an **IS_OBSERVED** property, it is in specification position, and the sequence is made partial through composition with timestop .

$$\llbracket - \rrbracket^{\text{sprop}} : \text{SequenceProperty} \rightarrow \text{Prop}$$

Split on $x.\text{type}$:

$$\begin{aligned} \llbracket x \rrbracket^{\text{sprop}} &\stackrel{\text{def}}{=} \text{if } x.\text{type} = \text{IS_OBSERVED} \\ &\quad \text{then } \llbracket x.\text{sequence.group.target} \rrbracket^{\text{tgt}} \sqsubseteq_{\llbracket x.\text{model} \rrbracket^{\text{mdl}}} \llbracket x.\text{sequence} \rrbracket^{\text{int}} ; \text{Stop}_U \\ &\quad \text{else } \llbracket x.\text{sequence} \rrbracket^{\text{int}} \sqsubseteq_{\llbracket x.\text{model} \rrbracket^{\text{mdl}}} \llbracket x.\text{sequence.group.target} \rrbracket^{\text{tgt}} \end{aligned}$$

10.5.2 Sequences (§ 4.2)

These rules implement the semantics of sequences (Interactions) and their constituent processes.

Definition 10.5.1 — Interaction contexts. Pseudo-class *InteractionContext* represents the context carried across semantic functions for interactions. It consists of:

- the Interaction, used to resolve interaction-global elements;
- the set of all Actor^a s that are covered by the element whose semantics are being elaborated. For a lifeline over actor a , this is a singleton set $\{a\}$. If we are within a sequentialised situation such as a *UntilFragment*, the set will instead be \mathbf{U} .

$$\text{InteractionContext} \stackrel{\text{def}}{=} (\text{Interaction} \times \mathbb{P}(\text{Actor}^a))$$

Rule 9 — Interaction (§ 4.2.2). Rule $\llbracket x \rrbracket^{\text{int}}$ maps Interaction x to a CSP process.

$$\llbracket - \rrbracket^{\text{int}} : \text{Interaction} \rightarrow \underline{\text{Process}}$$

We define interactions in stages, lifting a basic lifeline process set into various contexts. At the top level, we hide the special channel *term*, which coordinates termination across the sequence and its auxiliary processes.

$$\llbracket x \rrbracket^{\text{int}} \stackrel{\text{def}}{=} (\underline{\text{intMemory}(x)}) \setminus \{term\}$$

The next stages take the form of auxiliary context-lifting functions:

$$\begin{aligned} \text{intMemory} &: \text{Interaction} \rightarrow \underline{\text{Process}} & \text{intUntil} &: \text{Interaction} \rightarrow \underline{\text{Process}} \\ \text{intLifelines} &: \text{Interaction} \rightarrow \underline{\text{Process}} \end{aligned}$$

The first level of the rule applies the memory.^a We define $\text{msync}(x)$ and $\text{mem}(x)$ in § 10.5.7.

$$\underline{\text{intMemory}(x)} \stackrel{\text{def}}{=} (\underline{\text{intLifelines}(x)} \llbracket \text{msync}(x) \rrbracket \underline{\text{until}(x)}) \setminus \text{msync}(x)$$

The next level of the definition applies the until process and parallel composition synchronisation.^b We define the set *sync* in def. 10.5.2.

$$\underline{\text{intUntil}(x)} \stackrel{\text{def}}{=} (\underline{\text{intLifelines}(x)} \llbracket \text{sync} \rrbracket \underline{\text{until}(x)}) \setminus \text{sync}$$

Next, we define the core lifeline process.

$$\underline{\text{intLifelines}(x)} \stackrel{\text{def}}{=} \parallel a : \{a : x.\text{actors} \mid a \notin \text{World}\} \bullet \alpha_{x.\text{name}}(a) \circ \text{lifeline}_{x.\text{name}}(a)$$

where we define the following two process functions stepwise, in CSP, for each a :

$$\alpha_{x.\text{name}}(a) = \underline{\alpha(a, b)} \quad \text{lifeline}_{x.\text{name}}(a) = \underline{\% f : a.\text{fragments} \bullet \llbracket b_1 \rrbracket^{\text{frag}}_{(x, \{a\})}}$$

^aThe implementation **may** omit this lifting if there are no variables.

^bThe implementation **may** omit parts of this lifting if there are no UntilFragments or ParFragments.

Definition 10.5.2 — Sync channel set. Function *sync* generates a channel set for synchronising auxiliary processes in sequences.

$$\text{sync} : \text{Interaction} \rightarrow \mathbb{P}(\text{Channel})$$

We define $\text{sync}(s)$ as containing two types of synchronisation channel: *until* and *par*. The former, representing synchronisation on UntilFragments, communicates the index of the UntilFragment in question as well as the direction of synchronisation. The latter, representing synchronisation on ParFragments, communicates the index of the ParFragment; it has only one direction.

$$\begin{aligned} \text{sync}(s) \stackrel{\text{def}}{=} & \bigcup \{i : 0.. \# \text{flist}(s, \text{UntilFragment}) - 1, d : \{in, out\}, \bullet \text{until}.i.d\} \cup \\ & \bigcup \{i : 0.. \# \text{flist}(s, \text{ParFragment}) - 1 \bullet \text{par}.i\} \end{aligned}$$

Here, i numbers each *par* and *until* fragment by position in the diagram; d states whether we are entering or leaving the fragment.

10.5.3 Fragments (§ 4.3)

These rules implement the semantics of $\text{InteractionFragment}^a$ s.

Rule 10 — $\text{InteractionFragment}^a$ (§ 4.3). The top-level interaction fragment rule takes both a fragment and an interaction context (def. 10.5.1).

$$\llbracket - \rrbracket_{-}^{\text{frag}} : \text{InteractionFragment}^a \rightarrow \text{InteractionContext} \rightarrow \text{Process}$$

The fragment rule must first load from memory any variables referenced directly by the fragment. We generate the loading process by extracting expressions with fexprs (def. 9.2.4), extracting from those expressions the variables inside (def. 9.1.1), then passing them to load (def. 10.5.15).

$$\llbracket x \rrbracket_c^{\text{frag}} \stackrel{\text{def}}{=} \text{load}(\cap / \langle e : \text{fexprs}(x) \bullet \text{eVars}(e) \rangle) ; \text{fragBody}(x, c)$$

We now define fragBody , which handles the part of the fragment semantics that follows any control-flow variable loading.

$$\text{fragBody} : \text{InteractionFragment}^a \rightarrow \text{InteractionContext} \rightarrow \text{Process}$$

Case analysis on high-level types of fragment (we define oactors in def. 9.2.7):

$$\begin{aligned} \text{fragBody}(x : \text{BlockFragment}^a)(c) &\stackrel{\text{def}}{=} \llbracket x \rrbracket_c^{\text{blfrag}} \\ \text{fragBody}(x : \text{BranchFragment}^a)(c) &\stackrel{\text{def}}{=} \llbracket x \rrbracket_c^{\text{brfrag}} \\ \text{fragBody}(x : \text{OccurrenceFragment})(s, a) &\stackrel{\text{def}}{=} \text{if } a \cap A(o) = \emptyset \\ &\quad \text{then } \text{Skip} \text{ else } \llbracket x.\text{occurrence} \rrbracket_s^{\text{occ}} \end{aligned}$$

Rule 11 — BlockFragment^a (§ 4.3.2). The basic semantics of a BlockFragment^a is:^a

$$\llbracket - \rrbracket_{-}^{\text{blfrag}} : \text{BlockFragment}^a \rightarrow \text{InteractionContext} \rightarrow \text{Process}$$

Case analysis on concrete type of fragment:

$$\begin{aligned} \llbracket x : \text{DeadlineFragment} \rrbracket_{(s,a)}^{\text{blfrag}} &\stackrel{\text{def}}{=} \text{if } x.\text{actor} \in a \text{ then } \left(\llbracket x.\text{body} \rrbracket_{(s,a)}^{\text{iop}} \blacktriangleright \llbracket x.\text{units} \rrbracket_s^{\text{expr}} \right) \\ &\quad \text{else } \llbracket x.\text{body} \rrbracket_{(s,a)}^{\text{iop}} \\ \llbracket x : \text{LoopFragment} \rrbracket_{(s,a)}^{\text{blfrag}} &\stackrel{\text{def}}{=} \text{loop} \left(x.\text{bound}, s, \llbracket x.\text{body} \rrbracket_{(s,a)}^{\text{iop}} \right) \\ \llbracket x : \text{OptFragment} \rrbracket_c^{\text{blfrag}} &\stackrel{\text{def}}{=} \llbracket x.\text{body} \rrbracket_c^{\text{iop}} \sqcap \text{Skip} \\ \llbracket x : \text{UntilFragment} \rrbracket_c^{\text{blfrag}} &\stackrel{\text{def}}{=} \text{until}.\text{findindex}(u, \text{UntilFragment}).\text{in} \rightarrow \\ &\quad \text{until}.\text{findindex}(u, \text{UntilFragment}).\text{out} \rightarrow \text{Skip} \end{aligned}$$

^aImplementations **may** replace the semantics of UntilFragments with the contents of $\text{until}(x)$ when synchronisation with other lifelines is not needed.

Definition 10.5.3 — LoopFragment (§ 4.3.4). Because the semantics of a LoopFragment depends on its DiscreteBound , we use an auxiliary function loop to capture its semantics. This function

lifts a pre-expanded semantic transformation of the body based on the bound.

$$\text{loop} : \text{DiscreteBound}^? \rightarrow \text{Interaction} \rightarrow \underline{\text{Process}} \rightarrow \underline{\text{Process}}$$

Case analysis on the definedness of the bound:

$$\underline{\text{loop}}(\perp, s, P) \stackrel{\text{def}}{=} \mu x \bullet \underline{P} \circ x \quad \underline{\text{loop}}(b, s, P) \stackrel{\text{def}}{=} \underline{\text{bLoop}}(\llbracket b.\text{lower} \rrbracket_s^{\text{expr}}, \llbracket b.\text{upper} \rrbracket_s^{\text{expr}}, P)$$

Another function, `bLoop`, handles the further expansion of loops that have a defined `DiscreteBound`. It accepts the bounds as potentially-undefined natural numbers.

$$\text{bLoop} : \mathbb{N}^? \rightarrow \mathbb{N}^? \rightarrow \underline{\text{Process}} \rightarrow \underline{\text{Process}}$$

Case analysis on the definedness of the upper and lower bounds, where:

upper bound u only an *exact* bound (ie, we interpret the missing lower bound as being u); here, there is a straightforward parametric recursive process expansion that loops u times;

lower bound l only a combination of an exact-bounded loop for l iterations followed by a loop recursing a nondeterministic amount of times;

both bounds a combination of an exact-bounded loop for l iterations followed by a parametric recursive process that, for up to $u - l$ iterations, either terminates or iterates;

no bounds ill-formed.

$$\begin{aligned} \underline{\text{bLoop}}(\perp, \perp, P) &\stackrel{\text{def}}{=} \text{undefined} \\ \underline{\text{bLoop}}(l, \perp, P) &\stackrel{\text{def}}{=} \underline{\text{bLoop}}(\perp, l, P) \circ (\mu x \bullet \text{Skip} \sqcap (\underline{P} \circ x)) \\ \underline{\text{bLoop}}(\perp, e, P) &\stackrel{\text{def}}{=} \text{let } L(x) = (\text{if } x = 0 \text{ then } \text{Skip} \text{ else } (\underline{P} \circ L(x - 1))) \text{ within } L(e) \\ \underline{\text{bLoop}}(l, u, P) &\stackrel{\text{def}}{=} \underline{\text{bLoop}}(\perp, l, P) \circ \\ &\quad \left(\text{let } L(x) = \text{Skip} \sqcap (\text{if } x = 0 \text{ then } \text{Skip} \text{ else } (\underline{P} \circ L(x - 1))) \right) \\ &\quad \text{within } L(u - l) \end{aligned}$$

Rule 12 — BranchFragment^a (§ 4.3.8). The basic semantics of a BranchFragment^a is:

$$\llbracket - \rrbracket_-^{\text{bfrag}} : \text{BranchFragment}^a \rightarrow \text{InteractionContext} \rightarrow \underline{\text{Process}}$$

Case analysis on concrete type of fragment:

$$\begin{aligned} \llbracket x : \text{ParFragment} \rrbracket_c^{\text{bfrag}} &\stackrel{\text{def}}{=} \llbracket \{p : x.\text{branches} \bullet \llbracket p \rrbracket_c^{\text{lop}}\} \rrbracket \circ \text{par.pindex}(x).\text{out} \rightarrow \text{Skip} \\ \llbracket x : \text{AltFragment} \rrbracket_c^{\text{bfrag}} &\stackrel{\text{def}}{=} \sqcap \{p : x.\text{branches} \bullet \llbracket p \rrbracket_c^{\text{lop}}\} \\ \llbracket x : \text{XAltFragment} \rrbracket_c^{\text{bfrag}} &\stackrel{\text{def}}{=} \sqcap \{p : x.\text{branches} \bullet \llbracket p \rrbracket_c^{\text{lop}}\} \\ \underline{\text{pindex}}(x) &\stackrel{\text{def}}{=} \underline{\text{findex}}(x, \text{ParFragment}) \end{aligned}$$

The semantics of `OptFragment` is therefore equivalent to the semantics of a `AltFragment` with two *branches*: one containing the *body*, another containing a `EmptyGuard` and zero fragments.

10.5.4 Occurrences (§ 4.4)

These rules capture the semantics of Occurrence^as, as well as the temperature modality of MessageOccurrences. Occurrence semantics requires the parent Interaction, for expression evaluation.

Rule 13 — Occurrence^a (§ 4.4). This rule captures the top-level Occurrence^a semantics.

$$\llbracket - \rrbracket_-^{\text{occ}} : \text{Occurrence}^a \rightarrow \text{Interaction} \rightarrow \text{Process}$$

Case analysis on the type of Occurrence^a:

$$\begin{aligned} \llbracket x : \text{DeadlockOccurrence} \rrbracket_s^{\text{occ}} &\stackrel{\text{def}}{=} \text{Stop} \\ \llbracket x : \text{MessageOccurrence} \rrbracket_s^{\text{occ}} &\stackrel{\text{def}}{=} \text{heat}(\llbracket x.\text{message} \rrbracket_s^{\text{msg}}, x.\text{temperature}) \\ \llbracket x : \text{WaitOccurrence} \rrbracket_s^{\text{occ}} &\stackrel{\text{def}}{=} \text{wait} \llbracket x.\text{units} \rrbracket_s^{\text{expr}} \end{aligned}$$

Definition 10.5.4 — Temperature. Function $\text{heat}(p, t)$ applies temperature t to CSP process p .

$$\text{heat} : \text{Process} \rightarrow \text{Temperature} \rightarrow \text{Process}$$

Case analysis on temperature:

$$\text{heat}(p, \text{HOT}) \stackrel{\text{def}}{=} p \qquad \text{heat}(p, \text{COLD}) \stackrel{\text{def}}{=} \mu q \bullet (p \sqcap \text{tock} \rightarrow q)$$

10.5.5 Messages (§ 4.5)

These rules implement the semantics of Messages and related classes.

Message sets

The *RoboCert* message set language is a straightforward shallow embedding of a simple set algebra enriched with a notion of named references to sets, and the semantics reflects this.

Rule 14 — MessageSet^a (§ 4.5.1). Rule $\llbracket m \rrbracket_c^{\text{mset}}$ gives semantics to MessageSet^a m in context c .

$$\llbracket - \rrbracket_-^{\text{mset}} : \text{MessageSet}^a \rightarrow \text{InteractionContext} \rightarrow \mathbb{P}(\text{Event})$$

Case analysis of the subtypes of set. A UniverseMessageSet represents the universe of all events.^a A ExtensionalMessageSet is the union of the event sets of each message, which we get from expMsg (def. 10.5.5). A BinaryMessageSet represents binary set operations (which expand to those operations); a RefMessageSet captures references (which expand to the referred-to set).

$$\begin{aligned} \llbracket x : \text{UniverseMessageSet} \rrbracket_s^{\text{mset}} &\stackrel{\text{def}}{=} \mathbf{U} \\ \llbracket x : \text{ExtensionalMessageSet} \rrbracket_s^{\text{mset}} &\stackrel{\text{def}}{=} \bigcup \{m : x.\text{messages} \bullet \text{expMsg}(x, s)\} \\ \llbracket x : \text{BinaryMessageSet} \rrbracket_s^{\text{mset}} &\stackrel{\text{def}}{=} \text{binset}(x.\text{operator}, \llbracket x.\text{lhs} \rrbracket_s^{\text{mset}}, \llbracket x.\text{rhs} \rrbracket_s^{\text{mset}}) \\ \llbracket x : \text{RefMessageSet} \rrbracket_s^{\text{mset}} &\stackrel{\text{def}}{=} \llbracket x.\text{set} \rrbracket_s^{\text{nmset}} \end{aligned}$$

We define binset as follows:

$$\text{binset} : \text{BinarySetOperator} \rightarrow \text{MessageSet}^a \rightarrow \text{MessageSet}^a \rightarrow \mathbb{P}(\text{Event})$$

Case analysis on the operator:

$$\begin{aligned} \text{binset}(\text{UNION}, l, r) &\stackrel{\text{def}}{=} l \cup r & \text{binset}(\text{INTERSECTION}, l, r) &\stackrel{\text{def}}{=} l \cap r \\ \text{binset}(\text{DIFFERENCE}, l, r) &\stackrel{\text{def}}{=} l \setminus r \end{aligned}$$

^aFuture versions of *RoboCert* may refine this to the set of all events that can originate from participants in the sequence, or provide a mechanism for specifying the set of all events from a specific participant.

Rule 15 — NamedMessageSet (§ 4.5.2). The semantics of a NamedMessageSet is that of its set.

$$\llbracket - \rrbracket_{-}^{\text{nmset}} : \text{MessageSet}^a \rightarrow \text{Interaction} \rightarrow \mathbb{P}(\text{Event}) \quad \llbracket x \rrbracket_c^{\text{nmset}} \stackrel{\text{def}}{=} \llbracket x.\text{set} \rrbracket_c^{\text{mset}}$$

Messages

The semantics of a message depends on:

- whether the message is inside a MessageOccurrence (in which case, its semantics is that of a single event with possible binding and matching of argument patterns) or inside a MessageSet^a (its semantics is that of a *set* of events satisfying certain argument patterns);
- whether the *topic* of the feature is an operation or an event;
- the pair of Actor^as connected by the message, which determine the *RoboChart* model component associated with the translated CSP events and their direction, if applicable;
- the ValueSpecification^as forming message arguments, which become suffixes to the CSP event (and, in the case of message sets, quantifications in the set comprehension).

Rule 16 — Message (§ 4.5.3). The semantics of a Message resolves to a CSP process in which the message topic and arguments form a prefix (which may bind wildcard arguments to inputs), and a subsequent storage phase discharges those inputs into the sequence memory. As such, there is a distinction between this rule and `expMsg`, which expands *all* possible invocations of a message into an *event set*.

$$\llbracket - \rrbracket_{-}^{\text{msg}} : \text{Message} \rightarrow \text{Interaction} \rightarrow \text{Process}$$

To form a message, we expand its *topic* into a channel stub, then recursively attach each argument to the stub as an input or output. We assert that the result is a well-typed process, by appeal to the well-formed conditions in § 5.4. Any bindings of arguments to CSP outputs must be propagated to memory; we collect each `vBind` (def. 9.1.2) and store them to memory (def. 10.5.16).

$$\llbracket x \rrbracket_s^{\text{msg}} \stackrel{\text{def}}{=} \text{args}(\llbracket x.\text{topic} \rrbracket_{(x.\text{from}, x.\text{to})}^{\text{topic}}, x.\text{arguments}, s) \rightarrow \text{store}(\text{vBind}(x.\text{arguments}))$$

Definition 10.5.5 — Expansion of messages. Function `expMsg` expands a Message into the set of all possible CSP events that can arise from it.

$$\text{expMsg} : \text{Message} \rightarrow \text{Interaction}^? \rightarrow \mathbb{P}(\text{Event})$$

As with $\llbracket x \rrbracket_s^{\text{msg}}$, `expMsg` elaborates the topic first and then recursively applies arguments. There are two differences. First, the argument expansion is in two stages: one stage `cargs` produces a set comprehension over all wildcard arguments; a second `sargs` behaves similarly to `cargs` in that it appends arguments to the topic stub (but expands wildcards to comprehension bindings rather than inputs). we do not store bindings (as they are not allowed in message set position).

If there are no arguments, we do not emit a set comprehension.^a

$$\text{expMsg}(x, s) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } x.\text{arguments} = \langle \rangle \text{ then } \llbracket x.\text{topic} \rrbracket_{(x.\text{from}, x.\text{to})}^{\text{topic}} \\ \text{else } \left\{ \begin{array}{l} \text{cargs}(\text{params}(x.\text{topic}), x.\text{arguments}) \bullet \\ \text{sargs}(\llbracket x.\text{topic} \rrbracket_{(x.\text{from}, x.\text{to})}^{\text{topic}}, \text{params}(x.\text{topic}), x.\text{arguments}, s) \end{array} \right\} \end{array}$$

^aImplementations **may** also inline comprehension bindings for arguments where there is only one possible value.

Message topics

These rules concern the part of the Message semantics that varies with different MessageTopic^as.

Rule 17 — MessageTopic^a (§ 4.5.4). This rule expands a MessageTopic^a to a channel (which, if the message has arguments, must be completed with arguments to become an event). The semantics of a topic also depends on the pair of Actor^as being connected by the topic.

$$\llbracket - \rrbracket_{-}^{\text{topic}} : \text{MessageTopic}^a \rightarrow (\text{Actor}^a \times \text{Actor}^a) \rightarrow \text{Channel}$$

Case analysis on whether the topic is an operation (def. 10.5.6) or an event (def. 10.5.7).

$$\begin{array}{l} \llbracket x : \text{OperationTopic} \rrbracket_{(f, t)}^{\text{topic}} \stackrel{\text{def}}{=} \text{oTopic}(x.\text{operation}, t) \\ \llbracket x : \text{EventTopic} \rrbracket_a^{\text{topic}} \stackrel{\text{def}}{=} \text{eTopic}(x.\text{event}, a) \end{array}$$

Definition 10.5.6 — OperationTopic. Function oTopic handles the semantics of an OperationTopic.

$$\text{oTopic} : \text{OperationTopic} \rightarrow \text{Actor}^a \rightarrow \text{Channel}$$

We assume through well-formedness that the to-Actor^a is a World, and so the semantics of the topic is the elaboration of the operation *name* within the namespace of the Actor^a's associated Process. We find the latter using actNs (def. 9.2.1).

$$\text{oTopic}(x, t) \stackrel{\text{def}}{=} \text{actNs}(t) :: \text{x.operation.nameCall}$$

Definition 10.5.7 — EventTopic. Function eTopic handles the semantics of an EventTopic.

$$\text{eTopic} : \text{EventTopic} \rightarrow (\text{Actor}^a \times \text{Actor}^a) \rightarrow \text{Channel}$$

We define the event topic semantics as the combination of several components of information about an event as resolved by an auxiliary function eInfo. These components are, in order: the direction (input/output) of the event; the 'base' Actor^a from which this side of the event is visible; and the event itself. The high-level output of this rule is similar to that for oTopic, but accounting for the above.

$$\text{eTopic}(t, a) \stackrel{\text{def}}{=} \text{let } (d, b, e) = \text{eInfo}(t, a) \text{ within } \text{actNs}(b) :: \text{eventId}(e).d$$

Event resolution

Much of the semantics of messages concerns resolving EventTopics into triples of event direction, base Actor^a, and underlying Event. We implement this as a series of functions below.

Definition 10.5.8 — Event information. Function eInfo resolves the direction, base Actor^a, and

Event^{rc} of an EventTopic . This resolution reflects the *RoboChart* semantics for events.

$$\text{elInfo} : \text{EventTopic} \rightarrow (\text{Actor}^{\text{a}} \times \text{Actor}^{\text{a}}) \rightarrow (\{in, out\} \times \text{Actor}^{\text{a}} \times \underline{\text{Event}})$$

At the top level, elInfo is a case analysis on the Actor^{a} s. If one actor is a *World* (the message is either inbound or outbound), then the base actor is its *opposite*; the Event^{rc} is that defined on the base actor's end of the message; and the direction is that of the message with respect to the base actor (for instance, if the from-actor is a *World*, the direction is *in* as it is heading *into* the to-actor). This ensures that the semantics refers only to elements visible in the current Target^{a} .

If neither of the actors are a *World*, we have an inter-component communication. *RoboCert* version 0.1 does not yet support asynchronous communications here. For synchronous communications, we delegate to eConn (def. 9.2.9) to find the underlying $\text{Connection}^{\text{rc}}$, and eSync to expand it into a direction, actor, and event.

Well-formedness conditions prevent the ambiguous situation of both actors being *World*.

$$\begin{aligned} \text{elInfo}(e, (f : \text{World}, t)) &\stackrel{\text{def}}{=} (in, t, \text{if } e.\text{eto} = \perp \text{ then } e.\text{from} \text{ else } e.\text{to}) \\ \text{elInfo}(e, (f, t : \text{World})) &\stackrel{\text{def}}{=} (out, f, e.\text{efrom}) \\ \text{elInfo}(e, (f, t)) &\stackrel{\text{def}}{=} \text{eSync}((f, t), f.\text{node}, \text{eConn}(e, (f, t))) \quad (\text{default}) \end{aligned}$$

Definition 10.5.9 — Event information (synchronous inter-component). Function eSync resolves the direction, base Actor^{a} , and Event^{rc} of a synchronous, inter-component EventTopic , given its actor pair and already-resolved $\text{Connection}^{\text{rc}}$.

$$\text{eSync} : \text{EventTopic} \rightarrow (\text{Actor}^{\text{a}} \times \text{Actor}^{\text{a}}) \rightarrow \text{Connection}^{\text{rc}} \rightarrow (\{in, out\} \times \text{Actor}^{\text{a}} \times \underline{\text{Event}})$$

The definition of elInfo closely matches rule 15 of the *RoboChart* semantics. In that rule, synchronous input and output pairs merge to form a single CSP event from the perspective of the 'from' node of the $\text{Connection}^{\text{rc}}$.

Usually, then, we take the direction as being *out* (the message comes *out* of the 'from' node), the base actor as being that corresponding to the 'from' node, and the event as the 'from' event. We must, however, handle the possibility of an event topic that captures a bidirectional message, and whose Actor^{a} s are in the opposite position from the defined endpoints of the underlying $\text{Connection}^{\text{rc}}$. In this case, we reverse the direction and Actor^{a} choice.

$$\text{eSync}(e, (f, t), c) \stackrel{\text{def}}{=} \text{if } f = c.\text{from} \text{ then } (out, f, c.\text{efrom}) \text{ else } (in, t, c.\text{efrom})$$

Arguments

The following semantic functions handle the appendage of arguments ($\text{ValueSpecification}^{\text{a}}$ s) onto channels and sets formed by considering $\text{MessageTopic}^{\text{a}}$ s. (The definition of the semantics for $\text{ValueSpecification}^{\text{a}}$ can be found at rule 2 and def. 10.4.2.)

Definition 10.5.10 — Arguments in event position. Function args recursively adds the semantic elaboration of each argument, through $\llbracket - \rrbracket^{\text{val}}$ (rule 2), onto the channel being built.

$$\text{args} : \underline{\text{Channel}} \rightarrow \text{seq } \text{ValueSpecification}^{\text{a}} \rightarrow \text{Interaction} \rightarrow \underline{\text{Event}}$$

By recursion on the $\text{ValueSpecification}^{\text{a}}$ list.

$$\text{args}(c, \langle \rangle, s) \stackrel{\text{def}}{=} c \quad \text{args}(c, \langle a \rangle \frown x, s) \stackrel{\text{def}}{=} \text{args}(\llbracket a \rrbracket_s^{\text{val}}(c), x, s)$$

Definition 10.5.11 — Arguments in set position. Function sargs recursively appends the semantic elaboration of each argument, through $\llbracket - \rrbracket_{-}^{\text{svd}}$ (rule 3), onto the channel being built.

$$\text{sargs} : \text{Channel} \rightarrow \text{seq Parameter}^{\text{rc}} \rightarrow \text{seq ValueSpecification}^{\text{a}} \rightarrow \text{Interaction}^? \rightarrow \text{Event}$$

By recursion on the $\text{ValueSpecification}^{\text{a}}$ list. We assume the well-formedness condition that lists of arguments and corresponding parameters are of equal length.

$$\text{sargs}(c, \langle \rangle, s) \stackrel{\text{def}}{=} c \quad \text{sargs}(c, \langle p \rangle \frown x, \langle a \rangle \frown y, s) \stackrel{\text{def}}{=} \text{sargs}(c, \llbracket a \rrbracket_{(p,s)}^{\text{svd}}, x, y, s)$$

Definition 10.5.12 — Argument set comprehension. Function cargs recursively builds a set comprehension binding for a series of arguments, by applying expVal (def. 10.4.2) to each.

$$\text{cargs} : \text{seq Parameter}^{\text{rc}} \rightarrow \text{seq ValueSpecification}^{\text{a}} \rightarrow \text{Binding}$$

By recursion on the lists. We assume the well-formedness condition that lists of arguments and corresponding parameters are of equal length. We also assume that the lists are non-empty.

$$\begin{aligned} \text{cargs}(\langle p \rangle \langle a \rangle, s) &\stackrel{\text{def}}{=} \text{expVal}(a, p) \\ \text{cargs}(\langle p \rangle \frown x, \langle a \rangle \frown y, s) &\stackrel{\text{def}}{=} \text{expVal}(a, p), \text{cargs}(x, y, s) \end{aligned}$$

10.5.6 The until process

These rules implement the semantics of *UntilFragments* in terms of an auxiliary process that linearises the fragment bodies.

Definition 10.5.13 — Until process. The until process handles *UntilFragment* bodies for a *Interaction*.

$$\text{until} : \text{Interaction} \rightarrow \text{Process}$$

The process is a recursive offer of synchronisation on every *UntilFragment* in the *Interaction* until the *Interaction* terminates.

$$\text{until}(x) \stackrel{\text{def}}{=} \mu p \bullet (\text{term} \rightarrow \text{Skip}) \sqcap \left(\begin{array}{c} \square(i, u) : \text{flist}(x, \text{UntilFragment}) \bullet \\ \text{sync}.i.\text{in} \rightarrow \text{ufrag}(u, x) \rightarrow \text{sync}.i.\text{out} \rightarrow p \end{array} \right)$$

We define the semantics of an individual fragment through an auxiliary function. The semantics is to allow deadlock, time passage, or any event in the message set of the fragment until the trigger process has taken effect.

$$\text{ufrag} : \text{UntilFragment} \rightarrow \text{Interaction} \rightarrow \text{Process}$$

$$\text{ufrag}(u, s) \stackrel{\text{def}}{=} T\text{Chaos}(\llbracket u.\text{intraMessages} \rrbracket_s^{\text{mset}}) \triangle \llbracket u.\text{body} \rrbracket_{(s,U)}^{\text{iop}}$$

10.5.7 Memory

These rules implement the semantics of sequence memory. This includes the memory process as well as loads and stores.

Definition 10.5.14 — Memory process. The memory process persists *RoboCert*-level variables.

$$\text{mem} : \mathbb{P}(\text{Variable}^{\text{rc}}) \rightarrow \text{Process}$$

The memory process closes over a function from variables to values, starting with every variable mapped to its initial value.^a It offers the choice of setting variables (which overrides their definition in the function) and getting variables (which outputs their current value in the function). As with most auxiliary processes, it offers to terminate on *term*.

$$\underline{\text{mem}}(v) \stackrel{\text{def}}{=} \left(\begin{array}{l} (term \rightarrow Skip) \square \\ \text{let } M(r) = \left(\begin{array}{l} \square n : \{x : v \bullet x.name\} \bullet \\ \left((var.n.set?x \rightarrow M(r \oplus \{n \mapsto x\})) \square \right) \\ (var.n.get!r(x) \rightarrow M(r)) \end{array} \right) \\ \text{within } M(\{n : \{x : v \bullet x.name\} \bullet n \mapsto \text{initial}(x)\}) \end{array} \right) \end{array} \right)$$

^aImplementations **may** capture this relation in any way that provides a compatible semantics. The reference implementation uses sequences indexed on the position of the variable in a list, rather than functions over names.

Definition 10.5.15 — Loading. The load function implements variable loads.

$$\text{load} : \text{seq Variable}^{\text{rc}} \rightarrow \underline{\text{Process}}$$

A load process is the sequential composition of inputs on the *var* channel for every variable in the given variable list.^a

$$\underline{\text{load}}(v) \stackrel{\text{def}}{=} ; n : \{x : v \bullet x.name\} \bullet var.n.get?n \rightarrow Skip$$

^aImplementations **may** rename the destinations of inputs, for instance to avoid clashing, so long as the results are consistent with $\llbracket - \rrbracket_{-}^{\text{expr}}$ and the store function.

Definition 10.5.16 — Storing. The store function implements variable stores.

$$\text{store} : \text{seq Variable}^{\text{rc}} \rightarrow \underline{\text{Process}}$$

A store process is the inverse of a load process: the sequential composition of outputs on the *var* channel for every variable in the given variable list.

$$\underline{\text{store}}(v) \stackrel{\text{def}}{=} ; n : \{x : v \bullet x.name\} \bullet var.n.set!n \rightarrow Skip$$

A. Language changelog

This chapter lists changes in the RoboCert language (not the tool), in reverse chronological version order.

A.1 This draft

Semantics changes

- Sequence memory processes now terminate when the rest of the sequence terminates, fixing a semantic error.

A.2 Version 0.1 (2022-05-20)

Initial released version.

Credits

ℒ^AT_EX style based on the The Legrand Orange Book Template by Mathias Legrand and Vel from [LaTeXTemplates.com](https://www.latextemplates.com). Licensed under CC BY-NC-SA 3.0.

Bibliography

- [1] Marco Autili, Paola Inverardi, and Patrizio Pelliccione. “Graphical scenarios for specifying temporal properties: An automated approach”. In: *Autom. Softw. Eng.* 14 (Sept. 2007), pages 293–340. DOI: [10.1007/s10515-007-0012-6](https://doi.org/10.1007/s10515-007-0012-6) (cited on pages 31, 32).
- [2] Matthias Brill et al. “Live Sequence Charts”. In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*. Edited by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 374–399. ISBN: 978-3-540-27863-4. DOI: [10.1007/978-3-540-27863-4_21](https://doi.org/10.1007/978-3-540-27863-4_21). URL: https://doi.org/10.1007/978-3-540-27863-4_21 (cited on page 26).
- [3] Werner Damm and David Harel. “LSCs: Breathing Life into Message Sequence Charts”. In: *Formal Methods Syst. Des.* 19.1 (2001), pages 45–80. DOI: [10.1023/A:1011227529550](https://doi.org/10.1023/A:1011227529550). URL: <https://doi.org/10.1023/A:1011227529550> (cited on page 33).
- [4] *OMG Unified Modeling Language*. Dec. 2017. URL: <https://www.omg.org/spec/UML/2.5.1/PDF> (cited on pages 10, 25).
- [5] Øystein Haugen and Ketil Stølen. “STAIRS – Steps To Analyze Interactions with Refinement Semantics”. In: *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Edited by Perdita Stevens, Jon Whittle, and Grady Booch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pages 388–402. ISBN: 978-3-540-45221-8 (cited on page 26).
- [6] J. Baxter, P. Ribeiro, and A. Cavalcanti. “Sound reasoning in tock-CSP”. In: *Acta Informatica* (Apr. 2021). DOI: [10.1007/s00236-020-00394-3](https://doi.org/10.1007/s00236-020-00394-3). URL: <https://eprints.whiterose.ac.uk/174356/> (cited on page 61).
- [7] Waldeck Lindoso et al. “Visual Specification of Properties for Robotic Designs”. In: *Formal Methods: Foundations and Applications*. Edited by Sérgio Campos and Marius Minea. Cham: Springer International Publishing, 2021, pages 34–52. ISBN: 978-3-030-92137-8 (cited on page 32).

- [8] Zoltán Micskei and Hélène Waeselynck. “The Many Meanings of UML 2 Sequence Diagrams: A Survey”. In: *Softw. Syst. Model.* 10.4 (Oct. 2011), pages 489–514. ISSN: 1619-1366. DOI: [10.1007/s10270-010-0157-9](https://doi.org/10.1007/s10270-010-0157-9). URL: <https://doi.org/10.1007/s10270-010-0157-9> (cited on page 25).