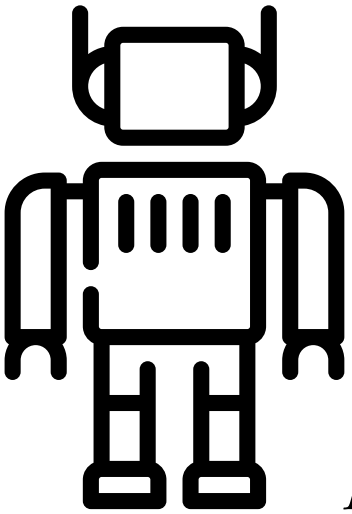


RoboChart Reference Manual

Alvaro Miyazawa Ana Cavalcanti Pedro Ribeiro Wei Li
Jim Woodcock Jon Timmis

July 2016

Autonomous and mobile robots are becoming ubiquitous. From domestic robotic vacuum cleaners to driverless cars, such robots interact with their environment and humans, leading to potential safety hazards. We propose a three-pronged solution to the problem of safety of mobile and autonomous robots: (1) domain-specific modelling with a formal underpinning; (2) automatic generation of sound simulations; and (3) verification based on model checking and theorem proving. Here, we report on a UML-like notation called RoboChart, designed specifically for modelling autonomous and mobile robots, and including timed and probabilistic primitives. We discuss a denotational semantics for a core subset of RoboChart, an approach for the development of sound simulations, and an implementation of RoboChart and its formal semantics as an Eclipse plugin supported by the CSP model checker FDR.

This report is a reference manual for the *RoboChart* notation. It describes the syntax of *RoboChart* and its extensions, as well as the well-formedness conditions and semantics of the language constructs. Additionally, usage of the language is discussed via a application programming interface (API), simulation support and a number of examples.

Contents

List of Figures

Chapter 1

Introduction

The current practice of programming mobile and autonomous robots does not reflect the modern outlook of their applications. Such practice is often based on standard state machines, without formal semantics, to describe the robot controller only, with time and probabilistic properties discussed in natural language. In the design stage, the state machine guides the development of a simulation, but no rigorous connection between them is established.

In this report, we present a state-machine based notation, called RoboChart, for the specification and design of robotic systems. Besides state machines, RoboChart includes elements to organise specifications, fostering reuse and taming complexity. These extra constructs embed the notions of robotic platforms and their controllers; communication between controllers can be synchronous or asynchronous. The state-machine notation is fully specified, including an action language and constructs to specify timing and probabilistic properties. Operations used in a state machine can be taken from a domain-specific API or defined by other state machines; communication between state machines inside a controller is synchronous. Operations can be given pre and postconditions.

The time primitives of RoboChart allow time budgets and deadlines to be specified for operations and events directly as part of a state machine. Constraints can be specified in association with the relative-time elapsed since the occurrence of events or the entering of states. Our time primitives are inspired by constructs of timed automata [2] and Timed CSP [25].

RoboChart also includes probabilistic transitions suggested for UML state machines in [15], with a semantics based on Markov Decision Processes. Probabilistic state machines are used in many robotic applications, such as adaptive foraging and swarming behaviour [18]. Probability in [15] is defined over actions, and the resulting state machines can be analysed using PRISM, a well-established model checker for probabilistic automata [17].

UML [20] state machines are popular. RoboChart, however, is customised to make it suitable for verification and automatic generation of simulations.

In this paper, we formalise the semantics of the core constructs of RoboChart using CSP [23]. Importantly, CSP is a front end for a mathematical model that supports model-checking and theorem proving, namely, Hoare and He's Unifying Theories of Programming [14] (UTP). Use of CSP enables model checking with FDR [11]. On the other hand, the underlying UTP model makes our core semantics adequate for extension to deal with time [28] and probability [31].

RoboChart and its semantics can also be used for the generation of sound simulations that can shed light on the actual behaviours of the robots within various configurations and environment. We describe a general architecture for simulations that can be automatically generated from RoboChart models.

Finally, we describe RoboTool, which provides support for modelling using the RoboChart graphical notation (and its optional textual counterpart), and for verification. Specifically, it provides a code generator that produces CSP specifications defined by the RoboChart semantics for use with FDR.

Chapter 2 describes RoboChart models, and Chapter 3 defines the well-formedness condition of RoboChart models. Chapter 4 presents their semantics in CSP. Chapter 5 describes the API available for modelling robotic and Chapter 6 describes our simulation approach. Chapter 7 presents a number of models specified in RoboChart. Finally, Chapter 8 concludes with a summary of the results and future work.

CHAPTER 2

Syntax

2.1 Introduction

In this section, we first describe the core features of a RoboChart model (Section 2.1.1). To illustrate the concepts, we present the model of a robot for chemical detection based on that in [13] [1]. In our example, the robot employs a random walk and, upon detection of a chemical source, it turns on a light and drops a flag. Sections 2.1.2 and 2.1.3 describe the features to define time and probabilistic properties. Finally, Section 2.2 describes the RoboChart metamodel.

2.1.1 RoboChart diagrams

A robotic system is specified in RoboChart by a module, where a robotic platform is connected to one or more controllers. A robotic platform is characterised by variables, events, and operations representing in-built facilities of the hardware. For our example, the module ChemicalDetector is shown in Figure 2.1, where we have a robotic platform Rover and controllers DetectAndFlagC and LightC.

Rover declares a number of events via named boxes on its border. The events lightOn and lightOff are used to request that the in-built light is turned on or off. The events l and r represent two sensors, one to the left and one to the right. They allow the rover to detect whether there is a wall on either side. The event alarm represents an in-built sensor for the position of a chemical source.

The operation move(v,s) declared in Rover takes a Vector v as parameter; it moves the rover in the direction defined by v with the speed s. In RoboChart, we can define given types (uninterpreted sets), record types, and other structured types. The primitive types include numbers and strings.

[1] `http://tinyurl.com/hdaws7o`

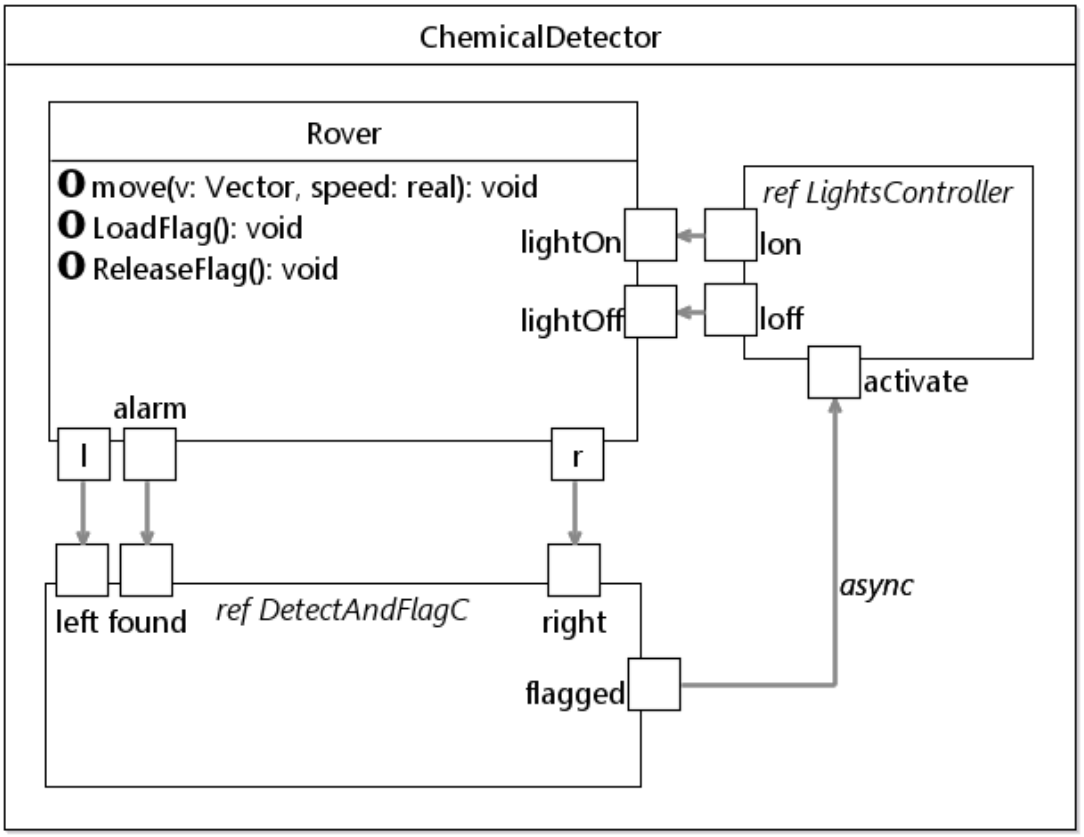


Figure 2.1: Chemical Detector

The definitions of move and Vector are part of the RoboChart API. This API is a collection of types, events, and operations organised in packages by function (moving, sensing, and so on), and, for each function, by the kind of equipment that performs it. The move(v,s) operation is in the package for moving using wheels. The definition of move(v,s) is a simple state machine that declares a precondition: $s > 0$.

The operations LoadFlag() and ReleaseFlag(), on the other hand, are very specific to the particular robotic platform used in this example. They are not part of our API, but are declared and not further defined, since they represent basic functionality provided by the hardware.

The Rover behaviour is defined by two controllers DetectAndFlagC and LightC defined in other diagrams. DetectAndFlagC requires events left, right, found and flagged, shown as bordered boxes. This controller interacts with Rover and LightC through these events, which

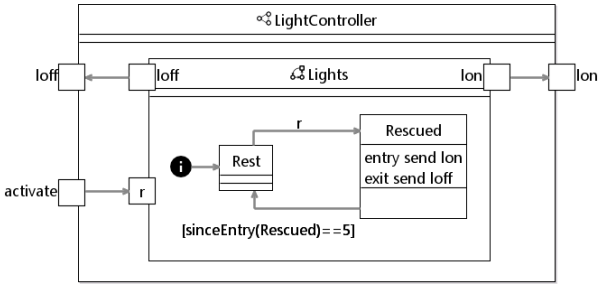


Figure 2.2: Light Controller

are associated with l, r, and alarm of Rover, and activate of LightC, as indicated by the arrows. The directions of the arrows indicate the flow of information. For instance, when the Rover finds a chemical, it sends an alarm to DetectAndFlagC. LightC similarly requires events lon, loff and activate, used to communicate with the Rover and DetectAndFlagC.

Communication with the robotic platform is synchronous, but communication between the controllers can be synchronous or asynchronous. In our example, the communication between DetectAndFlagC and LightC is asynchronous, as indicated by the label async on the arrow that connects their events.

A controller is specified by one or more parallel state machines. In their definitions, variables, events, and operations may be local or global. Required interfaces identify the outer definitions that can be used. Communication between states machines that define a controller is always synchronous, since parallelism at this level is used for convenience of modelling, rather than to indicate concrete designs. In our example, the controllers do not communicate synchronously.

The definition of LightC is shown in Figure 2.2; it consists of a single state-machine that is initially idle, and upon receipt of an event *activate*, iterates between two states every five seconds, sending the events *lon* and *loff* to the robotic platform to switch a light on and off. The diagram for DetectAndFlagC is shown in Figure 2.3. It requires an interface DF_I whose definition is omitted. It declares the operations move, LoadFlag, and ReleaseFlag. In the module ChemicalDetector, where DetectAndFlagC is used, this requirement is satisfied by the robotic platform Rover.

DetectAndFlagC contains three state-machines: DetectAndFlag to specify its behaviour, and DropFlag() and RandomWalk() to specify operations. The event found of this controller corresponds to the event f of DetectAndFlag; flagged to done in DropFlag(); and left and

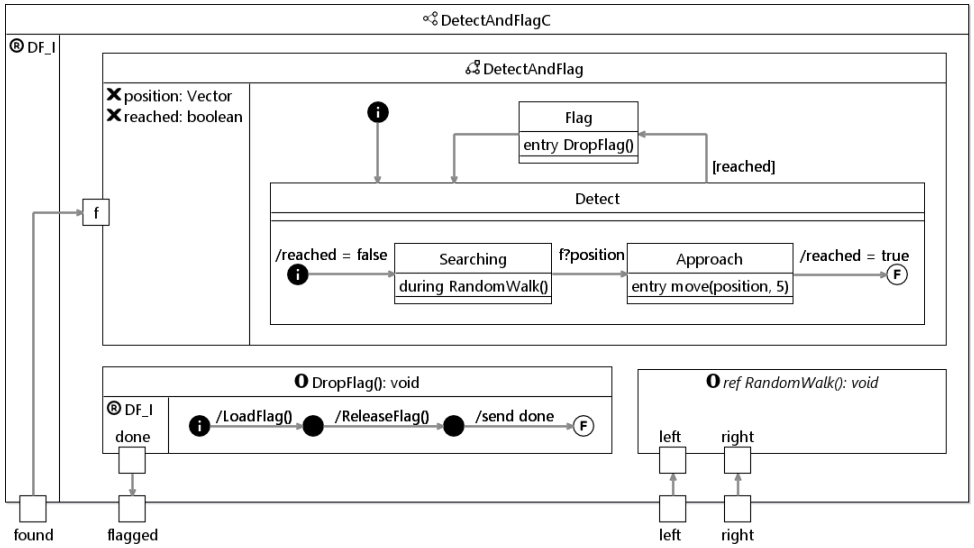


Figure 2.3: Detect and Flag Controller

right to the events of the same name in RandomWalk(). The definition of RandomWalk() is global, since this is an operation relevant for many applications. So, DetectAndFlagC contains a reference to this state machine, which we present in Section 2.1.3.

RoboChart state machines are standard, but restricted and with a well defined semantics. They can have composed states, junctions, and entry, during, exit, and transition actions defined using a well defined action language. Features of UML state machines [19] deemed not essential for robotics are not included, resulting in a streamlined semantics. Figure 2.3 illustrates our notation.

The state-machine DetectAndFlag models the robot roaming behaviour. Two local variables store the position where the chemical is detected and whether that position has been reached. The transition into the initial composed state Detect and its initial state Searching sets reached to false. Its during action is a call to RandomWalk(); this causes the robot to search indefinitely. If the event f, communicating the position of a chemical source, is raised, the machine moves to the state Approach. Its entry action move moves the robot towards the detected source. Once this operation finishes, the machine transitions to its final state, when the variable reached is set to true. This enables the transition from the state Detect to Flag, whose entry action is the operation DropFlag(). The state machine that defines it, also in Figure 2.3, has just junctions. The transitions defines that the operations LoadFlag and ReleaseFlag are

used to select and release a flag next to the chemical source. When DropFlag completes, in DetectAndFlag, the transition from Flag to Detect is taken and the robot restarts the search.

2.1.2 Time primitives

RoboChart operations take zero time, and enabled transitions take place as soon as they can be triggered. So, time constraints need to be explicitly defined. The time budget t for an operation call can be specified by sequentially composing it with the primitive wait(t), that waits for t time units. A deadline of d time units for a statement S is specified by S <{d}. Deadlines can also be set on events.

Clocks provide a way to associate the instant in time #T in which a transition is triggered with subsequent conditions on this instant. The primitive since(T) yields the time elapsed since the most recent time instant #T. A similar primitive sinceEntry(s) yields the time elapsed since entering state s. Finally, the transition primitive e@t records the time elapsed between the moment the event e is available and when the transition it guards is triggered.

To illustrate the time primitives we extend DetectAndFlag. In this version, shown in Figure 2.4, the additional states Charging, MovingToBase and Error model a robot that can charge its battery on a base station, whose location is fixed by a constant base. The robot can perform different actions depending on how much charge is available, as recorded by the variable battery. We assume that the robot cannot measure this quantity directly, but instead can estimate battery drain using time and an estimate of how much charge is consumed over time.

The state-machine starts at a junction, with outgoing transitions guarded by conditions on the variable battery: if battery is lower than the charge required to reach the base, as estimated by the operation PowerTo (omitted here), then the state-machine transitions to the state Error, otherwise it transitions to MovingToBase. In Error, the event error is sent (to LightC), which turns on the light to indicate that there is a problem. When the robot reaches the base, the state machine transitions to Charging, whose entry action models the charging activity: the primitive wait(10) models the fact that it takes 10 time units to charge the battery to its full capacity of 1000 mAh.

Once the battery is charged, a transition leads to the state Searching. There are now two possibilities: either the robot finds a chemical source, as signalled by the event f, or, if it takes more than 10 time units, then the walk is interrupted and another attempt is made to return

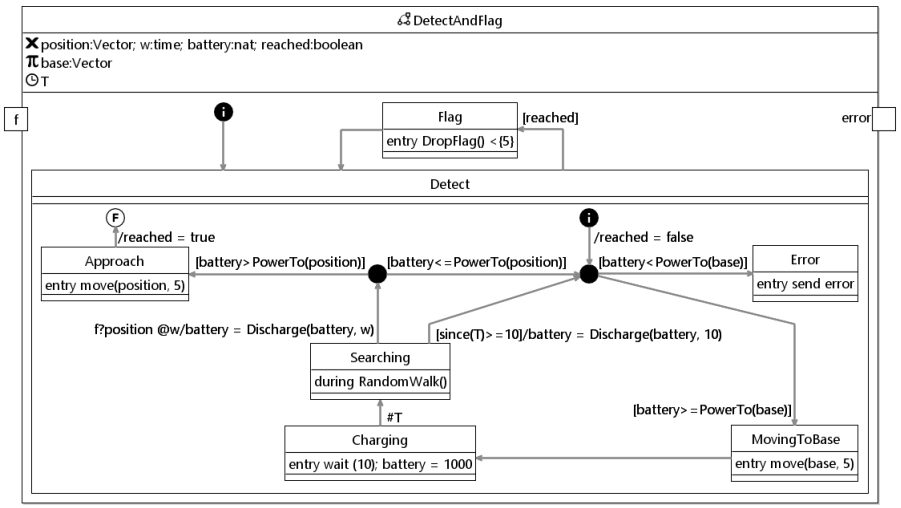


Figure 2.4: Timed Chemical Detector

to the base. The transition to the initial junction updates the estimated battery charge using the operation Discharge (whose definition is omitted here). To constrain the time allowed for the random walk, we associate with the incoming transition to Searching a timed instant #T, which is related to a modelling clock T. The transition from Searching to the initial junction is guarded by a condition on the time primitive since(T).

In case the robot finds a chemical source during the walk, signalled by f, we record in the variable w the amount of time elapsed since f was available, in this case immediately upon entering Searching, and its occurrence. This is used by Discharge to calculate a new estimate for the battery charge based on the time w. Moreover, the transition to Approach is guarded by a condition that requires that there is enough battery charge to reach the position, as estimated by PowerTo, otherwise the robot attempts to move back to the base. Finally, when the robot finishes approaching the chemical source, it transitions to the state Flag, where there is a deadline imposed on the operation DropFlag to terminate within 5 time units, so that the robot can quickly leave the place.

2.1.3 Probability primitives

RoboChart has just a single additional construct to cope with modelling of probabilistic behaviour, taken from [15]. It is the P-node: a junction, inscribed with the letter P, and a number of outgoing transitions. Each outgoing transition is labelled with an expression p{e} that de-

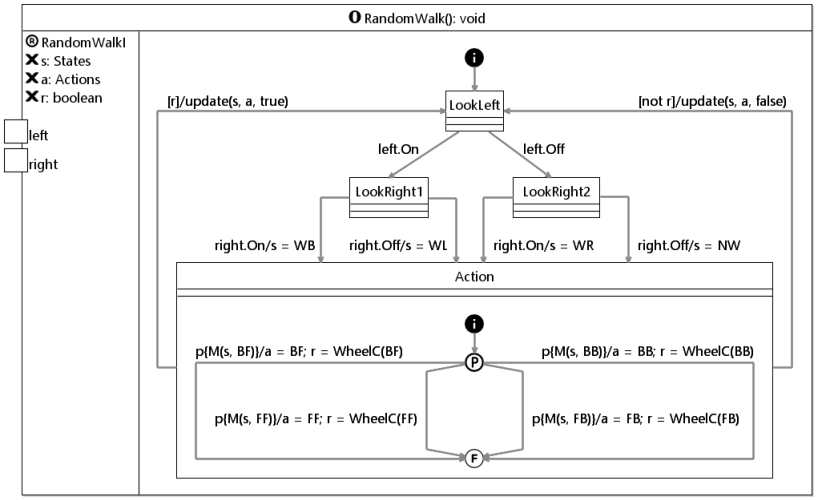


Figure 2.5: Random walk State-machine

notes a probability, together with an optional action. The probabilities for the outgoing edges must sum to 1.0.

In Figure 2.5, we describe the operation RandomWalk(). This state machine has a local variable s, which can take values WB, to indicate that there is a wall on both sides (that is, the rover is facing a corner), WL, if there is just a wall to the left, WR, for a wall to the right, or NW, if there is no adjacent wall. Another variable a takes as values actions corresponding to actuation of the rover's driving wheels: forwards or backwards. FF takes the rover forward; FB turns right; BF turns left; and BB reverses. Finally, the variable M is a matrix: M[s,a] shows the probability of the action a occurring in the scenario s.

The transition into the initial state LookLeft initializes M to give probability of $\frac{1}{4}$ to all actions in all scenarios using init(). The omitted operation init() is in the required interface RandomWalkI, which also includes two other operations. WheelC(a) controls the wheels to take the input action a and has a boolean result that indicates whether the action completes without hitting anything or not. Finally, update(s,a,b) updates M to record that in scenario s the action a is good or bad, as indicated by the boolean b. If a is good, then its probability in s is increased, at the expense of the other actions. If it was bad, then its probability is decreased, at the benefit of the others. The precise changes in probabilities can be varied to produce different behaviours.

In LookLeft, the rover decides how to move based on the events left and right of the wall sensors. These events are used to record the position of the walls in s. The action that follows

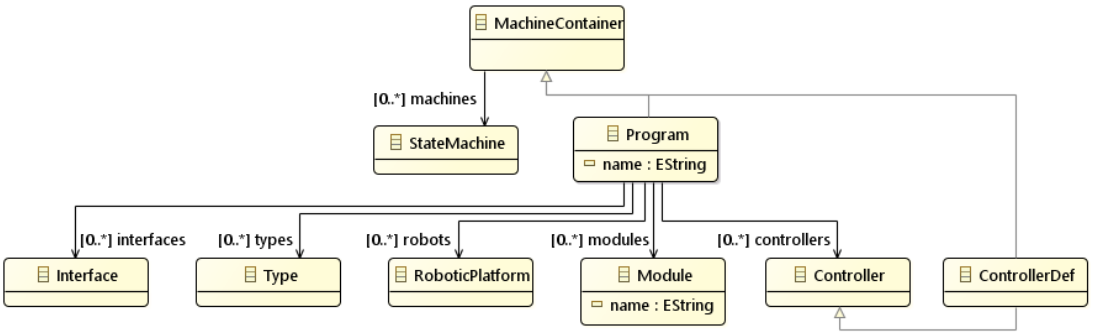


Figure 2.6: Metamodel of RoboChart models

depends on the history of successful actions recorded in M. The probabilistic choice defined by the P-node selects an action a with probability M[s,a] and executes it using WheelC. The fact that alternatives in the probabilistic choice sum to 1.0 is enforced through a row invariant in M. Once WheelC finishes, a transition out of the state Action back to LookLeft is taken, when the matrix M is updated using the operation update.

2.2 RoboChart metamodel

As illustrated above, RoboChart models are structured using elements shown in the metamodel in Figure 2.6, namely, modules, robotic platforms, controllers, state machines, interfaces, and types. Modules give a complete account of a robotic system. They define robotic platforms or include references to platforms defined elsewhere to indicate the robots available. Modules associate their robotic platforms with particular controllers and state machines to specify the behaviours of the robots. State machines can be directly associated to robotic platforms, but, when the behaviour is complex and is specified by multiple (potentially interacting) state machines, controllers can be used.

A module comprises a number of module nodes, which can be controllers, robotic platforms and state-machines, and connections between the nodes that establish the relationship between platforms and their specified behaviours.

A controller encapsulates state machines that can communicate with each other and the external environment through synchronous events. In this way, simpler behaviours can be co-

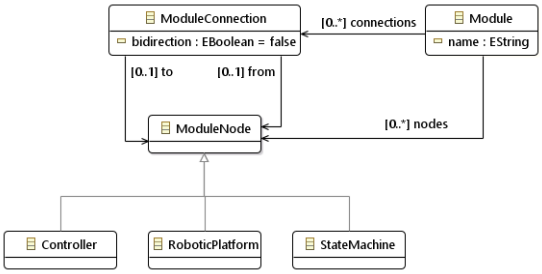


Figure 2.7: Metamodel of RoboChart modules

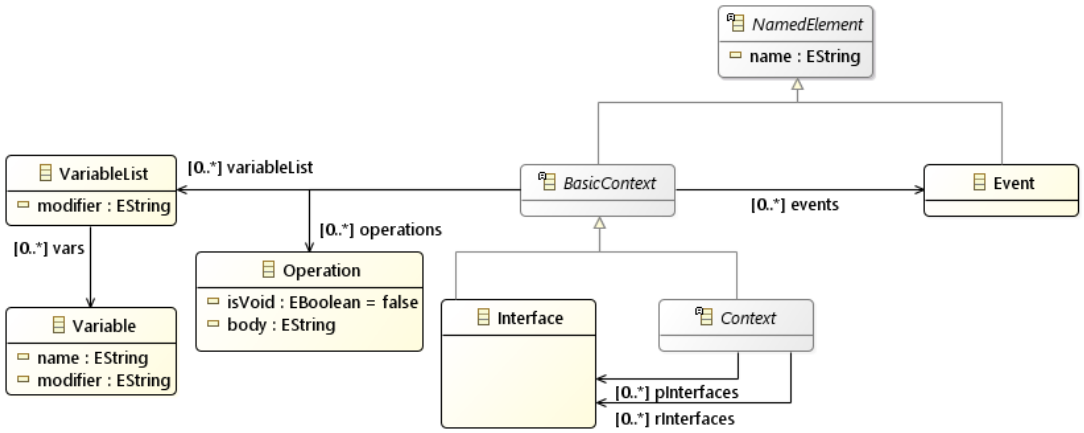


Figure 2.8: Metamodel of RoboChart constructs

ordinated to model a complex controller. Robotic platforms, controllers and state machines share features such as variables, operations, events, and provided and required interfaces.

The metamodel of RoboChart state machines is similar to that of UML state machines. Features that have been removed are parallel regions, history junctions, and interlevel transitions. Whilst the state machines are designed with sequential controllers (which may be in parallel with other controllers) in mind, there is space for parallelism in the execution of during actions (see Section 4.1). When state machines are used to specify the implementation of operations, they declare an operation signature, and may also specify pre and postconditions.

Expressions and statements include time primitives, such as wait. Triggers (for state transitions) include probability as well as time recording and deadline primitives (for example, #T). In addition, the metamodel for state machines includes P-nodes. These are simple variations of the usual metamodel.

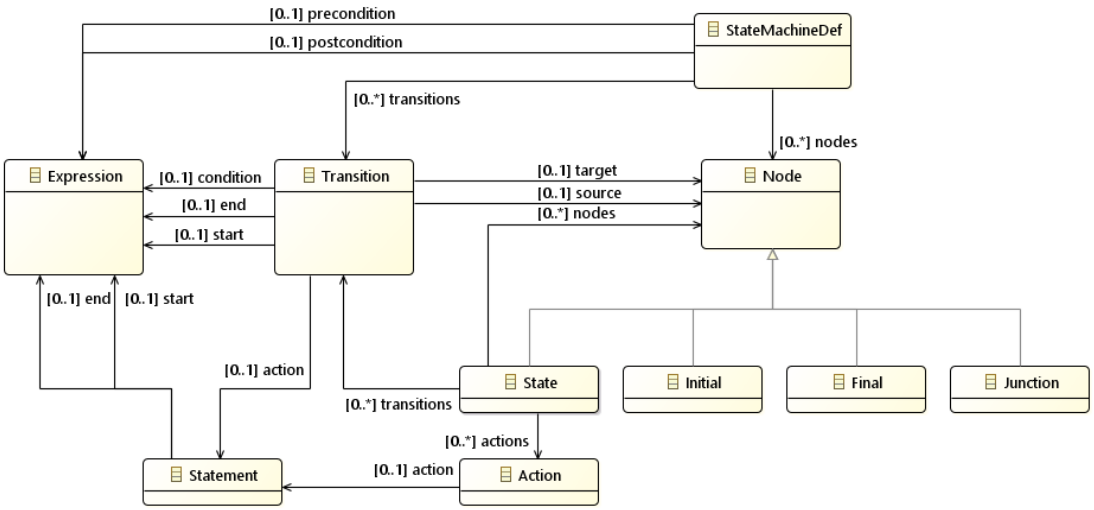


Figure 2.9: Metamodel of RoboChart state machines

2.3 Complete Metamodel: Core Language

Our core notation is a state-machine based language with specific components that provide for sequential behaviours and parallelism in a restricted manner. Essentially, state-machines are intended to specify sequential behaviours, whilst parallelism is modelled by controllers. The top-level components of a *RoboChart* specification is a module, which represents a single robot recording assumptions about the hardware as well as the controlling software.

2.3.1 Robotic Platforms

A robotic platform is characterised by variables, events, and operations representing in-built facilities of the hardware. It represents the observables interactions between the robot, its environment and controller.

```
class RoboticPlatform extends ModuleNode;
class RoboticPlatformDef extends NamedElement,Context,RoboticPlatform;
class RoboticPlatformRef extends NamedElement,RoboticPlatform {
  property ref : RoboticPlatformDef[?];
}
abstract class NamedElement {
  attribute name : String[?];
```

```
}
abstract class BasicContext {
  property variableList : VariableList[*|1] { ordered composes };
  property operations : Operation[*|1] { ordered composes };
  property events : Event[*|1] { ordered composes };
}
abstract class Context extends BasicContext {
  property pInterfaces : Interface[*|1] { ordered !unique };
  property rInterfaces : Interface[*|1] { ordered !unique };
}
```

Interfaces

 An interface encapsulates events and variables declarations as well as operation signatures. Interfaces are used to record information about the assumptions a component makes, and what assumptions can be made about a component.

```
class Interface extends NamedElement,BasicContext;
```

Variables

 An variable can be declared in interfaces, robotic platforms, controllers and state-machines. Variable declarations in interfaces are used to validate a model with respect to assumptions about definition and usage of variables in the context of a module (composition of controllers and robotic platforms). Variables declared in controllers and robotic platforms are shared among its associated elements (state-machines and controllers, respectively), but are ultimately only used by state-machines. State-machines can themselves declare variables that are local to the state-machine. Variables are typed and may declare an initial value.

```
class VariableList {
  attribute modifier : String[?];
  property vars : Variable[*|1] { ordered composes };
}
class Variable extends NamedExpression {
  attribute name : String[?];
  property type : Type[?] { composes };
```

```
  property initial : Expression[?] { composes };
  attribute modifier : String[?] { derived transient volatile };
}
```

Constants

 A constant is similar to a variable except that its value cannot be changed. Besides its usage as a meaningful name or abbreviation for otherwise complex or meaningless values, it can also be used without a concrete value to indicate a loose values that is fixed but unspecified.

```
Variable with modifier == 'con'
```

Inputs

 An input variable can be used in state-machines to model the interactions from the environment to the state-machine via a stream of data instead of explicit events. While this feature is available in the abstract syntax of the language it is not strictly necessary because it can be easily replaced by events that carry types. It will however become relevant as hybrid state-machines are taken into consideration.

```
Variable with modifier == 'in'
```

Outputs

 An output variable can be used in state-machines to model the interactions from the state-machine to its environment via a stream of data instead of explicit events. While this feature is available in the abstract syntax of the language it is not strictly necessary because it can be easily replaced by events that carry types. It will however become relevant as hybrid state-machines are taken into consideration.

```
Variable with modifier == 'out'
```

Events

An event are the main form of interaction between a state-machine and its environment, be it other state-machines, controllers or the robotic platform. Events can be typed or untyped, where typed events carry values, and untyped events model a simple interaction where no extra information can be inferred except that two parallel components interacted. Whilst events are not explicitly divided between input and output events, their roles are exclusive (events cannot be used as both inputs and outputs) and are determined from the connections in the model. The connections between events (not the events themselves) determine if the communication takes place synchronously or asynchronously.

```
class Event extends NamedElement {
  property type : Type[?] { composes };
}
```

Required Interfaces

A required interface specifies the assumptions a state-machine or controller makes about the environment, the robotic platform and other controllers. It is used to declare abstract controllers and state-machines that do not depend on specific platforms, only on specific operations, events and variables. It is worth mentioning that required interfaces can be used to specify assumptions about the kind of state variables are available in a robotic plaform. This allows for instance the specification of movement operations independently of the particular platform based solely on the assumption that a potential target platform supports changing linear and angular speed by setting specific variables.

```
rInterfaces of Context
```

Provided Interfaces

A provided interface specifies what assumptions can be made about a robotic platform or controller. It is used mainly to validate the well-formedness of controllers and modules by guaranteeing that the assumptions made (through required interfaces) are actually satisfied by some component in the composition.

```
pInterfaces of Context
```

2.3.2 State-Machines

A state-machine is the construct dedicated for the specification of sequential behaviours. It contains a number of nodes that represent steps (stable or not) of the behaviour and transitions that describe when and how control is transferred from one node to another.

```
class StateMachine extends ModuleNode;
class StateMachineDef extends NamedElement,StateMachineBody,StateMachine;
class StateMachineRef extends Node,StateMachine {
  property ref : StateMachineDef[?];
}
class StateMachineBody extends Context,NodeContainer {
  property clocks : Clock[*|1] { ordered composes };
}
class Node extends NamedElement;
```

States

A state is one of the main components of a state-machine. It describes a stable configuration of the state-machine and has three distinctive phases in its life-cycle: *entering*, *executing* and *exiting*. Each of these phases has an associated action: *entry*, *during* and *exit* actions.

States are divided between simple and composite states. Simple states can only contain the actions mentioned above, whilst composite states can themselves contain states as well as transitions and other nodes.

```
class State extends Node,NodeContainer {
  property actions : Action[*|1] { ordered composes };
}
class NodeContainer {
  property nodes : Node[*|1] { ordered composes };
  property transitions : Transition[*|1] { ordered composes };
}
```

Initial nodes

An initial node represents en entry point of a state-machine or composite state. It indicate where the state-machine or composite state must start executing to enter its substates.

The main validation rule related to initial nodes is that any state-machine or composite state (state with one or more subnodes) must have exactly one initial node.

```
class Initial extends Node;
```

Junction node

A junction node represents an unstable configuration of the state-machine. Unlike in a (stable) state, the state-machine cannot rest and execute other behaviours (actions, substates, etc) while in a junction node. At this point, all it can do it follow one of the outgoing transitions.

In order to guarantee that execution can progress once in a junction node, two well-formedness condition are defined. The first requires that there are event triggers in the outgoing transitions, and the second requires that the outgoing transitions form a cover, that is, the conjunction of all their guards is equivalent to true. Notice that we do not require them to be disjoint as the selection of outgoing transition may be non-deterministic.

```
class Junction extends Node;
```

Final states

A final state represents the completion of the internal behaviours of a state-machine or composite state. While the meaning of a final state is the same in both cases, state-machines and composite states react differently to reaching a final state. While a composite state rests in the final state and waits for one of its transitions (or the parents transitions) to be executed, the state-machine terminates as soon as the final state is reached.

```
class Final extends Node;
```

Transitions

A transition defines one possible path between to nodes in a state-machine or composite state. It contains source and target nodes as well as, optionally, a trigger in the form of an event, a boolean condition, and an action. The transition can only be executed if the event in the trigger is available and the condition is true.

The transition action is executed after the source state (if it exists) is exited, but before the target state (if it exists) is entered. Notice that source and target states are not necessarily available. For example transitions between junction nodes have neither.

```
class Transition extends NamedElement {
  property source : Node[?];
  property target : Node[?];
  property trigger : Trigger[?] { composes };
  property condition : Expression[?] { composes };
  property action : Statement[?] { composes };
}
class Trigger {
  property time : Variable[?];
  property event : Event[?];
}
class InputTrigger extends Trigger {
  attribute parameter : String[?];
}
class OutputTrigger extends Trigger {
  property value : Expression[?] { composes };
}
class SyncTrigger extends Trigger {
  property value : Expression[?] { composes };
}
class SimpleTrigger extends Trigger;
```

2.3.3 Controllers

A controller models a collection of potentially parallel cooperating state machines; it can be used, for instance, to encapsulate specific well-defined functionalities that are implemented by multiple state machines. Controllers are the elements of RoboChart that interact directly with robotic platform.

```
class Controller extends ModuleNode;
class ControllerDef extends NamedElement,Context,Controller,
    MachineContainer {
  property connections : CtrlConnection[*|1] { ordered composes };
}
class ControllerRef extends NamedElement,Controller {
  property ref : ControllerDef[?];
}
```

Controller Connection

A controller connection is a link between events within (and on the boundary) of a controller. These connections are used to specify the interactions between state machines in a controller as well as the interactions between the state machines and the controller itself.

```
class CtrlConnection {
  property from : ModuleNode[?];
  property efrom : Event[?];
  property to : ModuleNode[?];
  property eto : Event[?];
}
```

2.3.4 Modules

```
class Module {
  attribute name : String[?];
  property connections : ModuleConnection[*|1] { ordered composes };
  property nodes : ModuleNode[*|1] { ordered composes };
}
```

```
class ModuleNode;
```

Module Connection

→ A module connection is similar to a controller connection, except that it is contained within a module and is used to specify the interactions between controller in a module as well as the interactions between the controllers and the robotic platform.

```
class ModuleConnection {
  property from : ModuleNode[?];
  property efrom : Event[?];
  property to : ModuleNode[?];
  property eto : Event[?];
  attribute async : Boolean[?];
}
```

2.3.5 Statements

```
class Statement;
class OpCall extends Statement;
class Skip extends Statement;
class IfStmt extends Statement {
  property expression : Expression[?] { composes };
  property _'then' : Statement[?] { composes };
  property _'else' : Statement[?] { composes };
}
class Assignment extends Statement {
  property left : Assignable[?] { composes };
  property right : Expression[?] { composes };
}
class SendEvent extends Statement {
  attribute async : Boolean[?];
  property event : Event[?];
  property value : Expression[?] { composes };
}
class Return extends Statement {
```

```
  property expression : Expression[?] { composes };
}
class SeqStatement extends Statement {
  property statements : Statement[*|1] { ordered composes };
}
class ParStmt extends Statement;
class Call extends OpCall {
  attribute async : Boolean[?];
  property member : Operation[?];
  property args : Expression[*|1] { ordered composes };
}
```

2.3.6 Expressions

```
class Expression;
class ArrayExp extends Expression {
  property value : Expression[?] { composes };
  property parameters : Expression[*|1] { composes };
}
class Iff extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Implies extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Or extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class And extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Not extends Expression {
```

```
  property exp : Expression[?] { composes };
}
class Equals extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Different extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class GreaterThan extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class GreaterOrEqual extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class LessThan extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class LessOrEqual extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Plus extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Minus extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Modulus extends Expression {
  property left : Expression[?] { composes };
```

```
  property right : Expression[?] { composes };
}
class Mult extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Div extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Cat extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Neg extends Expression {
  property exp : Expression[?] { composes };
}
class Selection extends Expression {
  property receiver : Expression[?] { composes };
  property member : Member[?];
}
class IntegerExp extends Expression {
  attribute value : ecore::EInt[?];
}
class FloatExp extends Expression {
  attribute value : ecore::EFloat[?];
}
class StringExp extends Expression {
  attribute value : String[?];
}
class BooleanExp extends Expression {
  attribute value : String[?];
}
class VarExp extends Expression {
  property value : Variable[?];
}
```

```
class RefExp extends Expression {
  property value : NamedExpression[?];
}
class EnumExp extends Expression {
  property type : Enumeration[?];
  property constant : Constant[?];
}
class ParExp extends Expression {
  property exp : Expression[?] { composes };
}
class RangeExp extends Expression {
  attribute linterval : String[?];
  property lrange : Expression[?] { composes };
  property rrange : Expression[?] { composes };
  attribute rinterval : String[?];
}
class CallExp extends Expression {
  attribute async : Boolean[?];
  property member : Operation[?];
  property args : Expression[*|1] { ordered composes };
}
```

2.3.7 Type Declaration

```
class TypeDecl extends NamedElement;
```

Primitive Types

A primitive type ....

```
class PrimitiveType extends TypeDecl;
```

Datatypes

A datatype ....

```
class DataType extends TypeDecl {
  property fields : Field[*|1] { ordered composes };
}
```

A field ....

```
class Field extends Member,NamedExpression;
```

Enumeration

```
class Enumeration extends TypeDecl {
  property constants : Constant[*|1] { ordered composes };
}
class NamedExpression;
class Constant extends NamedElement,NamedExpression {
  property type : Enumeration[?];
}
```

2.3.8 Type Constructors

```
class Type;
class ProductType extends Type {
  property types : Type[*|1] { ordered composes };
}
class FunctionType extends Type {
  property domain : Type[?] { composes };
  property range : Type[?] { composes };
}
class SetType extends Type {
  property domain : Type[?] { composes };
}
```

```
class SeqType extends Type {
  property domain : Type[?] { composes };
}
class TypeRef extends Type {
  property ref : TypeDecl[?];
}
```

2.4 Complete Metamodel: Timed Language

Clock


A clock ....

```
class Instant {
  property instant : Clock[?];
}
class Clock {
  attribute type : String[?];
  attribute name : String[?];
}
```

Timed Statements

```
class Statement {
  property start : Expression[?] { composes };
  property stmt : Statement[?] { composes };
  property end : Expression[?] { composes };
}
class Wait extends Statement {
  property duration : Expression[?] { composes };
}
```

Timed Expressions

```
class ClockExp extends Expression {
```

```
  property instant : Clock[?];
}
class StateClockExp extends Expression {
  property state : State[?];
}
```

Timed Triggers

```
class Trigger {
  property time : Variable[?];
  property instant : Instant[*|1] { ordered composes };
  property event : Event[?];
}

class Transition extends NamedElement {
  property source : Node[?];
  property target : Node[?];
  property start : Expression[?] { composes };
  property trigger : Trigger[?] { composes };
  property end : Expression[?] { composes };
  property condition : Expression[?] { composes };
  property action : Statement[?] { composes };
}
```

2.5 Complete Metamodel: Probabilistic Language

2.5.1 Probabilistic Junction

A probabilistic junction ....

```
class ProbabilisticJunction extends Node;
```

2.5.2 Probabilistic Transition

```
class Transition extends NamedElement {
```

```
  property source : Node[?];
  property target : Node[?];
  property start : Expression[?] { composes };
  property trigger : Trigger[?] { composes };
  property end : Expression[?] { composes };
  property condition : Expression[?] { composes };
  property action : Statement[?] { composes };
  property probability : Expression[?] { composes };
}
```

2.6 Complete Metamodel: Hybrid Language

2.7 Tool Support

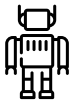
Chapter 3

Well-formedness Conditions

The meta-model presented in the previous section accepts models that are not meaningful. In order to restrict the set of meaningful models it is necessary to define a number of well-formedness conditions. These conditions encode restrictions that are necessary for an adequate semantics to be defined. For example, condition 2 for transitions establishes that there are no interlevel transitions and this is necessary to improve the compositionality of the semantics.

3.1 Core Language

3.1.1 Robotic Platforms



The well-formedness conditions associated with robotic platforms involve the declaration of elements of the provided interfaces.

$$WF\_RoboticPlatform : \mathbb{P}\, RoboticPlaftorm$$

$$\forall rp : RoboticPlatform \bullet rp \in WF\_RoboticPlatform \Leftrightarrow \forall i : rp.pInterfaces \bullet$$
$$\left(\begin{array}{l} \forall v : \bigcup_{l:i.variableList} l.variables \bullet v \in \bigcup_{l:rp.variableList} l.variables \wedge \\ \forall e : i.events \bullet v \in rp.events \wedge \\ \forall op : i.operations \bullet (v \in rp.operations \wedge \exists op_{Def} : OperationDef \bullet op_{Def} =_{Sig} op) \end{array} \right)$$

Interfaces

The only well-formedness conditionsfor interfaces requires uniqueness of names of variables, events and operations.

$$WF\_Interface : \mathbb{P}\, Interface$$

$$\forall i : Interface \bullet i \in WF\_Interface \Leftrightarrow \left(\begin{array}{l} \forall x, y : \bigcup_{l:i.variableList} l.variables \mid x \neq y \bullet x.name \neq y.name \wedge \\ \forall x, y : i.operations \mid x \neq y \bullet x.name \neq y.name \wedge \\ \forall x, y : i.events \mid x \neq y \bullet x.name \neq y.name \end{array} \right)$$

VariableList

The only well-formedness condtion for variable lists requires that the names of variables in a list are unique. This well-formedness apply equally to variables, constants, inputs and outputs.

$$WF\_VariableList : \mathbb{P}\, VariableList$$

$$\forall l : VariableList \bullet l \in WF\_VariableList \Leftrightarrow \forall x, y : l.variables \mid x \neq y \bullet x.name \neq y.name$$

3.1.2 State-Machines

State-machines have a number of well-formedness conditions, which are described here separately indexed by a natural number.

$$WF\_StateMachineDef_1 : \mathbb{P}\, StateMachineDef$$

$$\forall stm : StateMachineDef \bullet stm \in WF\_StateMachineDef_1 \Leftrightarrow \exists_1 n : Initial \bullet (n \in stm.nodes \wedge \exists_1 t : Transition \mid t \in stm.transitions \bullet t.source = n \wedge t.guard \wedge t.trigger = \emptyset)$$

$$RVariables : Context \rightarrow \mathbb{P}\, Variable$$

$$\forall c : Context \bullet RVariables\, c = \bigcup_{i:c.rInterfaces} \bigcup_{l:i.variableList} l.variables$$

$$REvents : Context \rightarrow \mathbb{P}\, Events$$

$$\forall c : Context \bullet REvents\, c = \bigcup_{i:c.rInterfaces} i.events$$

$$Variables : Context \to \mathbb{P}\, Variable$$

$$\forall\, c : Context \bullet Variables\, c = \bigcup_{l:stm.variableList} l.variables$$

$$WF\_StateMachineDef_2 : \mathbb{P}\, StateMachineDef$$

$$\forall\, stm : StateMachineDef \bullet stm \in WF\_StateMachineDef_2 \Leftrightarrow \left(\begin{array}{l} \forall\, x, y : Variables\, stm \mid x \neq y \bullet x.name \neq y.name \wedge \\ \forall\, x, y : i.events \mid x \neq y \bullet x.name \neq y.name \end{array} \right)$$

$$WF\_StateMachineDef_3 : \mathbb{P}\, StateMachineDef$$

$$\forall\, stm : StateMachineDef \bullet stm \in WF\_StateMachineDef_3 \Leftrightarrow \left(\forall\, x : Variables\, stm \mid x \notin RVariables\, stm \wedge \right)$$

$$WF\_StateMachineDef_4 : \mathbb{P}\, StateMachineDef$$

$$\forall\, stm : StateMachineDef \bullet stm \in WF\_StateMachineDef_4 \Leftrightarrow inputE(stm) \cap outputE(stm) = \emptyset$$

$$inputE : StateMachineDef \to \mathbb{P}\, Event$$

$$\forall\, stm : StateMachineDef \bullet inputE\, stm = \{e : Event \bullet \exists\, t : Trigger \mid t \in triggers\, stm \bullet e = t.event\}$$

$$outputE : StateMachineDef \to \mathbb{P}\, Event$$

$$\forall\, stm : StateMachineDef \bullet outputE\, stm = \{e : Event \bullet \exists\, s : SendEvent \mid s \in statements\, stm \bullet e = s.event\}$$

$$triggers : NodeContainer \to \mathbb{P}\, Trigger$$

$$\forall\, n : NodeContainer \bullet triggers\, n = t : Transition \mid t \in n.Transitions \bullet t.trigger \cup \bigcup_{x:n.nodes} triggers\, x$$

$$
\begin{array}{l}
statements : NodeContainer \to \mathbb{P}\, Statement \\
stms : Statement \to \mathbb{P}\, Statement \\
\hline
\forall\, nc : NodeContainer \bullet \\
\quad statements\, nc = (\textbf{if } nc \in State \textbf{ then } \bigcup_{a:nc.actions} stms\, a.statement \textbf{ else } \{\}) \\
\quad \cup \bigcup_{n:nc.nodes} statements\, n \\
\forall\, s : Statement \bullet \\
\quad stms\, s = (\textbf{if } s \in SeqStatement \textbf{ then } \bigcup_{x:s.statements} stms\, x \textbf{ else } \{s\})
\end{array}
$$

States

A state has similar well-formedness conditions to State-Machines when it has substates.

$$
\begin{array}{l}
WF\_State_1 : \mathbb{P}\, State \\
\hline
\forall\, s : State \bullet s \in WF\_State_1 \Leftrightarrow \\
s.nodes \neq \emptyset \Rightarrow \\
\quad \exists_1\, n : Initial \bullet (n \in stm.nodes \land \\
\qquad \exists_1\, t : Transition \mid t \in stm.transitions \bullet t.source = n \land t.guard \land t.trigger = \emptyset \\
\quad)
\end{array}
$$

Additionally, it has at most one of each type of action.

$$
\begin{array}{l}
WF\_State_2 : \mathbb{P}\, State \\
\hline
\forall\, s : State \bullet s \in WF\_State_2 \Leftrightarrow \\
\quad \#\{a : s.actions \mid a \in EntryAction\} \leq 1 \land \\
\quad \#\{a : s.actions \mid a \in DuringAction\} \leq 1 \land \\
\quad \#\{a : s.actions \mid a \in ExitAction\} \leq 1
\end{array}
$$

Initial nodes

An initial node does not have incoming transitions. Additionally, it contains at least one outgoing transitions and the guards of the outgoing transitions form a cover.

$$WF\_Initial_2 : \mathbb{P}\, Initial$$

$$\begin{aligned}
&\forall i : Initial \bullet i \in WF_Initial_2 \Leftrightarrow \\
&\quad \neg\ \exists t : Transition \mid t.target = i \land \\
&\quad \#\{t : Transition \mid t.source = i\} > 0 \land \\
&\quad \lor \{t : Transition \mid t.source = i \bullet t.guard\}
\end{aligned}$$

Junction node

A junction node must contain at least one outgoing transition, and the disjunction of the transition's guards is true.

$$WF\_Junction_2 : \mathbb{P}\, Junction$$

$$\begin{aligned}
&\forall j : Junction \bullet j \in WF_Junction_2 \Leftrightarrow \\
&\quad \#\{t : Transition \mid t.source = j\} > 0 \land \\
&\quad \#\{t : Transition \mid t.target = j\} > 0 \land \\
&\quad \lor \{t : Transition \mid t.source = j \bullet t.guard\}
\end{aligned}$$

Final states

An final node does not have outgoing transitions.

$$WF\_Final_2 : \mathbb{P}\, Final$$

$$\begin{aligned}
&\forall f : Final \bullet f \in WF_Final_2 \Leftrightarrow \\
&\quad \neg\ \exists t : Transition \mid t.source = f \land \\
&\quad \#\{t : Transition \mid t.target = f\} > 0 \land
\end{aligned}$$

Transitions

Only transitions starting in a state can have triggers, and the ends of a transitions must belong to the same container.

$$WF\_Transition_1 : \mathbb{P}\, Transition$$

$$\forall\, t : Transition \bullet t \in WF\_Transition_1 \Leftrightarrow \#t.trigger > 0 \Rightarrow t.source \in State$$

$$WF\_Transition_2 : \mathbb{P}\, Transition$$

$$\forall\, t : Transition \bullet t \in WF\_Transition_2 \Leftrightarrow \exists_1\, nc : NodeContainer \bullet t.source \in nc.nodes \land t.target \in nc.nodes$$

3.1.3 Controllers

A controller has at least one state-machine.

$$WF\_Controller_1 : \mathbb{P}\, Controller$$

$$\forall\, c : Controller \bullet c \in WF\_Controller_1 \Leftrightarrow \#c.nodes > 0$$

Controller Connection

There are no well-formedness conditions associated with controller connection yet.

3.1.4 Modules

A module has exactly one robotic platform and at least one controller.

$$WF\_Module_1 : \mathbb{P}\, Module$$

$$\forall\, m : Module \bullet m \in WF\_Module_1 \Leftrightarrow \#\{n : m.nodes \mid n \in RoboticPlatform\} = 1 \land \#\{n : m.nodes \mid n \in Controller\} > 0$$

Module Connection

 There are no well-formedness conditions associated with module connection yet.

3.1.5 Statements

There are no well-formedness conditions associated with statements yet.

3.1.6 Expressions

There are no well-formedness conditions associated with expressions yet.

3.1.7 Type Declaration

There are no well-formedness conditions associated with type declarations yet.

Primitive Types

 There are no well-formedness conditions associated with primitive types yet.

Datatypes

 There are no well-formedness conditions associated with datatype yet.

 There are no well-formedness conditions associated with field yet.

Enumeration

There are no well-formedness conditions associated with enumerations yet.

3.1.8 Type Constructors

There are no well-formedness conditions associated with type constructors yet.

3.2 Timed Language

Clock

There are no well-formedness conditions associated with clocks yet.

Timed Statements

There are no well-formedness conditions associated with timed statements yet.

Timed Expressions

There are no well-formedness conditions associated with timed expressions yet.

Timed Triggers

There are no well-formedness conditions associated with timed triggers yet.

3.3 Probabilistic Language

3.3.1 Probabilistic Junction

There are no well-formedness conditions associated with probabilistic junctions yet.

3.3.2 Probabilistic Transition

There are no well-formedness conditions associated with probabilistic transitions yet.

3.4 Hybrid Language

3.5 Tool Support

Chapter 4

Semantics

4.1 Behavioural semantics and verification

To support the verification of models and the generation of sound simulations, we give RoboChart a formal semantics. In this section, we discuss the semantics of the core RoboChart notation (Section 4.1.1) and its use for verification (Section 4.1.2). As already mentioned, our formalisation relies on the UTP framework, and on its expressiveness to cater for time and probability in an integrated way. On the other hand, for the core notation, CSP provides the ideal level of abstraction and support for validation via model checking.

4.1.1 CSP semantics

The semantics of a module is given by the parallel composition of the processes that define the semantics of its controllers interacting according to the connections in the module. We show below the semantics of ChemicalDetector, in terms of processes for the controllers LightC and DetectAndFlagC.

$$
\begin{array}{lll}
ChemicalDetector = & (Controllers \\
& \quad [|\{|activate, flagged|\}|] \\
& Buffer[flagged, activate/in, out]) \setminus [|\{|activate, flagged|\}|] \\
\\
Controllers = & LightC[lightOn, lightOff/lon, loff] \\
& \quad ||| \\
& DetectAndFlagC[l, r, alarm/left, right, found]
\end{array}
$$

Synchronous interactions such as the one between events *lon* and *lightOn* are modelled by renaming the events of the controller to those of the robotic platform. On the other hand, asynchronous interaction such as that between *activate* and *flagged* is realised via a buffer that runs in parallel ([|..|]) with the controllers, synchronising on these events. The occurrences of

flagged are inputs to the buffer, whose outputs are *activate* events. These events are hidden (\). The visible interactions of the system are the events of the robotic platform only.

The semantics of a controller is the parallel composition of the semantics of its main state machines, not including those that define operations, again, interacting according to their connections. This semantics is similar to that of a module, except that the components are the processes that model state machines.

A state machine specifies a sequential control flow hierarchically. Its semantics is the parallel composition of a process *States*, which models each of the states, with a process *InitialTransitions*, which describes the transition to the initial state. Below, we show the semantics of DetectAndFlag.

$$
\begin{aligned}
&\mathit{DetectAndFlag} = \\
&\quad (\mathit{InitialTransitions} \,[\![\, \mathit{EnterExitChannels} \,]\!]\, \mathit{States}) \setminus (\Sigma \setminus \mathit{Events}) \\
&\mathit{InitialTransitions} = \\
&\quad \mathit{enter.DetectAndFlag.Detect} \rightarrow \mathit{entered.DetectAndFlag.Detect} \rightarrow \mathit{SKIP}
\end{aligned}
$$

The CSP channels used to model the control flow of a state activation and deactivation are *enter*, *entered*, *exit* and *existed*. Events using these channels model the beginning and end of the actions of entering and exiting a state. Each of them takes two parameters: the state that requested the action to start and the target of the request. For instance, in *InitialTransitions*, the state machine itself requests the activation of the state *Detect*.

A process like *Detect* or *Flag* that models a state s does so in a compositional way, capturing only information about s itself, irrespective of the context (parent state or state machine) where it occurs. Such a process first offers events *enter* to request and then *entered* to acknowledge entry to s, and then offers a choice of events that trigger its transitions, including those of its substates, and any other transitions that might be available for its parents (which, if taken, also lead to an exit of s). When any of the transition events is chosen, then the *exit* and *exited* events to request and acknowledge exit are offered.

We note that the state is not aware of the transitions of its parents and, therefore, accepts any of them. In the definition of the process for a parent state or for the state machine, the available choices are further restricted. Only the transitions associated with the ancestor states are feasible.

At the top level, *States* composes in parallel restricted versions of processes for the top states; in our example, versions *DetectR* and *FlagR* of *Detect* and *Flag*. Because there are no parent states, the transition events enabled are those for their own transitions (that trigged by f, the final transition, and that with condition reached for *Detect*, and just the silent transition for *Flag*).

$$
\begin{array}{l}
States = \\
\quad DetectR \ [|\ \{|enter.x.y, entered.x.y, exit.x.y, exited.x.y \mid x, y : \{Detect, Flag\}|\}\ |] \\
FlagR \\
\qquad \setminus \{|enter.x.y, exit.x.y, exited.x.y \mid x, y : \{Detect, Flag\}|\}
\end{array}
$$

The processes synchronise on their common activation and deactivation events. For instance, the event *enter.Detect.Flag* is in the synchronisation set, indicating that *Detect* may request the activation of *Flag*, but *enter.DetectAndFlag.Flag* is not in the synchronisation set, because it involves the state machine and *Flag*, but not *Detect*. In other words, synchronisation establishes the flow of activation and deactivation of the states at the same level.

In the parallelism that defines a state machine, like *DetectAndFlag* above, the possibility of entering a state different from the initial state is blocked. To recall, *States* offers the possibility of entering (Detect or Flag), and later exiting the states. All these events are included in the set *EnterExitChannels*. So, *InitialTransitions* defines exactly which state is entered. The machine terminates if it reaches a final state, but it does not require that state to exit. So, an event like *exit.SM.S*, where *SM* is the state-machine identifier is not possible. Instead, if a machine terminates, a special event *terminated* is raised.

Finally, we hide all CSP events (set Σ), except those in the set *Events*, which represent events in the RoboChart model itself. These are the only events visible in the semantics, although, for verification, we may hide fewer events.

The CSP semantic models just described can be automatically generated for a RoboChart model using our RoboTool described in Section **??**. The complete model for our example can be found in Section 4.1.3

4.1.2 Verification

The automatically generated semantics outlined above, can be used for analysis and verification using the FDR model-checker. We can, for example, establish determinism, and absence

of divergence and deadlock. In our example, the DetectAndFlag state machine is very simple. We are encouraged, however, because, although to give a compositional semantics, we use a lot of extra events and parallelism, the use of the compression functions *diamond* and *sbisim* (that preserve the semantics of the processes) radically optimises the analysis reducing the number of states from thousands to just four as expected.

A second example of the kind analysis we can perform is the verification of assumptions about the hardware controlled by the state machine. For instance, the rover can carry a limited amount of flags. We have modelled this mechanism as a CSP process and used a refinement to verify whether the controller satisfies this restriction of the hardware. As probably already observed, it does not, and must be refined. In such a verification, FDR produces a counterexample that pinpoints a scenario that break this assumption. This information can be used to guide the redesign of the state machine.

As a final example, we can perform reachability analysis using FDR. For each state s of a state machine, we can define a process by renaming events of the form $entered.x.s$ to $hasEntered.s$, and hiding all events except $hasEntered.s$. We can then verify that this process contains the trace $\langle hasEntered.s \rangle$, and so s is entered at least once. We have applied this technique to a modified version of our example where the variable *reached* is not updated. In this case, the verification correctly points out that the state *Flag* is unreachable.

No doubt, further evaluation is necessary to gauge the effectiveness of model checking RoboChart semantic models. We are, however, encouraged by the results with compression, which are related to the structure of the models, and have the possibility of exploring theorem proving using the UTP theory for CSP.

4.1.3 Complete Semantics of *DetectAndFlag*

In this section, we present and explain the complete model of the state-machine *DetectAndFlag* shown in Figure 2.3. As briefly described in Section 4.1 this model is a denotational semantics of RoboChart state-machines specified in CSP where parallelism is used mainly to conjoin requirements. The behaviour of each state is specified as a CSP process, and these processes are composed in parallel to formalise the behaviours of composite state and the state-machine itself.

```
nat = {1,2,3}

nametype ID = Seq(Char)
```

```
FINAL = "_final_"

channel terminate

channel enter, entered: ID.ID
channel exit, exited: ID.ID

transparent chase
external prioritise
transparent diamond
transparent sbisim

sbdia(P) = sbisim(diamond(P))

channel not_deadlocked

channel internal__: ID

nametype Vector = (nat,nat)
nametype boolean = {True,False}

channel event_f: ID.Vector
channel f: Vector

InterruptsBut(ids) = {| internal__.x, event_f.x | x <- diff(TIDS,ids) |}
Interrupts(id) = {| internal__.id, event_f.id |}
InterruptsOf(ids) = {| internal__.x, event_f.x | x <- ids |}

nametype IDS = {"DetectAndFlag", "Detect", "Searching", "Approach", "Flag"}
nametype TIDS = {"t1", "t2", "t3", "t4"}

t1 = "t1"
t2 = "t2"
t3 = "t3"
t4 = "t4"
```

```
channel set_position, get_position: Vector
channel set_reached, get_reached: boolean

channel cmove, walk, drop

DropFlag = drop -> SKIP
RandomWalk = walk -> SKIP
move(x,y) = cmove -> SKIP

Flag = enter?x:diff(IDS,{"Flag"})!"Flag" -> (
    (
      (DropFlag; entered!x!"Flag" -> SKIP);
      (STOP/\(
        internal__.t1 -> enter!"Flag"!"Detect" -> entered!"Flag"!"Detect" -> S
        [] ([] e: InterruptsBut({t1}) @ e -> exit?x:diff(IDS,{"Flag"})!"Flag"
                    (exited!x!"Flag" -> SKIP))
      ))
    )
  ); Flag

Searching = enter?x:diff(IDS,{"Searching"})!"Searching" -> (
    (
      (entered!x!"Searching" -> SKIP);
      (RandomWalk;STOP/\(
        event_f.t2?position -> set_position!position ->
                  enter!"Searching"!"Approach" ->
                  entered!"Searching"!"Approach" -> SKIP
        [] ([] e: InterruptsBut({t2}) @ e ->
                  exit?x:diff(IDS,{"Searching"})!"Searching" ->
                  (exited!x!"Searching" -> SKIP))
      ))
    )
  ); Searching

Approach = enter?x:diff(IDS,{"Approach"})!"Approach" -> (
```

```
    (
      (get_position?p -> move(p,5); entered!x!"Approach" -> SKIP);
      (STOP/\(
        internal__.t3 -> set_reached!True -> Final
        [] ([] e: InterruptsBut({t3}) @ e ->
                        exit?x:diff(IDS,{"Approach"})!"Approach" ->
                            (exited!x!"Approach" -> SKIP))
      ))
    )
  ); Approach

Final = ([] e: InterruptsBut({t3}) @ e ->
  exit?x:diff(IDS,{FINAL})!FINAL ->
  (exited!x!FINAL -> SKIP))

DetectAux = enter?x:diff(IDS,{"Detect"})!"Detect" -> (
    set_reached!False -> enter!"Detect"!"Searching" ->
          entered!"Detect"!"Searching" -> entered!x!"Detect" -> SKIP
  );
  (STOP/\(
    internal__.t4 -> exit!"Detect"?y:{"Searching","Approach", FINAL} ->
          exited!"Detect"!y -> enter!"Detect"!"Flag" ->
          entered!"Detect"!"Flag" -> SKIP
    []
    ([] e: InterruptsBut({t2,t3,t4}) @ e ->
          exit?x:diff(IDS,{"Detect"})!"Detect" ->
          exit!"Detect"?y:{"Searching","Approach", FINAL} ->
          exited!"Detect"!y -> exited!x!"Detect" -> SKIP)
  )); DetectAux

SearchingR = Searching
  [|diff(union(Interrupts(t2),Interrupts(t3)),{|event_f.t2|})|]
  SKIP

ApproachR = Approach
  [|diff(union(Interrupts(t2),Interrupts(t3)),{|internal__.t3|}|]
```

```
  SKIP

DetectSubStates = (
  SearchingR
  [|{|enter.x.y, entered.x.y, exit.x.y, exited.x.y |
        x <- {"Searching","Approach",FINAL},
                y <- {"Searching","Approach",FINAL}|}|]
  ApproachR)
)\{|enter.x.y, exit.x.y, exited.x.y |
        x <- {"Searching","Approach"},
                y <- {"Searching","Approach"}|}

Detect =
(
  DetectAux
  [|
    union(diff(InterruptsBut({}),{|event_f.t2 ,internal__.t3|}),
            {|enter.y.x, entered.y.x, exit.y.x, exited.y.x |
            x <- {"Searching","Approach",FINAL},
                        y <- diff(IDS,{"Searching","Approach",FINAL})|})
  |]
  DetectSubStates
)\{|enter."Detect".x, exit."Detect".x, exited."Detect".x |
      x <- {"Searching","Approach",FINAL}|}


Machine = enter."DetectAndFlag"."Detect" -> entered."DetectAndFlag"."Detect" -
      [|
      {|enter.x.y,entered.x.y,exit.x.y,exited.x.y|
              x<-diff(IDS,{"Detect","Flag"}),
                  y<-{"Detect","Flag"}|}
    |]
    MachineSubStates

DetectR = Detect
  [|diff(InterruptsOf({t1,t2,t3,t4}),
```

```
            {|internal__.t4, event_f.t2, internal__.t3|})|]
  SKIP

FlagR = Flag
  [|diff(InterruptsOf({t1,t2,t3,t4}),{|internal__.t1|})|]
  SKIP

MachineSubStates = (
      DetectR
      [|{|enter.x.y, entered.x.y, exit.x.y, exited.x.y |
            x <- {"Detect","Flag"},
                y <- {"Detect","Flag"}|}|]
      FlagR
    )\{|enter.x.y, exit.x.y, exited.x.y |
             x <- {"Detect","Flag"}, y <- {"Detect","Flag"}|}


DetectAndFlagForReachability =
(((
  Machine\{|enter.x.y,exit.x.y,exited.x.y|x<-IDS,y<-{"Detect","Flag"}|}
  [|{|get_position,get_reached,set_position,set_reached,internal__.t4|}|]
  Memory((1,1),false)
)[[event_f.x <- f | x <- TIDS]])
\{|get_position,get_reached,set_position,set_reached,internal__.x|x<-TIDS|})

DetectAndFlag = DetectAndFlagForReachability\{|entered.x.y|x<-IDS,y<-IDS|}

Memory(position,reached) =
  get_position!position -> Memory(position,reached)
  []
  get_reached!reached -> Memory(position,reached)
  []
  set_position?x -> Memory(x,reached)
  []
  set_reached?x -> Memory(position,x)
  []
```

```
(reached)&internal__.t4 -> Memory(position,reached)
```

The properties discussed in Section 4.1.2 can be verified using the following assertions.

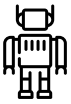
The limitation of the DropFlag mechanism that only two flags can be dropped during the execution of the robots is specified by the abstract models *AbstractSpecification*1 and *AbstractSpecification*2. These processes model the behaviour where any events can happen any number of times, except for the event *drop* that can only happen twice; the first process is used to verify the property under the traces model, whilst the second is used for verification in the failures-divergences model.

```
assert sbdia(DetectAndFlag) :[deterministic [FD]]
assert sbdia(DetectAndFlag) :[deadlock-free [FD]]
assert sbdia(DetectAndFlag) :[divergence-free [FD]]

Trace = walk -> f.(2,2) -> cmove -> drop -> Trace

AbstractSpecification1 =
Run({|f,cmove,walk|});
drop -> Run({|f,cmove,walk|}); drop -> Run({|f,cmove,walk|})
AbstractSpecification2 =
Chaos({|f,cmove,walk|});
drop -> Chaos({|f,cmove,walk|}); drop -> Chaos({|f,cmove,walk|})

Run(events) = [] e: events @ e -> Run(events) [] SKIP
Chaos(events)= |~| e: events @ e -> Chaos(events) |~| SKIP

assert sbdia(AbstractSpecification1) [T= sbdia(DetectAndFlag)
assert sbdia(AbstractSpecification2) [FD= sbdia(DetectAndFlag)

channel hasEntered: IDS

DetectAndFlagReachability(s) =
(sbdia(DetectAndFlagForReachability)[[entered.x.y<-hasEntered.y|x<-IDS,y<-IDS
\{|f,walk,cmove,drop,hasEntered.x|x<-diff(IDS,{s})|}

assert DetectAndFlagReachability("Approach") :[has trace]: <hasEntered."Appro
```

```
assert DetectAndFlagReachability("Searching") :[has trace]: <hasEntered."Searc
assert DetectAndFlagReachability("Detect") :[has trace]: <hasEntered."Detect">
assert DetectAndFlagReachability("Flag") :[has trace]: <hasEntered."Flag">

assert DetectAndFlag :[deterministic [FD]]
```

4.2 Detailed Semantics: Core Language

4.2.1 Robotic Platforms



At this stage, a robotic platform has not special meaning; it only contributes to the validation of modules.

Interfaces

Similarly to robotic platforms, interfaces are used to constraints the valid specifications and guarantee definedness of variables, operations and events.

VariableList

VariableLists and Variables contribute to the declaration of channels and the construction of the memory process. This is necessary because variables can be shared between parallel components (e.g., controllers and state-machines), but also between sequential components modelled in parallel (e.g., states).

```
val variables = new LinkedList<Variable>()
val transitions = if (stm instanceof StateMachineBody)
(stm as StateMachineBody).allTransitions
else (new LinkedList<Transition>)
if (stm instanceof OperationDef)
variables.addAll(stm.parameters)
variables.addAll(stm.allVariables)
'''
Memory_«stm.id»«stm.memoryParameterisation» =
«IF stm.allVariables.size == 0 && transitions.size == 0»  []«ELSE»
```

```
(
«IF stm.allVariables.size > 0»
«FOR v: stm.allVariables SEPARATOR '\n  []'»
«getter_setter(v,stm.allVariables,stm)»
«ENDFOR»
«ENDIF»
«IF stm.allVariables.size > 0 && transitions.size > 0»  []«ENDIF»
«FOR t: transitions SEPARATOR '\n  []'»
«t.memory_transition(stm)»
«ENDFOR»
)
«ENDIF»
'''
```

4.2.2 State-Machines

The semantics of state-machines is given by the method *compile* on values of type *ModuleNode*. Whist this method could in principle be applied to controllers as well, it would return an empty string. The semantics of controllers is given by a method *compile* that targets controllers directly.

```
def dispatch compile(ModuleNode node) {
val stm = if (node instanceof StateMachineBody)
  (node as StateMachineBody) else null
if (stm == null) return ""
'''
channel «id(stm)»_internal__: «id(stm)»_TIDS

channel «id(stm)»_enter, «id(stm)»_entered: «id(stm)»_SIDS.«id(stm)»_SIDS
channel «id(stm)»_exit,«id(stm)»_exited: «id(stm)»_SIDS.«id(stm)»_SIDS

«FOR v: stm.allVariables»
channel get_«v.id», set_«v.id»: «v.type.compile»
«ENDFOR»

-- declaring identifiers of state and final states
```

```
«id(stm)»_SIDS = {
"«id(stm)»"«IF stm.eAllContents.
filter[o|o instanceof State || o instanceof Final].size > 0»,«ENDIF»
«FOR s : stm.eAllContents.
  filter[o|o instanceof State || o instanceof Final].toIterable SEPARATOR ','»
"«id(s)»"
«ENDFOR»
}

-- declaring identifiers of transitions
«id(stm)»_TIDS = {
"__NULLTRANSITION__"«FOR s : stm.allTransitions»,
"«id(s)»"«ENDFOR»
}

-- declaring state machine events
«FOR e : getEvents(node)»
  channel «id(e)»__:«id(stm)»_TIDS
    «IF e.type != null && e.type instanceof TypeRef».«e.type.compile»«ENDIF»
  channel «id(e)»
    «IF e.type != null && e.type instanceof TypeRef»:«e.type.compile»«ENDIF»
«ENDFOR»

-- declaring all states
«FOR s : stm.eAllContents.filter(State).toIterable»
«IF s.eContainer instanceof NodeContainer»
«s.substatesCS(s.eContainer as NodeContainer)»
«ENDIF»

«IF (s as NodeContainer).nodes.size > 0»
«(s as NodeContainer).internalInterrupts»

«(s as NodeContainer).internalTriggersOf»

«(s as NodeContainer).compileComposite»
«ENDIF»
```

```
«s.triggersOf»
«s.compile»
«ENDFOR»

«FOR i : stm.eAllContents.filter(Initial).toIterable»
«i.compile»
«ENDFOR»

«FOR t : stm.eAllContents.filter[t|
t instanceof Transition && (
(t as Transition).source instanceof State ||
(t as Transition).source instanceof Initial
)
].toIterable»
«(t as Transition).compile»
«ENDFOR»

«stm.internalInterrupts»
«stm.compileComposite»
-- memory process
«stm.memory»
-- main process
«stm.id»«node.parameterisation» = (STM_«stm.id»«node.parameterisation»
[|union(
«get_set_channels(stm)»,
«event_channels(stm)»
)|]
Memory_«stm.id»«node.memoryInstantiation»
)«(stm as StateMachineBody).stateMachineRenaming»\«get_set_channels(stm)»
'''
}
```

States

The semantics of states distinguishes between simple and composite states. In the case of composite states, additional semantic functions are used to compose the substates and the container state.

```
def dispatch compile(State state)
'''
«IF state.nodes.size == 0»S_«id(state)»
«ELSE»S_«id(state)»_aux
«ENDIF»
«state.moduleNode.parameterisation» = «enterState(state)»;
(«state.duringAction»; STOP /\ (
«FOR t : state.transitionsFrom SEPARATOR '[]'»
T_«id(t)»«state.moduleNode.parameterisation»
«ENDFOR»
«IF state.transitionsFrom.size > 0»[]«ENDIF»
«id(state.moduleNode)»_internal__?x:
  diff(«id(state.moduleNode)»_TIDS,«getTransitionIDS(state)») ->
    «exitState(state)»
«FOR e : getEvents(state.moduleNode)»
[] «id(e)»__?x:diff(«id(state.moduleNode)»_TIDS,«getTransitionIDS(state)»)
«IF e.type != null»?x__«ENDIF» -> «exitState(state)»
«ENDFOR»
));
«IF state.nodes.size == 0»S_«id(state)»
«ELSE»S_«id(state)»_aux
«ENDIF»
«state.moduleNode.parameterisation»
'''

def dispatch compileComposite(State node)
'''
«FOR s : node.nodes.filter(State)»
S_«id(s)»_R«s.moduleNode.parameterisation» =
  S_«id(s)»«s.moduleNode.parameterisation»
  [|diff(«id(node)»_int_int,«s.id»_triggers)|]
  SKIP
```

```
«ENDFOR»
S_«id(node)»«node.moduleNode.parameterisation» =
  (S_«id(node)»_aux«node.moduleNode.parameterisation»
  [|union(diff(«node.moduleNode.id»_int_int, «node.id»_int_triggers),
    {|«id(node.moduleNode)»_enter.x.y,
      «id(node.moduleNode)»_entered.x.y,
      «id(node.moduleNode)»_exit.x.y,
      «id(node.moduleNode)»_exited.x.y |
      x <- diff(«id(node.moduleNode)»_SIDS,«node.substatesNoFinal»),
      y <- «node.substates»
    |})|]
  «compileSubstates(node.nodes.filter(State))»
)\{|«id(node.moduleNode)»_enter.x.y,
    «id(node.moduleNode)»_entered.x.y,
    «id(node.moduleNode)»_exit.x.y,
    «id(node.moduleNode)»_exited.x.y |
    x <- diff(«id(node.moduleNode)»_SIDS,«node.substates»),
    y <- «node.substates»
|}
'''

def dispatch compileTarget(State tgt, NodeContainer origin) {
'''
«id(origin.moduleNode)»_enter!"«id(origin)»"!"«id(tgt)»" ->
 «id(origin.moduleNode)»_entered!"«id(origin)»"!"«id(tgt)»" -> SKIP
'''
}
```

Initial nodes

Initial nodes are part of a state or state-machine. The semantic function produces an action which is used as part of the semantics of its container.

```
def dispatch compile(Initial i) {
val parent = i.eContainer
```

```
'''I_«id(i)»«i.moduleNode.parameterisation» =
«IF i.transitionsFrom.size == 0»SKIP
«ELSE»T_«i.transitionsFrom.get(0).id»«i.moduleNode.parameterisation»
«ENDIF»'''
}
```

Junction node

Since the semantics of transitions cover the whole path from an initial node or regular state to a final or regular state, junction nodes are only trated as targets of transitions.

```
def dispatch compileTarget(Junction j, NodeContainer origin) {
if (j.transitionsFrom.size == 0) {
'''SKIP'''
} else {
val vars = new HashSet<Variable>()
for (tt : j.transitionsFrom) {
vars.addAll(tt.condition.usedV)
}
val choice = '''
«FOR t : j.transitionsFrom SEPARATOR '[]'»
(«t.compileTransition(origin)»)
«ENDFOR»
'''
readState(vars, choice)
}
}
```

Final states

Final states are also only treated as targets of transitions.

```
def dispatch compileTarget(Final f, NodeContainer origin) {
'''«id(origin.moduleNode)»_exit?x!"«id(f)»" ->
   «id(origin.moduleNode)»_exited?x!"«id(f)»" -> SKIP'''
}
```

Transitions

→ The semantics of transitions is given by *Circus* actions, which are used by the source states (and initial nodes) and ultimately includes all the transitions in a path.

```
def compileTransition(Transition t, NodeContainer origin) {
val src = t.source
val tgt = t.target
if (src instanceof State) {
val e = t.trigger?.event
var trigger = ""
if (e != null) {
val stm = src.moduleNode
trigger = '''«id(e)»__!"«id(t)»"«IF e.type != null»?x__«ENDIF»'''
} else {
val stm = src.moduleNode
trigger = '''«id(stm)»_internal__!"«id(t)»"'''
}
return '''«trigger» ->
«IF src.nodes.size > 0»«exitSubStates(src)»;«ENDIF»
«IF t.action != null»«t.action.compile»;«ENDIF»«tgt.compileTarget(src)»'''
} else if (src instanceof Junction) {
'''«IF t.condition != null»(«t.condition.compile»)&«ENDIF»
«IF t.action != null»«t.action.compile»;«ENDIF»
«tgt.compileTarget(origin)»'''
} else if (src instanceof Initial) {
var NodeContainer parent = null
if (src.eContainer instanceof State)
parent = (src.eContainer as State)
else if (src.eContainer instanceof StateMachineDef)
parent = src.eContainer as StateMachineDef
else if (src.eContainer instanceof OperationDef)
parent = src.eContainer as OperationDef
else
throw new Exception("An initial node is not...")
'''«IF t.action != null»«t.action.compile»;«ENDIF»
```

```
  «tgt.compileTarget(parent)»'''
}
}
```

4.2.3 Controllers

 The semantics of a controller is the parallel composition of its state-machines with channels appropriately synchronised and renamed according to the controller connections.

```
def compile(ControllerDef c)
'''
-- declaring controller events
«FOR e: c.allEvents»
channel «id(e)»
«IF e.type != null && e.type instanceof TypeRef»:«e.type.compile»«ENDIF»
«ENDFOR»

«c.id» = «c.composeStateMachines(c.machines,c.connections)»
'''

def composeStateMachines(ControllerDef ctrl,
  List<StateMachine> stms, List<CtrlConnection> connections) {
if (stms.size == 0) {
return "SKIP"
} else if (stms.size == 1) {
var stm = stms.get(0)
var s = if (stm instanceof StateMachineRef) (stm as StateMachineRef).ref.id
        else stm.id
var open = false
for (c: connections) {
var ModuleNode other = null
var Event eorigin = null
var Event etarget = null
if (c.from == stm) {
other = c.to
eorigin = c.efrom
```

```
etarget = c.eto

} else if (c.to == stm) {
other = c.from
eorigin = c.eto
etarget = c.efrom
}
if (other instanceof ControllerDef && eorigin != null && etarget != null) {
if (!open) {
s = '''«s»[[«eorigin.id» <- «etarget.id»'''
open = true
}
else
s = '''«s»,
«eorigin.id» <- «etarget.id»'''
}
}
if (open)
s = '''«s»]]'''
return s
} else {
val head = stms.head
val tail = stms.tail
var s = if (head instanceof StateMachineRef) (head as StateMachineRef).ref.id
        else head.id
var chanset = new ArrayList<Event>()
var open = false
for (c: connections) {
var ModuleNode other = null
var Event eorigin = null
var Event etarget = null
if (c.from == head) {
other = c.to
eorigin = c.efrom
etarget = c.eto
```

```
} else if (c.to == head) {
other = c.from
eorigin = c.eto
etarget = c.efrom
}
if (other != null && eorigin != null && etarget != null) {
if (!open) {
if (!(other instanceof ControllerDef))
chanset.add(etarget)
s = '''«s»[[«eorigin.id» <- «etarget.id»'''
open = true
}
else {
if (!(other instanceof ControllerDef))
chanset.add(etarget)
s = '''«s»,«eorigin.id» <- «etarget.id»'''
}
}
}
if (open) {
s = '''«s»]]'''
}

s = '''
(«s»
[|{|«FOR c: chanset»«c.id»«ENDFOR»|}|]
«ctrl.composeStateMachines(tail.toList,connections)») \ diff(
  {|«FOR c: chanset»«c.id»«ENDFOR»|},
  {|«FOR e: ctrl.allEvents SEPARATOR ','»«e.id»«ENDFOR»|}
)
'''

return s
}
}
```

Controller Connection

→ Controller connections are treated as part of the semantics of a controller. They correspond to channel renamings.

4.2.4 Modules

```
def compile(Module m) {
val aux = m.nodes.filter(RoboticPlatform).get(0)
val rp = if (aux instanceof RoboticPlatformRef) aux.ref else aux as RoboticPlatformDef
'''
-- declaring robotic platform events
«FOR e: rp.allEvents»
channel «id(e)»«IF e.type != null && e.type instanceof TypeRef»:«e.type.compile»«ENDIF»
«ENDFOR»

«m.id» = «rp.composeControllers(m.nodes.filter(Controller).toList,m.connections)»
'''
}

def composeControllers(RoboticPlatform rp, List<Controller> ctrls, List<ModuleConnection>
if (ctrls.size == 0) {
return "SKIP"
} else if (ctrls.size == 1) {
var ctrl = ctrls.get(0)
var s = if (ctrl instanceof ControllerRef) (ctrl as ControllerRef).ref.id
        else ctrl.id
var open = false
for (c: connections) {
var ModuleNode other = null
var Event eorigin = null
var Event etarget = null
if (c.from == ctrl) {
other = c.to
eorigin = c.efrom
etarget = c.eto
```

```
} else if (c.to == ctrl) {
other = c.from
eorigin = c.eto
etarget = c.efrom
}
if (other instanceof RoboticPlatform && eorigin != null && etarget != null) {
if (!open) {
s = '''«s»[[«eorigin.id» <- «etarget.id»'''
open = true
}
else
s = '''«s»,
«eorigin.id» <- «etarget.id»'''
}
}
if (open)
s = '''«s»]]'''
return s
} else {
val head = ctrls.head
val tail = ctrls.tail
var s = if (head instanceof ControllerRef) (head as ControllerRef).ref.id
        else head.id
var chanset = new ArrayList<Event>()
var open = false
for (c: connections) {
var ModuleNode other = null
var Event eorigin = null
var Event etarget = null
if (c.from == head) {
other = c.to
eorigin = c.efrom
etarget = c.eto

} else if (c.to == head) {
```

```
other = c.from
eorigin = c.eto
etarget = c.efrom
}
if (other != null && eorigin != null && etarget != null) {
if (!open) {
if (!(other instanceof RoboticPlatform)) {
chanset.add(etarget)
}
s = '''«s»[[«eorigin.id» <- «etarget.id»'''
open = true
}
else {
if (!(other instanceof RoboticPlatform)) {
chanset.add(etarget)
}
s = '''«s»,«eorigin.id» <- «etarget.id»'''
}
}
}
if (open) {
s = '''«s»]]'''
}

val plat = (if (rp instanceof RoboticPlatformRef) rp.ref
            else rp as RoboticPlatformDef)

s = '''
(«s»
[|{|«FOR c: chanset»«c.id»«ENDFOR»|}|]
«rp.composeControllers(tail.toList,connections)») \ diff(
  {|«FOR c: chanset»«c.id»«ENDFOR»|},
  {|«FOR e: plat.allEvents SEPARATOR ','»«e.id»«ENDFOR»|}
)
'''
```

```
return s
}
}
```

Module Connection

→ Module connections are treated as part of the semantics of a module. They correspond to channel renamings.

4.2.5 Statements

```
def dispatch CharSequence compile(Statement s) {
if (s instanceof Assignment) {
val l = s.left
if (l instanceof VarRef) {
val assign = '''set_«l.name.id»!«s.right.compile» -> SKIP'''
val readandassign = readState(s.right.usedV,assign)
return readandassign
} else {
// Not dealing with datatypes, so no varselection, but we could
//encode datatypes in CSP using regular parameters and the name
//of the fields prefixed by the name of the variable
return "SKIP"
}
} else if (s instanceof Skip) {
return '''SKIP'''
} else if (s instanceof Call) {
val op = s.member.definition
val args = s.args
var variables = new HashSet<Variable>
for (e: args) {
variables.addAll(e.usedV)
}
val call = '''OP_«op.id»«IF args.size > 0»(
```

```
  «FOR a:args SEPARATOR ','»«a.compile»«ENDFOR»
)«ENDIF»'''
val readandcall = readState(variables,call)
return readandcall
} else if (s instanceof SeqStatement) {
var iterator = s.statements.iterator
if (iterator.hasNext) {
var ss = iterator.next.compile
while (iterator.hasNext) {
ss = ss+";"+iterator.next.compile
}
return ss
}
} else if (s instanceof SendEvent) {
val e = s.event
'''«e.id» -> SKIP'''
} else
'''SKIP'''
}
```

4.2.6 Expressions

```
def dispatch CharSequence compile(Expression e) {
  if (e instanceof BooleanExp) {
  e.value
  } else if (e instanceof IntegerExp) {
  e.value.toString
  } else if (e instanceof RefExp) {
  var n = e.value
  if (n instanceof Variable) {
  n.name
  } else if (n instanceof Constant) {
  n.name
  } else {
  throw new Exception("Fields have not yet been implemented")
  }
```

```
} else if (e instanceof GreaterThan) {
'''(«e.left.compile»>«e.right.compile»)'''
} else if (e instanceof GreaterOrEqual) {
'''(«e.left.compile»>=«e.right.compile»)'''
} else if (e instanceof LessThan) {
'''(«e.left.compile»<«e.right.compile»)'''
} else if (e instanceof LessOrEqual) {
'''(«e.left.compile»<=«e.right.compile»)'''
} else if (e instanceof Equals) {
'''(«e.left.compile»==«e.right.compile»)'''
} else if (e instanceof Different) {
'''(«e.left.compile»!=«e.right.compile»)'''
} else if (e instanceof And) {
'''(«e.left.compile» and «e.right.compile»)'''
} else if (e instanceof Or) {
'''(«e.left.compile» or «e.right.compile»)'''
} else if (e instanceof Implies) {
'''(not «e.left.compile» or «e.right.compile»)'''
} else if (e instanceof Iff) {
'''((not «e.left.compile» or «e.right.compile»)
   and
   («e.left.compile» or not «e.right.compile»))'''
} else if (e instanceof Not) {
'''(not «e.exp.compile»'''
} else if (e instanceof Plus) {
'''(«e.left.compile» + «e.right.compile»)'''
} else if (e instanceof Minus) {
'''(«e.left.compile» - «e.right.compile»)'''
} else if (e instanceof Mult) {
'''(«e.left.compile» * «e.right.compile»)'''
} else if (e instanceof Div) {
'''(«e.left.compile» / «e.right.compile»)'''
} else if (e instanceof Modulus) {
'''(«e.left.compile» % «e.right.compile»)'''
} else if (e instanceof Neg) {
'''(-«e.exp.compile»)'''
```

```
  } else if (e instanceof Cat) {
  '''(«e.left.compile» ^ «e.right.compile»)'''
  } else if (e instanceof ParExp) {
  '''(«e.exp.compile»)'''
  }
}
```

4.2.7 Type Declaration

```
def dispatch compile(TypeDecl t) {
if (t.name.equals("int"))
'''nametype int = {-«N»..«N»}'''
else if (t.name.equals("nat"))
'''nametype nat = {0..«N»}'''
else if (t.name.equals("string"))
'''nametype string = Seq(Char)'''
else if (t.name.equals("boolean"))
'''nametype boolean = Bool'''
else
'''nametype «t.name» = {1}'''
}
```

Primitive Types

The semantics of primitive types has not yet been defined.

Datatypes

The semantics of datatypes has not yet been defined.

The semantics of fields has not yet been defined.

Enumeration

The semantics of enumerations has not yet been defined.

4.2.8 Type Constructors

```
def dispatch CharSequence compile(Type t) {
if (t instanceof ProductType) {
var CharSequence aux = null
val iter = t.types.iterator
if (iter.hasNext) {
val type = iter.next
aux = type.compile
}
while (iter.hasNext) {
val type = iter.next
aux = aux+'.'+type.compile
}
return aux
} else if (t instanceof SetType) {
return '''Set(«t.domain.compile»)'''
} else if (t instanceof SeqType) {
return '''Seq(«t.domain.compile»)'''
} else if (t instanceof TypeRef)
t.ref.name
else
"Object"
}
```

4.3 Detailed Semantics: Timed Language

Clock

The semantics of clocks has not yet been defined.

Timed Statements

The semantics of timed statements has not yet been defined.

Timed Expressions

The semantics of timed expressions has not yet been defined.

Timed Triggers

The semantics of timed triggers has not yet been defined.

4.4 Detailed Semantics: Probabilistic Language

4.4.1 Probabilistic Junction

The semantics of probabilistic junctions has not yet been defined.

4.4.2 Probabilistic Transition

The semantics of probabilistic transitions has not yet been defined.

4.5 Detailed Semantics: Hybrid Language

4.6 Tool Support

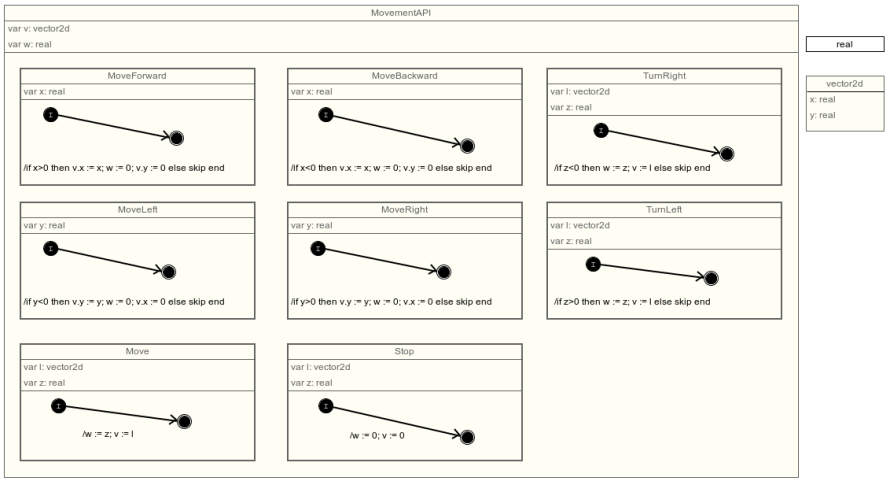


Figure 5.1: An example of API defining the movement of a robot moving in a 2D environment.

CHAPTER 5

API

In this chapter, we describe the API in more details. The purpose of the API is to provide a library of generic operations that are useful across a variety of robotic applications and platforms. Some operations are primitive, while others are composite. The composite operations can be implemented by a set of primitive operations. The API is organized by type of robots (e.g. wheeled robots, flying robots, and so on) and equipment (e.g. camera, infrared sensors). The operations also accept inputs such as the speed of the robot.

For example, in the operation `MoveForward`, the linear speed of the robot is set if it is positive and the angular speed will be set to `0`. Other operations related with the movement of a 2D robot are shown in Figure 5.1.

Some operations such as `MoveUp`, `MoveDown`, `Roll/Pitch` are included in the API of flying robots. For humanoid robots, the specific operations include `Lean`, `StandUp` and `FallDown`.

For a composite operation such as `ObstacleAvoidance`, the robot needs to decide how to move depending on the position of the obstacle. `ObstacleAvoidance` can be realized using different sensors such as camera or infrared sensors. Note that although we define the operations in the API in an abstract way, the implementation of a particular operation is related with one robotic platform.

The API can be extended in the tool.

Chapter 6

Simulation

Chapter 7

Examples

7.1 Core Language

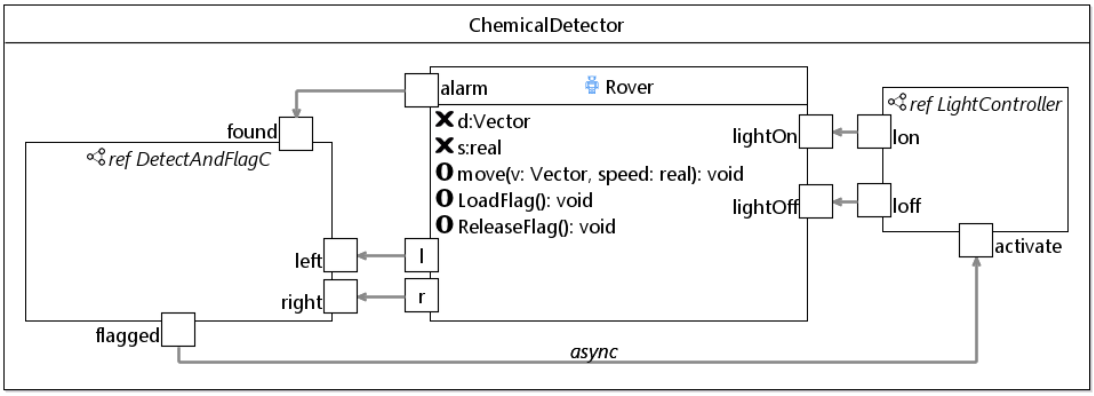


Figure 7.1: Module of the simplified chemical detector

7.1.1 Chemical Detector – Simplified

This example is a simplified version of the chemical detector presented in [?]. It abstracts away the particular detection algorithm that is the focus of the paper and focus on the other aspect such as architecture, locomotion, battery monitoring etc.

7.1.2 Chemical Detector – Complete

This example is an accurate model of the chemical detector presented in [?].

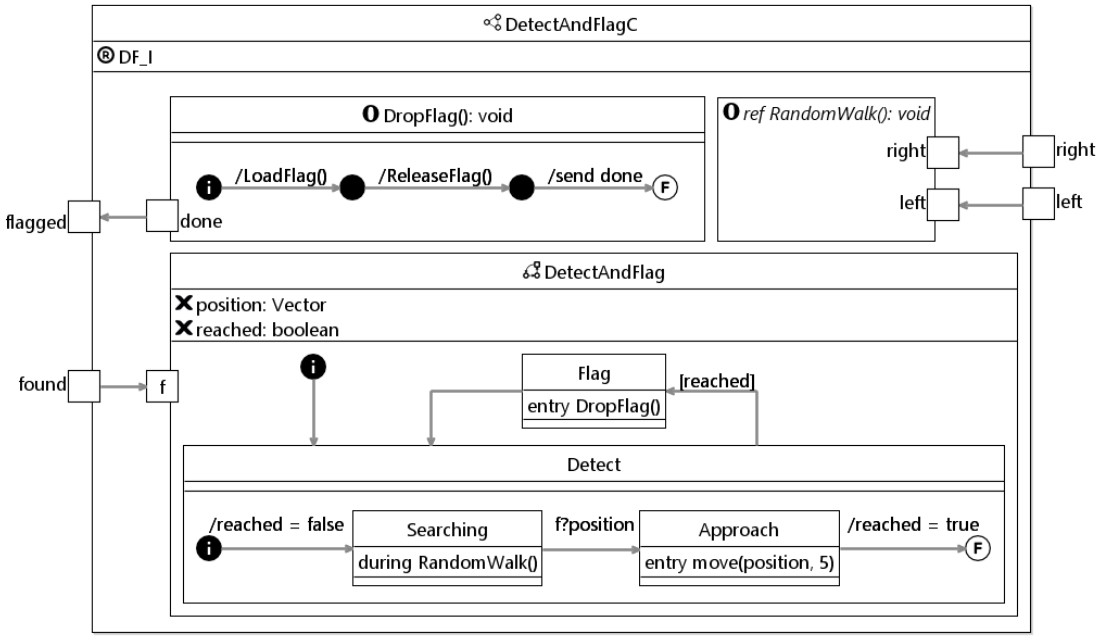


Figure 7.2: Detect and Flag Controller

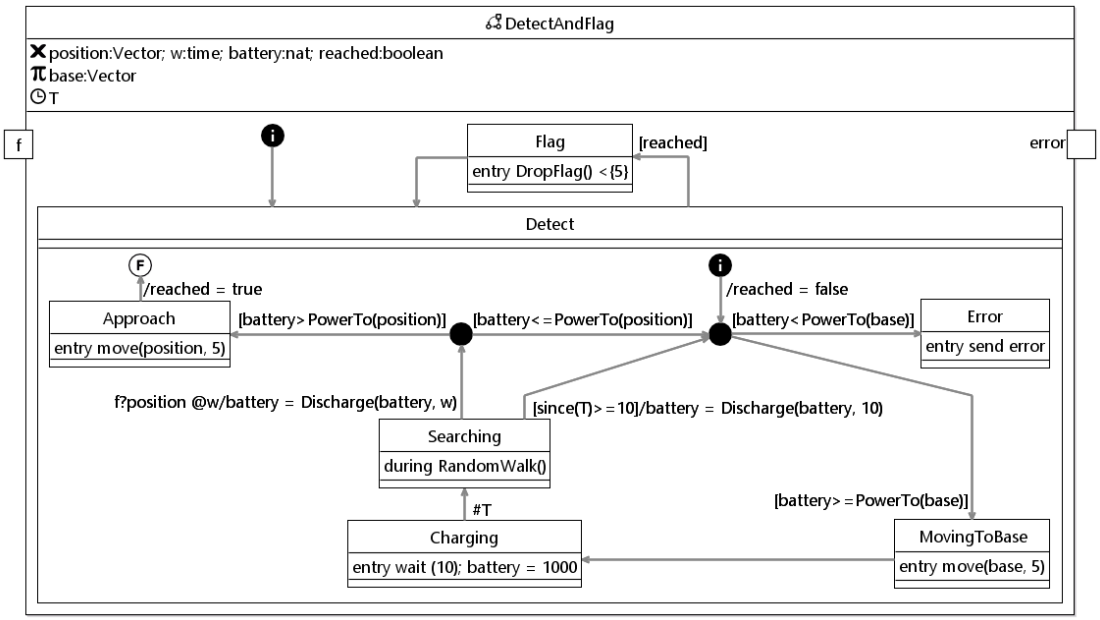


Figure 7.3: Detect and Flag with time

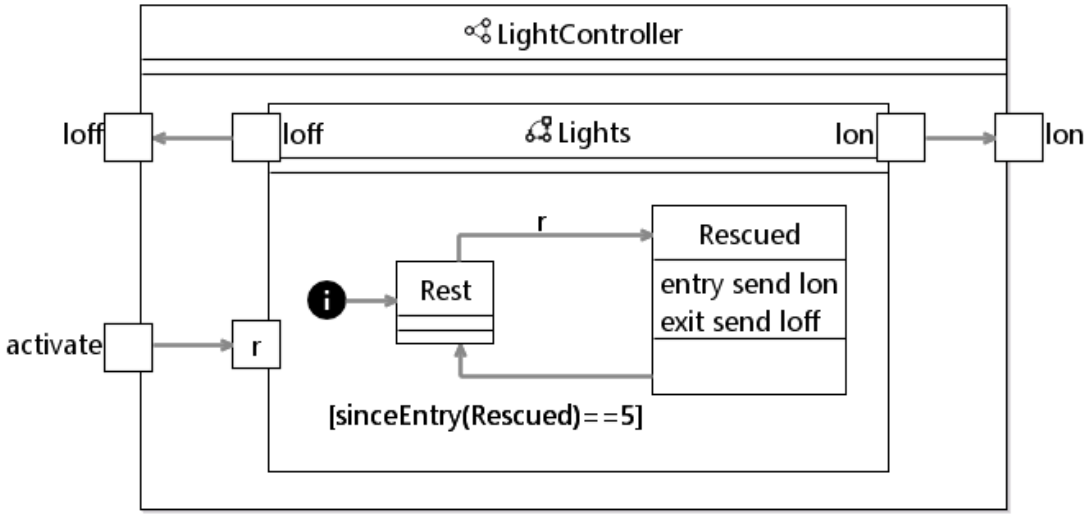


Figure 7.4: Light controller

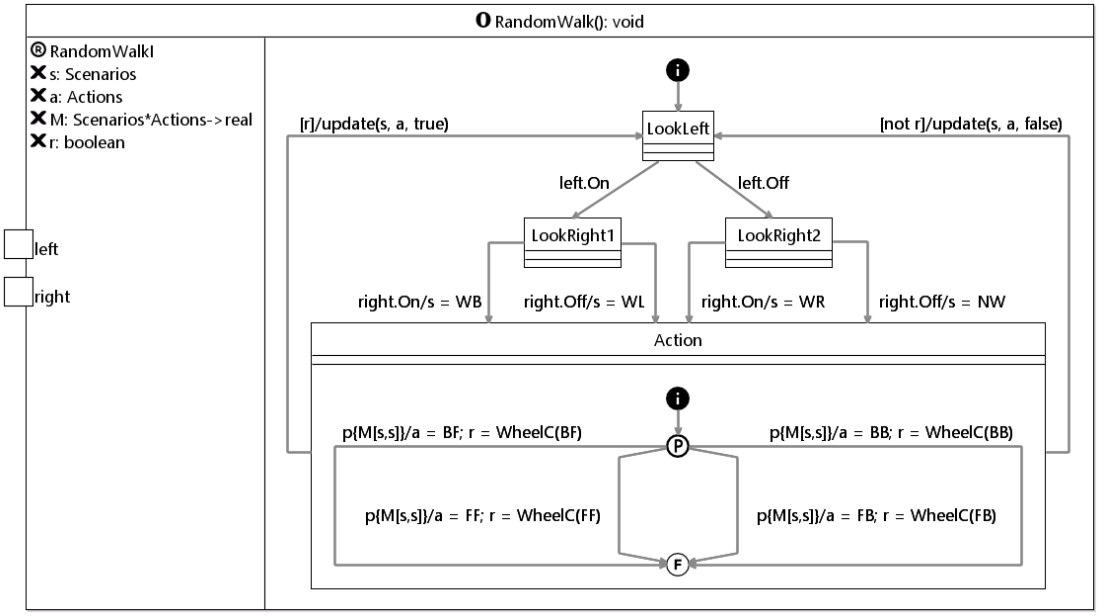


Figure 7.5: Random Walk Operation

7.2 Timed Language

7.3 Probabilistic Language

7.4 Hybrid Language

7.5 Tool Support

Chapter 8

Conclusions

We have presented here, RoboChart, a new notation for modelling of robots. It is based on UML state machines, but includes the notions of robotic platform and controller, synchronous and asynchronous communications, an API of operations common to autonomous and mobile robots, a well defined action language, pre and postconditions, and time and probability primitives.

We have described a semantics for the core constructs of RoboChart. It uses CSP, but we envisage its extension to use *Circus* [4], a process algebra that combines Z [29] and CSP, and includes time constructs [28]. We can already reason about untimed and non-probabilistic robotic systems [10]. Use of *Circus* and its UTP foundation will enable use of theorem proving as well as model checking. Work on probability is available in the UTP [31], but we will pursue an encoding of Markov decision processes in the UTP.

An approach for writing object-oriented simulations of RoboChart diagrams has also been defined. Automatic generation of simulations is possible and part of our future work. Verification of correctness of simulations will use the object-oriented version of *Circus* [5], with a semantics given by the UTP theory in [30].

Finally, we have presented RoboTool, a user friendly tool for modelling and verification of core RoboChart diagrams. Extensions of RoboTool will support the timed and probabilistic primitives, and automatic generation of simulations.

RoboChart itself misses support for modelling the environment and the robotic platforms in model detail. It is also in our plans to take inspiration from hybrid automata [12] to extend the notation, and from the UTP model of continuous variables [9] to define the semantics.

8.1 Related work

Another graphical domain-specific language for robotics is presented in [7]. It also aims to support design modelling and automatic generation of platform-independent code. It was defined as a UML profile. Model-based engineering of robotic systems is also advocated in [24], where a component-based framework that uses UML to develop robotics software is presented. In contrast, RoboChart is a small language, with a well defined semantics to support sound generation of formal models as well as simulations. It is a generic domain-specific language for designing and verifying the robot controllers.

There are a multitude of models for UML state machines. Kuske *et al.* [16] gave an integrated semantics for UML class, object, and state-machine diagrams using graph transformation. Rasch and Wehrheim [22] presented integrated semantics in CSP for (extended) class diagrams and state machines. Davies and Crichton [6] described CSP models for UML class, object, statechart, sequence and collaboration diagrams. Broy *et al.* [3] presented one of the first foundational semantics for a subset of UML2. Similarly, our semantics gives a precise characterisation of state machines and is close to [22] and [6] in our use of CSP. Our state machines, however, do not include components like history junctions and inter-level transitions to enable a compositional semantics.

UML has a simple notion of time. Its profile UML-MARTE [27] supports logical, discrete and continuous time through the notion of clocks. Complex constraints may be specified using CCSL (Clock Constraint Specification Language). Specification of time budgets and deadlines is focused on particular instances of behaviour specified through sequence and time diagrams. It is not possible to define timed constraints in terms of transitions and states.

UML-RT [26], an extension to UML, includes the notion of capsules, which encapsulate state machines; communication between capsules takes place through ports, whose valid communications are defined by protocols. A timing protocol can act as a timer by raising timeouts in response to the passage of a certain amount of time. It is not obvious how timed constraints, such as deadlines, can be specified directly on state machines beyond informal annotations.

Timed automata [2] caters for timed models using synchronous continuous-time clocks. Interesting properties can be checked using the model checker UPPAAL. Modelling the execution time of operations, and more complex constraints, requires UPPAAL patterns consisting of additional states and appropriate state invariants. Our aim is to provide a rich language suitable for directly capturing timed aspects of interest in robotic controllers.

Calendar automata [8] have been used to model time-triggered architecture systems. Support for model-checking is available using SAL. Calendar automata adopts a strict interleaving between time evolving till the next calendar entry, and events taking place. This model, strictly less expressive than timed automata, is not adequate for modelling scenarios where events do not necessarily alternate with time passing, and where transitions may be nondeterministic.

In [21] a semantics is given for a subset of UML-RT in *Circus* without considering time. An extension to UML-RT is considered in [1] with semantics given in terms of CSP+T [32], an extension of CSP that supports the timing of events. Inspired by the constructs of CSP+T, in [1] annotations are added for recording the occurrence time of events and constraining the occurrence time of other subsequent events. Although timed primitives such as *since* bear a resemblance, we have a richer set of primitives inspired by timed automata and Timed CSP [25].

Appendix A

Credits

Icons used in RoboTool and this report have been obtained from www.flaticon.com. Individual credits are given below.

 Icon made by Iconnice from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Sarfraz Shoukat from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Dario Ferrando from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Lyolya from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Google from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Revicon from www.flaticon.com is licensed by CC 3.0 BY

 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

Bibliography

[1] K. B. Akhlaki, M. I. C. Tunon, J. A. H. Terriza, and L. E. M. Morales. A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. *Science of Computer Programming*, 65(1):41–56, 2007.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.

[4] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.

[5] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277–296, 2005.

[6] J. Davies and C. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Formal Aspects of Computing*, 15(2-3):118–145, 2003.

[7] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. *Simulation, Modeling, and Programming for Autonomous Robots*, chapter RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pages 149–160. Springer, 2012.

[8] B. Dutertre and M Sorea. *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems: Joint International Conferences on Formal Modeling and Analysis of Timed Systmes*, chapter Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata, pages 199–214. Springer, 2004.

[9] S. Foster, B. Thiele, A. L. C. Cavalcanti, and J. C. P. Woodcock. Towards a UTP semantics for Modelica. In *Unifying Theories of Programming*, Lecture Notes in Computer Science. Springer, 2016.

[10] S. Foster, F. Zeyda, and J. C. P. Woodcock. Isabelle/UTP: A Mechanised Theory Engineering Framework. In D. Naumann, editor, *Unifying Theories of Programming*, volume

8963 of *Lecture Notes in Computer Science*, pages 21–41. Springer, 2015.

[11] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3: A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.

[12] T. A. Henzinger. The theory of hybrid automata. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

[13] J. A. Hilder, N. D. L. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. A. Kilgour, J. Timmis, and A. M. Tyrrell. Chemical detection using the receptor density algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1730–1741, 2012.

[14] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[15] D. N. Jansen, H. Hermanns, and J.-P. Katoen. A Probabilistic Extension of UML Statecharts. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 355–374. Springer, 2002.

[16] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In M. Butler, L. Petre, and K. SereKaisa, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.

[17] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[18] Wenguo Liu and Alan F. T. Winfield. Modeling and optimization of adaptive foraging in swarm robotic systems. *The International Journal of Robotics Research*, 29(14):1743–1760, 2010.

[19] Object Management Group. OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. Technical report, OMG, 2011.

[20] Object Management Group. *OMG Unified Modeling Language*, March 2015.

[21] R. Ramos, A. C. A. Sampaio, and A. C. Mota. A Semantics for UML-RT Active Classes via Mapping into *Circus*. In *Formal Methods for Open Object-based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 99–114, 2005.

[22] H. Rasch and H. Wehrheim. Checking consistency in UML diagramms: Classes and state machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2003.

[23] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[24] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *14th International Conference on Advanced Robotics*, pages 1–8. IEEE, 2009.

[25] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.

[26] B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.

[27] B. Selic and S. Grard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Morgan Kaufmann Publishers Inc., 2013.

[28] A. Sherif, A. L. C. Cavalcanti, J. He, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.

[29] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[30] F. Zeyda, T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. A modular theory of object orientation in higher-order UTP. In *Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 627–642. Springer, 2014.

[31] H. Zhu, J. W. Sanders, He Jifeng, and S. Qin. Denotational Semantics for a Probabilistic Timed Shared-Variable Language. In B. Wolff, M.-C. Gaudel, and A. Feliachi, editors,

Unifying Theories of Programming, volume 7681 of *Lecture Notes in Computer Science*, pages 224–247. Springer, 2013.

[32] J. J. Zic. Time-constrained Buffer Specifications in CSP + T and Timed CSP. *ACM Transactions on Programming Languages and Systems*, 16(6):1661–1674, 1994.