Modelling and verification for swarm robotics

Ana Cavalcanti[1], Alvaro Miyazawa[1], Augusto Sampaio[2],
Wei Li[3], Pedro Ribeiro[1], and Jon Timmis[3]

[1] Department of Computer Science, University of York, UK
[2] Centro de Informática, Universidade Federal de Pernambuco, Brazil
[3] Department of Electronic Engineering, University of York, UK

Abstract. RoboChart is a graphical domain-specific language, based on UML, but tailored for the modelling and verification of single robot systems. In this paper, we introduce RoboChart facilities for modelling and verifying heterogeneous collections of interacting robots. We propose a new construct that describes the collection itself, and a new communication construct that allows fine-grained control over the communication patterns of the robots. Using these novel constructs, we apply RoboChart to model a simple yet powerful and widely used algorithm to maintain the aggregation of a swarm. Our constructs can be useful also in the context of other diagrammatic languages, including UML, to describe collections of arbitrary interacting entities.

1 Introduction

In [15, 22], RoboChart, a domain-specific language tailored for robotics, is presented. The core of RoboChart is based on state machines, a modelling construct widely employed in the embedded-software and robotics domains. RoboChart is endowed with a denotational semantics that supports both automatic and semi-automatic verification in the form of model checking and theorem proving.

Unlike general purpose notations, like, for example, UML [12], RoboChart is concise, with well defined syntax and well-formedness conditions that guarantee meaningfulness of models. RoboChart also includes constructs for modelling abstraction (given types, operations definitions via pre and postconditions, and so on), nondeterminism, and time. Most languages of the same nature avoid abstraction and nondeterminism since these features make code generation difficult or impossible. While time is considered in UML MARTE [11] and UML-RT [18], the RoboChart approach based on budgets and deadlines is distinctive.

RoboChart is supported by RoboTool, which provides facilities for graphical modelling, validation, and automatic generation of C++ simulations. Importantly, RoboTool automatically generates also the formal semantics of RoboChart models. The semantics definition uses the process algebra CSP [20], and RoboTool also provides a direct connection to the FDR model checker for CSP [10].

RoboChart as presented in [15], however, enforces the use of design patterns appropriate for systems composed of a single robot. While such applications are relevant and widespread, collections of robots, that is, swarms, are becoming

popular. In robotic swarms, a goal such as pushing a block is achieved by a collection of simple and cheap robots that individually cannot complete the task, but can in cooperation. Their relative low cost allows for defective robots to be easily replaced, making a swarm more robust than single robot applications.

Here, we extend RoboChart to support modelling and verification of collections of interacting robots. We focus on the nature of the robots, and how they communicate with each other. With these, we can specify the behaviour of a swarm as the result of the interaction of a(n unspecified) number of robotic systems. We can describe abstractly heterogeneous collections of interacting robots through the use of underspecified constants, different robot specifications, and communication patterns. We introduce a new inter-robot communication mechanism for fine-grained control over interactions, supporting both identification of the source of interactions and restriction of the possible targets.

Although our focus is on capturing precisely descriptions of swarm applications from the robotics literature, our modelling constructs can be useful to model arbitrary heterogeneous distributed systems. As far as we know, our constructs are entirely novel. In UML or SysML [17], for instance, a variant of UML for systems modelling, the definition of a collection and their connections requires two diagrams, and fixes the number of components in the collection.

We note, however, that as a design language, RoboChart does not cover the explicit specification of global properties of the swarm, such as aggregation. A property language for RoboChart is part of our agenda for future work.

Section 2 briefly introduces the RoboChart notation by means of a simple example of an aggregation algorithm running on a single robot. Section 3 describes the extensions necessary to accommodate the new features, and Section 4 describes their semantics. Section 5 reviews the tool support available for RoboChart and its extensions. Section 6 discusses related work. Finally, Section 7 concludes and discusses further opportunities for work.

2 RoboChart and its semantics

Here, to illustrate the RoboChart notation, we present in Section 2.1 a model of the alpha algorithm [4][4], whose goal is to maintain a collection of robots in an aggregate. This algorithm estimates the number of neighbours of a robot, and uses that to decide whether to maintain its direction or turn around. The idea is that the robot recognises when it has moved away from the aggregate by counting the number of neighbours. Later, we show how our support for collections allows for a much simpler and clearer model. In Section 2.2, we briefly introduce CSP. Finally, Section 2.3 gives an overview of the semantics of RoboChart.

2.1 The notation

A RoboChart model describes two main aspects of an application: structure and behaviours. The root element of a model is called a module, and provides an

[4] www.cs.york.ac.uk/circus/RoboCalc/case-studies/

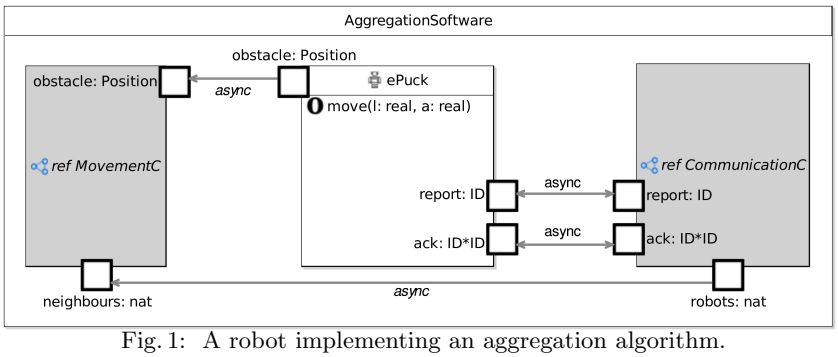

Fig. 1: A robot implementing an aggregation algorithm.

overall view of the system. The module for our example is in Figure 1. A module specifies two aspects of the applications: (1) assumptions about the platform, and (2) available behaviours. The assumptions are modelled via a robotic-platform block, illustrated in Figure 1 by the block ePuck, which describes the variables, operations, and events that must be available for the application to be feasible.

In this example, the platform ePuck abstracts a piece of hardware containing a number of sensors and actuators. The operation move that takes two parameters l and a, both of type real, models the actuator responsible for moving the robot forward and turning. The event obstacle represents an obstacle sensor and communicates the position of the obstacle. The events report and ack represent sensors and actuators responsible for inter-robot communication.

The event obstacle carries a value of type Position that is defined by an enumeration containing the values left and right. The event report carries a value of type ID representing the source of the communication, and ack carries a pair of ID values representing the source and target of the communication. ID is an abstract type about which nothing is assumed except for its non-emptiness.

The behaviours of a RoboChart module are specified by one or more controllers; they run in parallel and interact with each other and with the robotic platform. The possible interactions are indicated by connections, which, at the level of modules, are either synchronous or asynchronous.

In our example, the aggregation behaviour is decomposed into two controllers: MovementC and CommunicationC. The first describes how the robot moves based on the number of neighbours; the second uses inter-robot communications to estimate that number. There are four interaction points: the occurrence of the event obstacle represents an interaction between the robotic platform and MovementC to communicate the position of the obstacle with respect to the robot; report and ack are used to interact with other robots, intermediated by the robotic platform, and communicating the identity of the robot; finally robots in CommunicationC is used for interaction with MovementC through neighbours. (Connected events do not need to have the same name, just the same type.)

Controllers are defined by one or more state machines interacting via synchronous connections. Figure 2 shows CommunicationC. It declares two interfaces

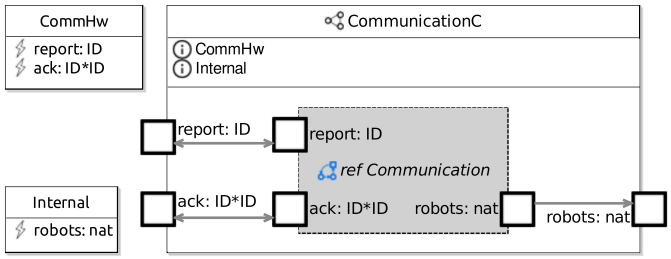

Fig. 2: Communication controller of the aggregation algorithm.

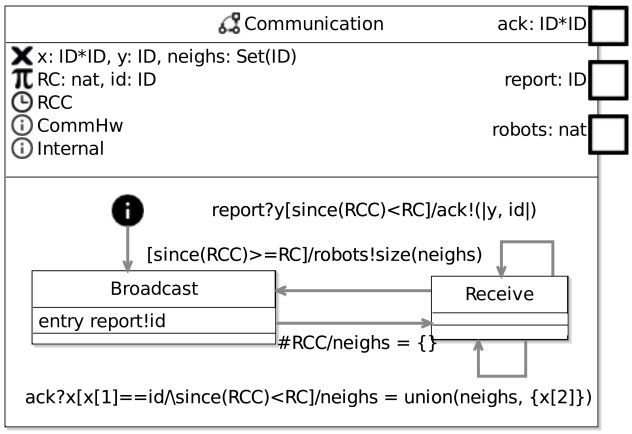

Fig. 3: Communication state machine of the aggregation algorithm.

CommHw and Internal, also shown in Figure 2. They define the events report, ack and robots, connected to identically named events in the referenced state machine Communication, whose definition is shown in Figure 3. Just like the connection between CommunicationC and the platform, the connections for report and ack are bidirectional. The connection for robots provides an output, matching the connection between CommunicationC and MovementC.

Communication models a cyclic behaviour, where each cycle takes RC time units. At each cycle, the machine indicates its presence via report, and then monitors for responses from other robots through ack. At the same time, reports from other robots are acknowledged. This allows estimating how many robots are within the communication range. The size of the set of neighbours is sent through robots and propagated to MovementC to decide how to move.

Similarly to the controller CommunicationC, the machine Communication declares the two interfaces CommHw and Internal to define the events it uses. Additionally, it declares two constants: id of type ID, representing the identifier of the robot and RC with type nat of natural numbers. There are also three variables: x

of type ID*ID (pair of values of type ID) records values received through ack, y of type ID records values received through report, and neighs of type Set(ID) (sets of values of type ID) records the identifiers of the robots that respond to the event report. Finally, Communication declares a clock RCC used to mark the cycles.

Communication initially enters the state Broadcast executing its entry action, which sends id through report. After that, the transition to the state Receive is taken, since it has no guard or event. This resets the clock (#RCC) and assigns the empty set ({}) to neighs. From Receive, there are three possible transitions. If the clock runs out (since(RCC)>=RC), the size of neighs is communicated via robots, and Broadcast is entered. If, before the clock runs out (since(RCC)<RC), the event report occurs, the pair (|y,id|) formed input y taken and id is sent through ack, and the machine stays in Receive. Finally, if an event ack, where the first element x[1] of its tuple input x is id, happens before the clock runs out, the second element of x is added to neighs, and the machine stays in Receive.

The semantics of RoboChart captures this behaviour and that of the whole module as a CSP process. We next give a brief overview of CSP.

2.2 CSP

Systems and components are modelled in CSP via processes. They are all regarded as black boxes and defined by the patterns of their interactions with each other and the environment. Interaction is via atomic and instantaneous events.

Accordingly, the semantics of a RoboChart model is defined by a process that captures the behaviour of its module. Each component, controller, and state machine is defined by a process as well. The events of the module process match the interface of the robotic system characterised by the robotic platform. They correspond to accesses to variables, to calls to operations of the platform, and to communications using the RoboChart events of the platform.

We explain the CSP notation as we use it; Table 1 gives a summary. A core operator is prefixing $c \rightarrow P$; it describes a process that engages in a communication (event) c and then behaves as the process P. The event may be an input $c?x$ that records the value input via a channel c in a variable x, an output $c!e$ of the value of an expression e, or a simple synchronisation on c.

The parallel operators are also very important. $P \,[\![\, cs \,]\!]\, Q$ defines the parallel execution of P and Q, synchronising on the events in the set cs. Communications internal to a component or system can be hidden using the operator $P \setminus cs$, which hides the events in the set cs in the execution of P.

A dialect of CSP, called tock-CSP, uses a special event *tock* to mark the passage of time. We use tock-CSP to capture the semantics of the timed constructs of RoboChart: clocks, budgets, and deadlines.

Most importantly for our agenda of work, CSP has a relational predicative semantics defined using the Unifying Theories of Programming (UTP) [13]. Support for this semantics in Isabelle/HOL [8] means that our CSP semantics for RoboChart is a front-end for a UTP theory. With that, we can carry out verification using theorem proving. For large models, and for swarm models in particular, this is crucial to ensure scalability.

Symbol	Name	Description
$Skip$	skip	Terminate immediately without any side effects.
$P \llbracket cs \rrbracket Q$	parallel composition	Run P and Q in parallel synchronising on events in cs.
$P \mid\mid\mid Q$	interleaving	Run P and Q in parallel without synchronisation.
$\{\!\mid e \mid\!\}$	channel set	Set of all possible events associated with channel e.
$c \rightarrow P$	prefix	Synchronise on channel c and then behave like P.
$P \setminus cs$	hiding	Run P with events in cs hidden.
$P[\![c \leftarrow d]\!]$	renaming	Rename the occurrences of event c to d in P.
$\mid\mid\mid i : I \bullet P(i)$	replicated interleave	Run $P(i)$ in parallel for all i in I without synchronisation.

Table 1: Summary of CSP operators

2.3 Semantics

The module process is defined by the parallel composition of the processes that model its controllers, where events are renamed and the synchronisation set is constructed so that the controllers interact according to the module connections. Asynchronous connections between controllers are modelled by single-cell buffers. For example, the semantics of the module in Figure 1 is as follows.

$$
\begin{array}{l}
AggregationRobot = \\
\quad \left(\left(\begin{array}{l} MovementC[MovementC\_obstacle.out \leftarrow ePuck\_obstacle.in] \\ \quad \mid\mid\mid \\ CommunicationC[\, CommunicationC\_report.out \leftarrow ePuck\_report.in, \\ \qquad\qquad\qquad CommunicationC\_report.in \leftarrow ePuck\_report.out, \\ \qquad\qquad\qquad CommunicationC\_ack.out \leftarrow ePuck\_ack.in] \\ \qquad\qquad\qquad CommunicationC\_ack.in \leftarrow ePuck\_ack.out] \end{array} \right) \right. \\
\qquad \left. \begin{array}{l} \llbracket \{\!\mid CommunicationC\_robots, MovementC\_neighbours \mid\!\} \rrbracket \\ Neighbours\_Buffer(\langle\rangle) \end{array} \right) \\
\quad \setminus \{\!\mid CommunicationC\_robots, MovementC\_neighbours \mid\!\}
\end{array}
$$

AggregationRobot composes in parallel the controller processes *MovementC* and *CommunicationC*, with their channels that represent events connected directly to the platform, that is, *CommunicationC_report* and *CommunicationC_ack*, renamed to the platform channels *ePuck_report* and *ePuck_ack*. For each event, we have a channel that takes tokens *in* and *out* that identify the direction of communication: *in*put or *out*put. The renamings identify the events to establish the connections between the controllers and the platform. The inputs of the platform are identified with the outputs of the controllers and vice-versa. For an unidirectional connection, like that for obstacle, just one renaming is needed.

In the above example, the controller processes do not communicate because they do not interact synchronously. So, the parallelism is an interleaving ($\mid\mid\mid$).

The parallel composition of the controller processes is composed in parallel with a buffer process $Neighbours\_Buffer(\langle\rangle)$ that records RoboChart events input through a channel $CommunicationC\_robots$ modelling the RoboChart event robots, and sends them through another channel $MovementC\_neighbours$ modelling neighbours. Communications using these channels are hidden as the RoboChart events they represent are internal to the module (see Figure 1). The set $\{\!|c|\!\}$ contains all events that represent communications over the channel c.

The semantics of controllers is similarly defined as the parallel composition of the processes that model its state machines. Since all connections between state machines are synchronous, there is no need for buffers. Channels corresponding to events of the state machine connected to the controller are renamed to channels of the controller. In our example, since the controller CommunicationC contains only one state machine, only the renamings take place.

$$
\begin{array}{l}
CommunicationC = \\
\quad Communication[\, Communication\_robots \leftarrow CommunicationC\_robots \\
\qquad\qquad\quad Communication\_report \leftarrow CommunicationC\_report \\
\qquad\qquad\quad Communication\_ack \leftarrow CommunicationC\_ack]
\end{array}
$$

The renamings establish the connections between the events of the controller CommunicationC and those of the state machine Communication.

The processes for state machines are defined in a compositional way in terms of processes for states and transitions. They capture the control flow defined by the machine in terms of CSP events that represent accesses and updates to variables required or provided by the machine, calls to operations, and occurrence of RoboChart events. Compositionality is important for verification and can be achieved because we rule out constructs like inter-level transitions, for example. The complete semantics of RoboChart is described and formalised in [22].

Next, we discuss the metamodel of the new constructs to deal with collections, and present an updated version of our example that uses these constructs.

3 Collections in RoboChart: overview and metamodel

Our example models the controller of a single robot, but the application itself is a swarm, where multiple robots communicate to estimate their numbers of neighbours. To capture this behaviour, in the previous section we explicitly model the source and destination of the event ack and the source of the event report.

To model the swarm as a whole and simplify the specification of the communications, we include two new features in RoboChart. We have a construct to specify (heterogeneous) collections, and communications that support extraction and restriction of attributes, namely, source and target robots.

Here, we describe our reworked example (Section 3.1), and discuss the collection (Section 3.2) and the extended communication (Section 3.3) mechanisms.

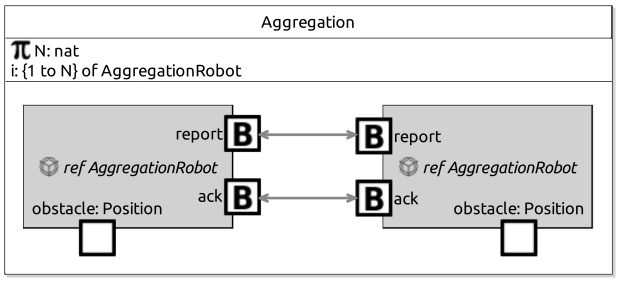

Fig. 4: Collection for the aggregation system.

3.1 Extended example

In the previous section, the model of our example focused on an individual robot. Here, we model how multiple robots interact with each other, and show how the original model can be reworked to use our extended communication mechanism. We model a swarm of N robots interacting through broadcast events.

Figure 4 shows a collection diagram for our example. It defines the collection as N instances of the module AggregationRobot indexed by values i ranging from 1 to N, where N is an uninitialised constant of type nat. The index specified in the collection is used to identify its individual robots.

Within the bottom compartment of the collection diagram, there are two placeholders for AggregationRobot, with their events report and ack connected. The placeholders stand for any two distinct instances of AggregationRobot. They show how any two robots modelled by AggregationRobot interact.

While communication within a robot (between controllers and state machines) is one-to-one, communication between robots in a collection is a broadcast: all instances of a module with a broadcast event can potentially receive messages from all other instances that are connected to that event as defined in the collection. These events are indicated by the letter B inside the event box.

This system of idealised broadcast connections may seem too restrictive, but, in conjunction with the communication mechanisms discussed in Section 3.3, it can be used to model more constrained forms of interaction. We can, for example, model that communication is only sent to a subset of other robots, or even to a particular robot. These can capture the fact that there may be robots out of range, or that there is a communication device in the platform with a protocol that identifies when the communications are directed to its robot.

A broadcast event has implicit from and to attributes of a generic type ID to identify the source and target of the communication. The type ID is that of robot identifiers; it is instantiated in a collection diagram via the definition of instances of modules (robots). The diagram in Figure 4, for example, defines instances with identifiers i in the range 1 to N. In doing so, it instantiates the type ID for AggregationRobot to the set of natural numbers between 1 and N.

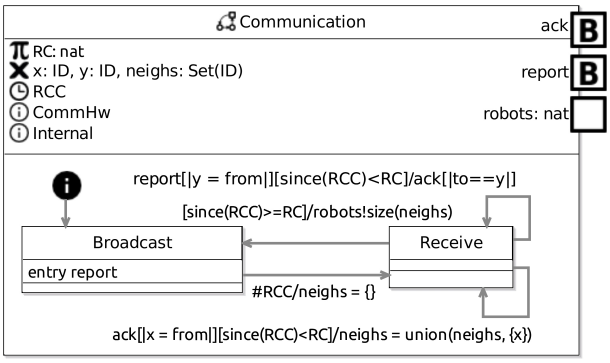

Fig. 5: Broadcast communication in the Communication state machine.

Using the collection communication mechanism, we can rewrite the machine in Figure 3 to eliminate the parameters of the report and ack events, using the attributes of the broadcast events instead. The updated version is in Figure 5.

In this version, the transition triggered by a report event uses an assignment to record its attribute from in the variable y. It is used in the action that sends the event ack to restrict its attribute to in a predicate that equates to to y. We use a predicate, not an assignment to restrict to, so that, in general, several robots can be targetted. We record the source of the report to direct the acknowledgement to the right robot, since the scope of from and to is the broadcast event.

Similarly, upon receipt of the event ack the source is stored in x. It is that identifier x that is recorded in the set neighs. We note, in particular, that the condition x[1]==id in the guard of the corresponding transition in Figure 3, becomes unnecessary, since ack is accepted only when the target is the robot id.

In what follows, we describe the metamodel and well-formedness conditions for the collection diagrams and broadcast communications.

3.2 Collections

A collection diagram, illustrated in Figure 4, describes exactly which types of robots (RoboChart Modules) form the collection and how many instances of each type exist. Furthermore, it specifies how different instances or types of robots can communicate using connections of broadcast events.

In the metamodel of RoboChart, a collection diagram is modelled by a construct RCCollection shown in Figure 6. It can contain four components. A variableList declares the constants used to define the number of instances of each type of robot. In the metamodel, they are identified as variables, but a well-formedness condition ensures that they are constants. The value of a constant may or may not be defined, like in our example in Figure 4.

Collections can also have Instantiations consisting of an index with a range, of the Module being instantiated, and of parameters that initialise unspecified

constants in the module, if any. For example, we could define the number alpha of neighbours used as a threshold to determine whether a robot should turn as a constant in AggregationRobot. In this case, we may define a value for alpha in the collection, perhaps in terms of the number N of robots in the swarm.

A well-formedness condition ensures that the range is defined by an Expression that denotes a finite set. We note that a collection may have just one instance of a particular Module, so the range may be a singleton. This may be, for example, a model for a collection that contains one robot controlling others.

Another well-formedness condition ensures that the parameters defined via InstantiationParameters are values for constants of the Module instantiated. There may be have several instantiations with different values for these constants.

Placeholders correspond to generic instances of modules. They are references to a Module, that is, an element of ModuleRef. A well-formedness restriction ensures that these are modules that occur in one of the instantiations. An extra attribute IDInst of Module records an instantiation, if any, of the generic type ID of identifiers for instances of the robots defined by the Module.

Since ID is a generic type, there are no operations beyond equality and inequality that can be used to manipulate its elements. If more operations are needed, ID needs to be instantiated. For example, if the models in a component (module, controller, or machine) use arithmetic operations on identifiers, the type ID needs to be instantiated to a numeric type. A well-formedness condition ensures that the same instantiation is used in the collections that use the module. Different instantiations cannot be used in the same context.

Finally, we can include Connections between the placeholders establishing the possible interactions between instances. Connections are between ConnectionNodes, one of which is a ModuleRef. A Connection can be bidirectional, and in a collection, a well-formedness guarantees that it is asynchronous.

If there are several instantiations for the same Module, giving different values for its constants, there may be more than two placeholders for such a Module. In general, for each instantiation, there may be up to one or two placeholders for its Module, depending on whether the range is a singleton or a larger set. An empty range has a well defined semantics that can be used to explore behaviour in the presence of missing robots, perhaps due to failure. Normally, however, we expect that such a range is left unspecified, so that the collection may have instances of the robot defined by the Module in some scenarios.

Events have a type that define the values that can be communicated. In addition, broadcast events model the form of communication used in swarms. Well-formedness requires that all connected events in a collection are broadcast events, and broadcast events are only connected to other broadcast events.

The complete metamodel of RoboChart is available at [22], where elements omitted in Figure 6, like Type or Expression, are defined.

3.3 Communications

The collection construct provides a general view of the potential communication patterns between robots. In particular, it clearly specifies which communications

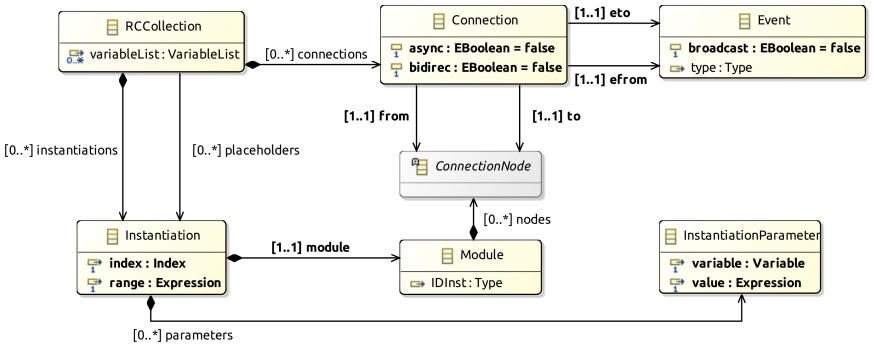

Fig. 6: Metamodel of collections.

cannot happen. For example, in Figure 4, it is specified that there are no interactions using the event obstacle. Nevertheless, the actual communication pattern cannot be specified at this level. For instance, in our example, while the event report is used to communicate with all other robots in the swarm, the event ack is used in a more restricted fashion. It is used to communicate only with the robot from which a report has been received.

Although a communication device available in a platform may not be able to enforce a restricted protocol like this, it is simple to program functionality to provide directed communication. The possibility to specify interactions that follow a particular protocol allows us to construct more abstract models, where the programming of any particular protocol is assumed to be in place.

The restricted interactions need to be specified at the level of the communication definitions using intrinsic information about their source and target. Triggers, which are used in transitions and actions to define communications, include extra source and target attributes _from and _predicate. They allow both identification and restriction of the participants.

The attribute _from records the value of a predefined variable from: the identifier of the source of the communication. The _predicate attribute defines a restriction on the target of the communication identified by a predefined variable to. A well-formedness condition ensures that to is free in _predicate. The variables from and to have type ID and are local to a Trigger whose event is a broadcast.

These extensions to Trigger to consider the predefined variables from and to affects the usage of events in triggers in transitions and actions. In an input c?x[|v = from|], the occurrence of the communication c?x is accepted, and the identifier of the source of the communication is recorded in the variable v. An output c!e[|to in S|], for example, sends the value of e via c to each instance specified by the collection diagram whose identifier belongs to the set S.

In the next section, we define the semantics of our collection constructs.

4 Collections in RoboChart: semantics

The semantics of RoboChart is given by a function $[\![\_]\!]_{\mathcal{M}}$ from RoboChart modules to CSP processes. This function is defined compositionally over the metamodel of RoboChart, in terms of other functions that calculate the semantics of the controllers and connections that form the module.

The timed semantics is also defined by a function from RoboChart modules, but its range is the set of tock-CSP processes. The definition of this function reuses much of the original (untimed) semantics. In particular the description of the collection semantics we provide here is valid in the context of both semantics. We note that a tock-CSP process is itself a CSP process.

We define a new semantic function from RCCollection to CSP processes, and modify three functions of the original semantics. The function $[\![\_]\!]_{\mathcal{M}}$ is modified just to add a parameter *id* to the module process to record the module identifier. It is used by the processes defined by the functions $[\![\_]\!]_{Trigger}$ and $[\![\_]\!]_{Statement}$, which specify the semantics of triggers in transitions and in actions.

The definitions of $[\![\_]\!]_{Trigger}$ and $[\![\_]\!]_{Statement}$ deal with the extra properties _from and _predicate of triggers. In addition, events corresponding to accesses to variables of the platform, to calls to its operations, and to simple platform events, that is, that are not for broadcast, get *id* as an extra parameter. This is so that the individual interactions of the platform with the controller and the environment can be distinguished. Before we discuss all the affected semantics functions, we present the CSP process *Aggregation* below that defines our example collection to illustrate the overall idea of the semantics.

$$Aggregation = \left(\begin{array}{l} |||\ i : 1..N \bullet AggregationRobot(i) \\ \quad [\![\{\!| report.in, report.out, ack.in, ack.out |\!\}]\!] \\ |||\ (i,j) : 1..N \times (1..N \setminus \{i\}) \bullet \\ \quad \big(Buffer(\langle\rangle, report, i, report, j) \ |||\ Buffer(\langle\rangle, ack, i, ack, j) \big) \end{array} \right)$$

The robots do not communicate directly. Using, for example, infrared or radio devices, they communicate asynchronously. So, formally, they communicate via buffers. Accordingly, in *Aggregation*, we combine the instances with identifiers 1 to N in interleaving. Instantiation defines the parameter *id* of the module.

There is a buffer for each direction of each connection between each pair of robots. So, for each pair (i, j) of robots i and j, we have a buffer to connect their report and their ack events. We note that i and j are different robots, since j is taken from the set $(1 \,..\, N \setminus \{i\})$, which excludes i. $Buffer(elems, e_1, id_1, e_2, id_2)$ defines a buffer containing the elements in the sequence *elems*, taking inputs from robot id_1 via the events $e_1.in$ and producing outputs to robot id_2 via events $e_2.out$. The buffers are initially empty: their sequence of elements is $\langle\rangle$.

The buffers do not interact with each other, so they are also combined in interleaving. Their inputs and outputs are connected to the events of the module processes. So, the module processes and the buffers are composed in parallel synchronising on the events $report.in$, $report.out$, $ack.in$, and $ack.out$ corresponding to the ends of the bidirectional connections in Figure 4.

Rule 1. Semantics of Collections $[\![\mathsf{c} : \mathsf{RCCollection}]\!]_{\mathcal{C}ol} : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\left(
\begin{array}{l}
\left(
\begin{array}{l}
|\!|\!| \ \mathsf{inst} : \mathsf{c.instantiations} \bullet |\!|\!| \ i : \mathsf{inst.range} \bullet [\![\mathsf{inst.module}]\!]_{\mathcal{M}}(i) \\
\quad |[\{\!| e_1, e_2 \mid (e_1, e_2) \leftarrow \mathsf{connectedEvents(c)} |\!\}]| \\
\left(
\begin{array}{l}
\left(
\begin{array}{l}
|\!|\!| \ \mathsf{conn} : \mathsf{c.connections} \bullet |\!|\!| \ (i,j) : \mathsf{inds(conn, c)} \bullet \\
\quad \mathit{Buffer}(\langle\rangle, \mathsf{eventId(conn.efrom)}, i, \mathsf{eventId(conn.eto)}, j)
\end{array}
\right) \\
|\!|\!| \\
\left(
\begin{array}{l}
|\!|\!| \ \mathsf{conn} : \mathsf{c.connections} \mid \mathsf{conn.bidirec} \bullet |\!|\!| \ (i,j) : \mathsf{inds(conn, c)} \bullet \\
\quad \mathit{Buffer}(\langle\rangle, \mathsf{eventId(conn.eto)}, j, \mathsf{eventId(conn.efrom)}, i)
\end{array}
\right)
\end{array}
\right)
\end{array}
\right) \\
\textsf{where} \\
\quad \mathsf{connectedEvents(c : Collection)} : \mathbb{P}(\mathsf{Event} \times \mathsf{Event}) = \\
\quad\quad \{\mathsf{conn} : \mathsf{c.connections} \bullet (\mathsf{eventId(conn.efrom)}, \mathsf{eventId(conn.eto)})\} \\
\quad\quad \cup \\
\quad\quad \{\mathsf{conn} : \mathsf{c.connections} \mid \mathsf{conn.bidirec} \bullet (\mathsf{eventId(conn.eto)}, \mathsf{eventId(conn.efrom)})\} \\
\quad \mathsf{inds(conn : Connection, c : Collection)} : \mathbb{P}(\mathsf{ID} \times \mathsf{ID}) = \\
\quad\quad \textsf{if } \mathsf{conn.from.ref} = \mathsf{conn.to.ref} \textsf{ then} \\
\quad\quad\quad \mathsf{range(conn.to, c)} \times \mathsf{range(conn.from, c)} \setminus \{i : \mathsf{range(conn.to, c)} \bullet (i,i)\} \\
\quad\quad \textsf{else} \\
\quad\quad\quad \mathsf{range(conn.to, c)} \times \mathsf{range(conn.from, c)} \\
\quad \mathsf{range(m : Module, c : Collection)} : \mathbb{P}\,\mathsf{ID} = (\iota \mathsf{i} : \mathsf{c.instantiations} \mid \mathsf{i.module} = \mathsf{m}).\mathsf{range}
\end{array}
$$

Rule 1 defines the semantic function $[\![\_]\!]_{\mathcal{C}ol}$ that specifies the CSP processes for collections. We use a simple meta-notation based on CSP itself; its terms are underlined. CSP terms are in the usual mathematical font.

The function $[\![\_]\!]_{\mathcal{C}ol}$ takes an element c of the type RCCollection defined in the metamodel and returns an element of CSPProcess, a CSP process, defined by the parallel composition of two interleavings as illustrated above. For each instantiation inst of c, that is, in the set $\mathsf{c.instantiations}$, we have a replicated interleave, which is itself combined in interleaving with the interleaving for the other instantiations. For each index i in the range $\mathsf{inst.range}$ of the instantiation, we have a process $[\![\mathsf{inst.module}]\!]_{\mathcal{M}}(i)$ combined in interleaving, where $[\![\mathsf{inst.module}]\!]_{\mathcal{M}}$ defines the semantics of the referred module $\mathsf{inst.module}$. This semantic function is as defined in [22] and previously illustrated in Section 2, but now the process it defines includes the extra parameter *id* as explained above.

The second interleaving is of buffer processes. There are two groups of such processes: the first models unidirectional communication, and the second complements the first with buffers for the reverse direction of communication for bidirectional connections. In each case, for each connection conn of c (from the set $\mathsf{c.connections}$), we have a replicated interleaving of processes modelling buffers. We have a buffer for each pair of instances of each module connected by conn.

The replicated interleavings are indexed by pairs (i, j) in the set $\mathsf{inds(conn, c)}$ containing identifiers of the instances of the modules that can communicate

through the connection conn. The definition of inds(conn, c) presented in Rule 1 takes into account the module of the two connected instances. If they are different (conn.from.ref = conn.to.refmodules is false), the pairs in inds(conn, c) are those in the cartesian product of the indices of the source and target modules. Otherwise, we need to discard the identity pairs (i, i), since there is no connection associating an instance of a module to itself.

The set of indices is specified using the function range. It takes a module m and a collection c in which it is instantiated, determines the unique (ℓ) instantiation whose module is m, and returns the associated range.

The second group of interleavings corresponding to the bidirectional connections is similar, except that the parameters of the process *Buffer* are reversed.

The two top-level interleavings are composed in parallel synchronising on the events corresponding to the source and target of the connections. This is calculated by the function connectedEvents, which takes a collection c and returns a set of pairs of events. It is presented in Rule 1. For each connection in c, we have a pair formed of the channels that model the source event (efrom) and the target event (eto) of the connection. For bidirectional connections (conn.bidirec), we also include the reversed pair in connectedEvents(c).

The semantics of triggers uses the *in* and *out* components of the broadcast channels as well as the source and target identifiers to coordinate the exchange of events between the module instances. To illustrate, we present the semantics of two communications in our example in Figure 5.

The semantics of report[|y = from|] is given by the CSP process below, which is used to define the semantics of the state machine, itself used as a component in the module process and, therefore, in the collection. Here, *id* is the identifier of the module defined, as explained above, as a parameter of its process.

$$report.out?from!id \rightarrow set\_y!from \rightarrow Skip$$

If the buffer process for the connection to the report event of a robot *id* contains a value, it can synchronise with this process. It accepts the source *from* of the connection as input, and assigns its value to the variable y using a communication on a channel set_y, and terminates (*Skip*). Variables in state machines are held in memories represented by processes with *set* and *get* channels.

The semantics of ack[|to == y|] is given by the CSP process below.

$$|||\ t : \{to \mid to \leftarrow ID, to == y\} \bullet ack.in!id?t \rightarrow Skip$$

While the trigger process for the previous example only synchronises with one of the buffers, the above process synchronises with all buffers whose target identifiers t satisfy the predicate $t == y$. As previously indicated, the CSP trigger processes illustrated above interact with the buffers in *Aggregation*.

The definition of the semantics of triggers requires a simple change to what is presented in [22] to specify enriched processes like those shown above, which can record identifier information, and synchronise with various buffers.

Validation of the semantics just described is provided by its mechanisation RoboTool, which is presented in the next section.

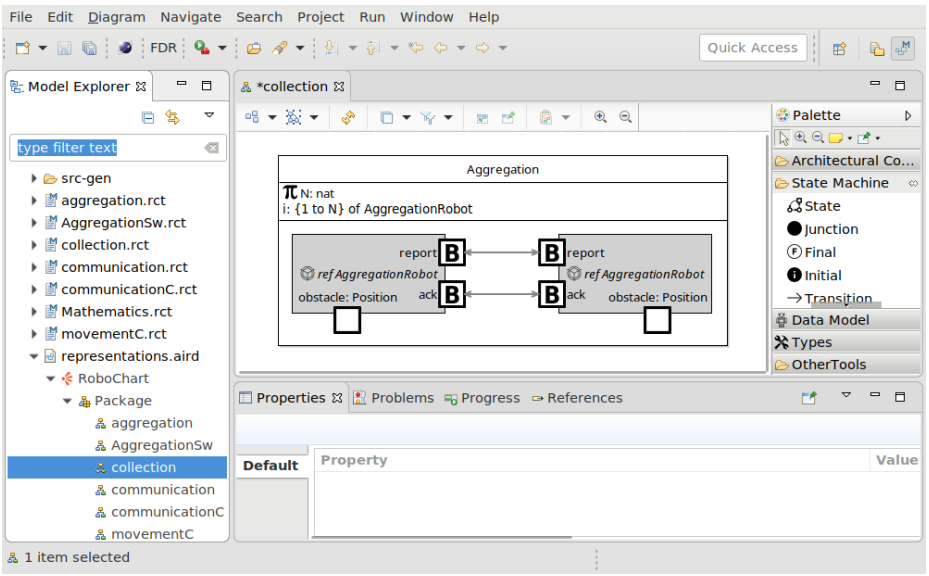

Fig. 7: Graphical editor for RoboChart.

5 Tool support

Tool support for RoboChart is implemented in RoboTool[5]. It provides a graphical editor for RoboChart models, a parser for the textual elements of the graphical notations (expressions, statements, transition labels, and so on), validators that check well-formedness conditions, and code generators. These tools are implemented as Eclipse[6] plugins using the Xtext[7] and Sirius[8] frameworks.

The RoboTool graphical editor is shown in Figure 7. It has been enriched with facilities for the creation of collections, and parsing of collection declarations and of the new triggers. The validator has been extended to check for well-formedness conditions for collections described in Section 2.

RoboTool provides three code generators: for the untimed semantics, the timed semantics, and a C++ simulation. The CSP files that are generated can be opened from RoboTool directly into FDR to verify properties of the models. A number of assertions of classical properties are also automatically generated for checking. The code generator for the untimed semantics has been updated to implement the semantics of collections of the previous section, and the update of the remaining code generators is part of our future work.

Using RoboTool and FDR, we have checked deadlock freedom for our example collection. For the empty collection, with $N = 0$, we have a deadlock: if there is no working robot, there is no observable behaviour. Similarly, for $N = 1$,

[5] `https://www.cs.york.ac.uk/circus/RoboCalc/robotool/`
[6] `www.eclipse.org`
[7] `www.eclipse.org/Xtext`
[8] `www.eclipse.org/sirius/`

we have a deadlock, because there is no other robot to accept a *report*. In the analyses for larger values of N, there is no deadlock, as expected[9]. Although we have been able to analyse this example, analysis of swarm applications requires theorem proving and our approach is well suited for that. Ongoing work allows automated proof of deadlock freedom using Isabelle/UTP.

6 Related work

UML and its various extensions are widely used for modelling in a number of different domains. Due to its generality, UML is an option for modelling robots and swarms. However, while a number of formalisations have been proposed for UML using techniques such as graph transformations [14], CSP [19, 2], as well as tailored semantic domains [1], these formalisations only cover subsets of UML.

RoboChart, on the other hand, is a small language, with a well defined process algebraic semantics suitable for verification using model checking and theorem proving. In addition, it caters for timed properties and now has specialised notation for collections of robots. Nordmann *et al.* [16] indicates that domain-specific languages (DSL) for robotics are growing in popularity, further motivating our choice of a small DSL over a more general notation such as UML.

Indeed, several robotic modelling notations have been proposed, but they mostly aim at code generation for execution or simulation. In RoboChart, we also focus on a formal semantics for verification and generation of sound simulations.

RobotML [3] is a UML-based notation for robotics that supports automatic code generation, but support for formal verification is not yet available. Schlegel *et al.*[21] propose the use of a UML-based framework for engineering robotic systems, but formal verification is also not supported. Work on GenoM [9] is one of the closest to ours. It supports verification of schedulability and deadlock checking. Unlike RoboChart, GenoM is an executable language (potentially including C code) with limited support for abstractions.

The approach in [7] uses model checking to identify optimal configurations, but verification of behavioural properties is not the goal. Orccad [5] supports modelling, simulation, programming, and verification of timed behavioural properties. Verification is supported by translating models into formal languages like in our work. However, Orccad differs from RoboChart in its limited support for graphical modelling and granularity of its modelling elements.

To the best of our knowledge, the notations for robotics in the literature do not support modelling and verification of collections of robots. UML-like notations provide support for modelling of components. A distinguishing feature of RoboChart, though, is that it allows specifying a swarm configuration by relating meta (as opposed to concrete) robot instances via placeholders. Therefore, a model identifies the relevant communication patterns and specifies them as templates. The number of robots that form a concrete configuration can be a parameter of the more abstract configuration specification. As far as we are

[9] `www.cs.york.ac.uk/circus/RoboCalc/case_studies/`

aware, all other existing graphical notations require that configurations are specified by relating concrete instances. This is the case, for instance, when we use Structure Diagrams to define a system configuration in terms of UML or SysML components. For every change in the number of components, the diagram needs to be revised. In RoboChart, we need to provide only a new value for a constant.

7 Conclusions

RoboChart supports modelling, verification, and simulation of robotic applications. Its concise design allows for the full specification of well-formedness conditions and semantics, as well as the implementation in the form of RoboTool. In this paper, we have described RoboChart support for explicit modelling of collections, and complex communication patterns within collections. The formal semantics of these facilities has been described and mechanised in RoboTool.

Currently, RoboChart is being extended with support for probabilistic modelling and verification, and a library of robotic platforms and common behaviours is under development. Furthermore, extensions to support modelling of the continuous aspects of the hardware and the environment are planned. All these extensions under development are useful also for modelling swarms.

Model checking is limited in the size of the models that it can handle. Our plan, especially for verification of swarms, is to explore compositional techniques for the efficient verification of CSP specifications [6], and semi-automatic verification using a CSP mechanisation in the theorem prover Isabelle [8].

The only form of communication available within a collection is perfect broadcast. We plan to provide: (1) a catalogue of types of communication media including mechanisms to model message loss and corruption; and (2) constructs and a library of models to support the specification of environments.

Acknowledgements The work mentioned here is supported by the EPSRC grants EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engineering, and by INES, grants CNPq/465614/2014-0 and FACEPE/APQ/0388-1.03/14.

References

1. 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.
2. J. Davies and C. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Formal Aspects of Computing*, 15(2-3):118–145, 2003.
3. 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.
4. C. Dixon, A. F. T. Winfield, M. Fisher, and C. Zeng. Towards temporal verification of swarm robotic systems. *Robotics and Autonomous Systems*, 60(11):1429–1441, 2012.

5. B. Espiau, K. Kapellos, and M. Jourdan. *Formal Verification in Robotics: Why and How?*, pages 225–236. Springer London, 1996.
6. M. S. Conserva Filho, M. V. M. Oliveira, A. C. A. Sampaio, and A. L. C. Cavalcanti. Compositional and local livelock analysis for CSP. *Information Processing Letters*, 2018.
7. F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 606–621. Springer-Verlag, 2009.
8. 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.
9. M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model checking real-time properties on the functional layer of autonomous robots. In K. Ogata, M. Lawford, and S. Liu, editors, *Formal Methods and Software Engineering*, pages 383–399. Springer, 2016.
10. 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.
11. Object Management Group. OMG: UML Profile for MARTE, v1.0, November 2009. OMG Document Number: formal/(2009-11-02).
12. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.
13. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. 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.
15. A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3869–3876, 2017.
16. A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede. A survey on domain-specific modeling and languages in robotics. *Journal of Software Engineering for Robotics*, 7(1):75–99, 2016.
17. OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012.
18. E. Posse and J. Dingel. An executable formal semantics for UML-RT. *Software and Systems Modeling*, pages 1–39, 2014.
19. H. Rasch and H. Wehrheim. Checking consistency in UML diagrams: 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.
20. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
21. 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.
22. University of York. *RoboChart Reference Manual*. `www.cs.york.ac.uk/circus/RoboCalc/robotool/`.