

# From Formalised State Machines to Implementations of Robotic Controllers

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

**Abstract** Controllers for autonomous robotic systems can be specified using state machines. However, these are typically developed in an *ad hoc* manner without formal semantics, which makes it difficult to analyse the controller. Simulations are often used during the development, but a rigorous connection between the designed controller and the implementation is often overlooked. This paper presents a state-machine based notation, RoboChart, together with a tool to automatically create code from the state machines, establishing a rigorous connection between specification and implementation. In RoboChart, a robot's controller is specified either graphically or using a textual description language. The controller code for simulation is automatically generated through a direct mapping from the specification. We demonstrate our approach using two case studies (self-organized aggregation and swarm taxis) in swarm robotics. The simulations are presented using two different simulators showing the general applicability of our approach.

---

W. Li (✉) · J. Timmis  
Department of Electronics, University of York, York, UK  
e-mail: wei.li@york.ac.uk

J. Timmis  
e-mail: jon.timmis@york.ac.uk

A. Miyazawa · P. Ribeiro · A. Cavalcanti · J. Woodcock  
Department of Computer Science, University of York, York, UK  
e-mail: alvaro.miyazawa@york.ac.uk

P. Ribeiro  
e-mail: pedro.ribeiro@york.ac.uk

A. Cavalcanti  
e-mail: ana.cavalcanti@york.ac.uk

J. Woodcock  
e-mail: jim.woodcock@york.ac.uk

## 1 Introduction

Safety is a major concern for autonomous robots, and the ability to provide evidence that a robotic system is safe can be demanding. Formal verification is the process of checking whether a design satisfies some requirements (properties) or that an implementation conforms to a design, and it has been used to verify a variety of robotic systems such as service robots [23] and swarming robots [20, 24].

Swarm robotics investigates how multiple robots, each with limited ability, communicate, coordinate and self-organize to accomplish certain tasks. Swarm robotics has potential in a wide range of real-world applications such as search and rescue, object transportation and environmental monitoring [4]. While using a number of simple robots to collectively perform complex tasks is desirable, designing individual controllers to guarantee the emergence of certain swarm behaviour is challenging. If swarm robotic systems are to transfer from lab-based experiments to real applications, especially those that are safety-critical, the verification of the individual controllers as well as their resulting emergent swarm behaviours needs to be conducted in a rigorous way.

Typically, the implementation of a robotic control system is conducted without establishing a strong connection between the controller code and the high-level design specifications. Here we explore the usage of a state-machine based notation, RoboChart [17], for designing robotic controllers. RoboChart has a formal semantics that allows for verification. In this paper, we extend RoboChart to support automatic code generation from the designed controllers to simulations.

Finite state machines are often adopted to design robot controllers in swarm robotics [2, 5, 10, 11, 14]. A commonly used state-machine notation is that of UML [1]. RoboChart takes inspiration from UML, and provides facilities to model timed and probabilistic systems, composed of one or more controllers.

Formal verification has been investigated in the design of controllers in swarm robotic systems [3, 7, 12, 20, 24]. In [7, 24], the authors used a temporal logic to formally specify and verify the emergent behaviour of a swarm robotic system performing aggregation. In [12], the authors used PRISM, a model checker for probabilistic automata, to formally verify the global behaviour of a foraging case scenario through exhausting all possible swarm behaviours. The analysis results were compared with those reported in [14], which used the test-driven simulation and showed a good correspondence. In these works, finite state-machine controllers were described using natural language, and there was no direct mapping from the high-level specification to low-level controller code.

In [15], the authors applied supervisory control theory to control a swarm of robots. Their approach supported automatic code generation. The controllers were specified using standard finite state machines, without any of the extra facilities for architectural modelling available, for example, in UML.

Various researchers have also explored the use of model-driven approaches to develop the high-level control of robots [6, 8, 21, 22]. The architecture analysis and design language (AADL) is a unifying component-based framework for mod-

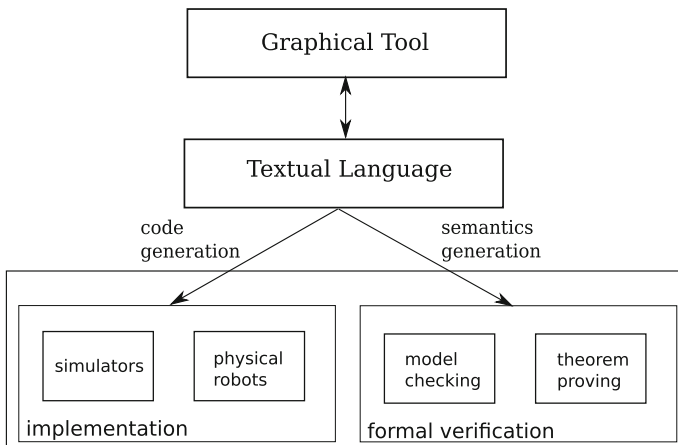
elling software systems with a particular focus on embedded real-time systems [8]. RoboChart could in principle be integrated into the controller component in AADL. In [22], a language was developed to program self-assembling robots. They proposed a role-based language that allowed the programmer to define the behavioural roles of each component independently from the concrete physical structure of the robots. However, in these works, the controllers of robots (e.g. state machine) were not formally specified, which makes it difficult to reason about robotic systems.

The main contribution of this paper is to reduce the gap between high-level specification and implementation of robotic controllers.

This paper is organized as follows. Section 2 briefly introduces RoboChart. This includes the elements of RoboChart and the approach to automatic code generation for simulation and deployment. Section 3 presents two case studies (self-organized aggregation [11] and swarm taxis [2]) in swarm robotics. The simulations using the automatically generated C++ code are presented. Section 4 concludes the paper and presents future work.

## 2 RoboChart

Figure 1 shows the RoboChart framework to combine formalised state machines and automatic implementation of robotic controllers. Once the controller is developed, code is generated automatically to be used in different simulation platforms or physical robots. Formal semantics are also automatically generated for verification. Details of the formal semantics of RoboChart can be found in [17]. In the following section, we focus on the automatic code generation for simulation and deployment.



**Fig. 1** The RoboChart framework for combining formalised state machines and implementation of robotic controllers

## 2.1 Elements of RoboChart

Central to RoboChart is a state-machine notation. RoboChart machines include states and their *entry*, *during* and *exit* operations (actions), as well as transitions possibly triggered by events. The entry operation is executed when the robot enters a state, and followed by the execution of the during operation. When a transition is triggered, the exit operation of the source state is executed. If an action is associated with the transition, it is also executed before the state machine enters the target state.

Operations and events of a state machine are described in an *interface*. A state machine can *requires* an interface. An operation can either be described without implementation or implemented by the user in a state-machine style. An operation can include a *precondition* and a *postcondition*.

Variables can be defined in a state machine, an interface or an operation. Different data types (primitive or composite) can be defined. When the behaviour is complex, multiple (potentially interacting) state machines can be used.

In addition to state machines, RoboChart also includes elements to organize specifications such as modules and robotic platforms [17]. A module defines a system, including a robotic platform and associated controllers. Each controller can be specified by one or more state machines.

RoboChart also includes time constraints. A *clock* can be defined inside a state machine to record the instant in time #T in which a transition is triggered. For example, the primitive `since(T)` yields the time elapsed since the most recent time instant #T. If `since(T)` is used as a condition (guard) on a transition with no events, then the transition will be taken immediately once the guard is true. Unless time is specified, we assume an operation takes no (or a significantly small) time.

For full details of RoboChart, refer to [17].

## 2.2 Simulation and Deployment

In RoboChart, the robot's controller is specified either graphically or using a textual description language. The automatically generated controller code can be imported into a wide variety of simulation platforms.

We adopt the model-view-controller (MVC) pattern in the design of simulations, where, the terms model and controller are used in a different way from that adopted in RoboChart. Figure 2 maps the RoboChart constructs to an MVC architecture. The model (M) component contains a simulation of the environment and of the RoboChart controller. We can generate a simulation of the RoboChart controller, potentially together with a simulation of the environment.<sup>1</sup> The controller component (C) implements the robotic platform, which corresponds to a particular robot

---

<sup>1</sup>The specification of environment is still under development. Currently the environmental stimuli are manually defined in the simulation.

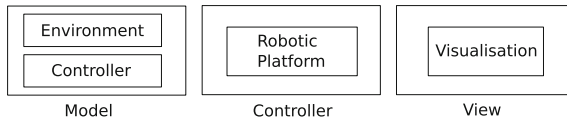


Fig. 2 RoboChart simulations pattern

RoboChart	State machine class	RoboChart	Interface class
states	attribute of enumerated type	events	attribute of enumerated type
clocks	attribute of timer class	variables	attribute
interfaces	inherit interface class	operations	methods

Fig. 3 RoboChart state machine and interface classes

in a simulation. Finally, the view component (V) defines the visualisation of the simulation.

We now describe how the controllers defined in RoboChart can be mapped into an executable language, specifically C++. Other object-oriented languages can be considered in a similar way, but are currently outside the scope of our work. The simulation of a controller is the simulation of its state machine(s). Each machine is implemented by a class. If the machine requires an interface, that interface is also implemented by a class, which is inherited by the state machine.

Figure 3 defines how constructs of a state machine and interface are mapped to elements of a class. The variables and events defined in an interface are generated as attributes of the class. The operations (*entry*, *during* and *exit*) that the robot executes in a state give rise to methods. We note that, even if an operation is specified in a state-machine style, it is generated as a method. To update the state machine, some other methods such as *MakeTransition* are also generated.

If a *clock* is defined in a state machine, a timer class is generated. It has a attribute *counter*, indicating the elapsed time, and methods such as *StartTimer* and *ResetTimer*. The state machine includes an object of the timer class as an attribute. The timer is used as a service of the state machine, which means the state machine can assess the counter. The state of the robot is updated in a cyclic manner, with the length of the cycle linked to the length of time required to capture events. The counter of the timer is updated in each control cycle.

A primitive data type is directly mapped into one in C++. For example, the type *real* corresponds to *double* in the code. A composite type is generated as a pre-defined class. For example, *vector2d* corresponds to a 2D vector class. The data-type system in RoboChart as well as its mapping are still under development.

### 3 Modelling Robotic Controllers Using RoboChart

To demonstrate our approach, we investigate two case studies on canonical problems in swarm robotics: aggregation [11] and swarm taxis (flocking towards a beacon) [2]. In these case studies, the robots are homogeneous. The controller of each robot is defined by a single state machine, and it is executed in the e-puck [18], which is a differential wheeled robot. It has an inter-wheel distance of 5.1 cm. The maximum speed for the left and right wheels of the e-puck is 12.8 cm/s, forward or backward.

#### 3.1 Case Study One: Aggregation

##### 3.1.1 Aggregation Behaviour

In this behaviour, each robot is equipped with a line-of-sight sensor that detects the type of item in front of it. The range of this sensor is unlimited in simulation. It gives a reading of  $I = 1$  if there is a robot in the line of sight, and  $I = 0$  otherwise. The environment is free of obstacles. The objective for the robots is to aggregate into a single compact cluster as fast as possible.

Each robot implements a reactive behaviour by mapping the sensor input ( $I$ ) onto the outputs, that is, a pair of predefined speeds for the left and right wheels,  $(v_{\ell I}, v_{r I})$ ,  $v_{\ell I}, v_{r I} \in [-1, 1]$ , where  $-1$  and  $1$  correspond to the wheel rotating backwards and forwards respectively with maximum speed.

The parameters of the aggregation controller were found by performing a grid search over the space of possible combinations [11]. The controller exhibiting the highest performance was:

$$\mathbf{p} = (v_{\ell 0}, v_{r 0}, v_{\ell 1}, v_{r 1}) = (-0.7, -1.0, 1.0, -1.0). \quad (1)$$

When  $I = 0$ , a robot moves backwards along a clockwise circular trajectory with a linear speed of  $-10.88$  cm/s and an angular speed of  $-0.75$  rad/s. When  $I = 1$ , a robot rotates clockwise on the spot with a linear speed of  $0$  and the maximum angular speed of  $-5.02$  rad/s.

##### 3.1.2 Modelling the Aggregation Controller in RoboChart

Figure 4 shows the diagram of the aggregation controller modelled in RoboChart. An interface, `AggregationIface`, declares the variables, operations and events. The state machine (`AggregationFSM`) requires `AggregationIface`. The state machine has an initial node, `i`, pointing to the initial state. The aggregation controller includes two states (`S1` and `S2`), two events (`seeWall` and `seeRobot`, which correspond to  $I = 0$  and  $I = 1$  respectively), and two operations (`MoveClockwise` and `RotateClockwise`).

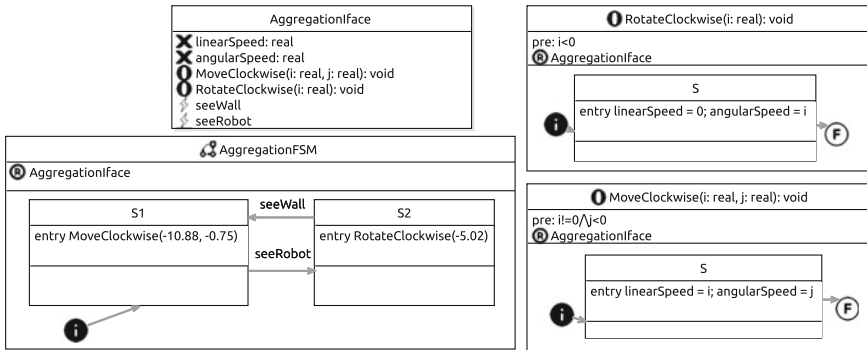


Fig. 4 Diagram of the aggregation controller modelled in RoboChart

```

stm AggregationFSM {
  requires AggregationIFace
  initial I
  state S1 {
    entry MoveClockwise(-10.88, -0.75)
  }
  state S2 {
    entry RotateClockwise(-5.02)
  }
  transition T1 {
    from I to S1
  }
  transition T2 {
    from S2 to S1
    trigger seeWall
  }
  transition T3 {
    from S1 to S2
    trigger seeRobot
  }
}

operation MoveClockwise(i: real, j:real) : void {
  precondition i != 0 /\ j < 0
  requires AggregationIFace
  initial I
  final F
  state S {
    entry linearSpeed = i; angularSpeed = j
  }
  transition T1 {
    from I
    to S
  }
  transition T2 {
    from S
    to F
  }
}
    
```

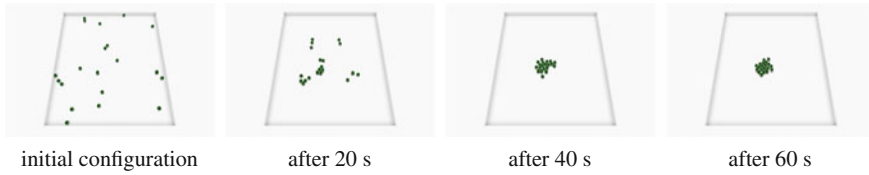
(a) aggregation controller

(b) *MoveClockwise* operation

Fig. 5 Textual description of the aggregation controller and an operation in RoboChart

These operations are implemented in a state machine style with only an initial state S and final state F. Different from the *AggregationFSM* state machine, both operations have a final state. An operation can include precondition that must be satisfied by the caller to guarantee that the functionality of this operation is realised as specified. For example, in the *MoveClockwise* operation, the precondition requires that its first argument, an angular speed, is negative, and the second, the linear speed, is not zero. In the generated C++ code, this is realized using the *assert* function. A textual description of the *AggregationFSM* state machine and the *MoveClockwise* operation is shown in Fig. 5.

In the generated C++ code, two classes (*AggregationInterface* and *AggregationFSM*) are generated. The class *AggregationInterface* includes the attributes of two double variables (*linearSpeed* and *angularSpeed*), two methods (*MoveClockwise* and *RotateClockwise*) and two boolean events (*seeWall* and *seeRobot*). The operations are generated as virtual functions that can be overridden if necessary. The



**Fig. 6** Snapshots of the aggregation behaviour of 20 robots in simulation, using the automatically generated controller code from the RoboChart model

`Aggregationface` class is inherited by the state machine `AggregationFSM` class. It has the attributes of states `S1` and `S2`, and other methods that are used to run the state machine. The generated C++ controller code can be found in the online supplementary materials [13].

### 3.1.3 Simulating the Aggregation Behaviour

The automatically generated code of the aggregation controller is tested in Enki [16], which has a built-in model of the e-puck robot. Enki is a 2D simulator and it can simulate swarms of robots a hundred times faster than real time. The speed of the left and right wheels of the e-puck can be set separately. The line-of-sight sensor in Enki is simulated by casting a ray from the e-puck's front and checking the first item with which it intersects (if any). The arena size is  $250 \times 250 \text{ cm}^2$ , and the initial position and orientation of the robots are randomly distributed. The length of the control step is set to 0.1 s, and the physics is updated every 0.01 s.

We performed 10 simulation trials with 20 robots, and in each trial the robots can aggregate into a single cluster. Figure 6 shows snapshots from a simulation trial using the automatically generated controller code from the model in RoboChart.

## 3.2 Case Study Two: Swarm Taxis

### 3.2.1 Swarm Taxis Behaviour

In the swarm taxis behaviour, the robots move towards a beacon while maintaining a coherent group. Each robot has three states: `Forward`, `Coherence` and `Avoidance`. The initial state is `Forward`. If the robot is in the `Forward` state for a certain number of time units without detecting any robots within avoidance radius, it enters the `Coherence` state. In this state, the robot turns towards the estimated center of the nearby robots. If the robot detects any robot within the avoidance radius while it is in the `Forward` state, it enters the `Avoidance` state. In this state, the robot turns away from the estimated center of the robots being avoided.



The robot can be illuminated by a beacon in the environment or shadowed by other robots (unilluminated). The avoidance radius when the robot is illuminated is larger. The avoidance radius is updated while the robot is in the **Forward** state. It is this mechanism that leads to the emergent swarm taxis behaviour [2].

### 3.2.2 Modelling the Swarm Taxis Controller in RoboChart

Figure 7 shows the diagram of the swarm taxis controller in RoboChart. The full model can be found in the online supplementary materials [13]. The interface **SwarmTaxisInterface** defines the variables, operations and an event. A clock is defined inside the controller **SwarmTaxisFSM**. The initial state of the controller is **Forward**, where a timer **T** is started immediately. The timer records the time the robot stays in the state **Forward**. In RoboChart, an expression marked in square brackets (such as *reached == true* or *since(T) < 25* in Fig. 7) is a guard for the transition. If there is no event associated with a transition, satisfaction of the condition will trigger the transition immediately. For example, once 25 time units have elapsed since the robot is in the **Forward** state, a transition from the **Forward** state to the **Coherence** state is triggered.

In the **Forward** state, the robot updates its avoidance radius through the operation *UpdateAvoidanceRadius*. The actual avoidance radius is set based on the boolean variable *illuminated* resulting from the operation *CheckIlluminationStatus*. If the robot is illuminated, the avoidance radius is set to 0.2; otherwise it is set to 0.1. As a consequence of this choice, the robots that have longer avoidance radius (are illuminated) tend to move towards the beacon and thus give rise to the beacon taxis behaviour of the whole swarm. Note that although we have declared the operation *CheckIlluminatedStatus*, we have chosen not to specify it in the RoboChart model, since it relies on the usage of the robot’s sensors, which is platform dependent. If the robot detects any other robots nearby within the avoidance radius, it enters the **Avoidance** state, where the robot calculates the desirable turning degree

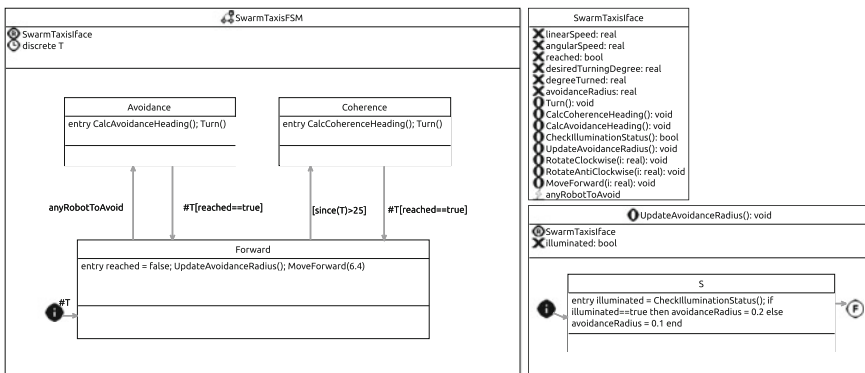
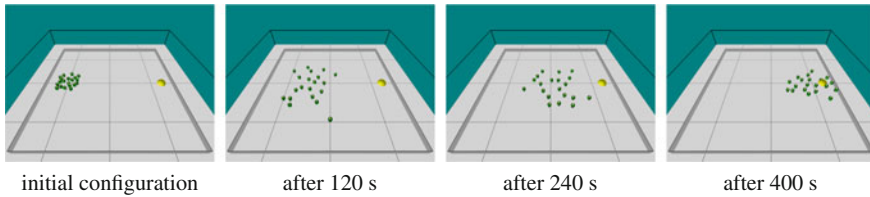


Fig. 7 Model of the swarm taxis controller in RoboChart



**Fig. 8** Snapshots of the swarm taxis behaviour in simulation, using the automatically generated controller code from the RoboChart model. There are 20 robots (green) and one beacon (yellow)

(*desiredTurningDegree*) using the operation *CalcAvoidanceHeading* and then executes the operation *Turn*. In the operation *Turn*, the boolean variable *reached* is updated to indicate whether the robot has turned the desirable degree. Once the desirable turning degree has been achieved, the variable *reached* is set to *true*, which triggers the transition from *Avoidance* to *Forward*. Every time a transition is triggered, the timer *T* is started. Similar operations occur in the transition from *Coherence* to *Forward*.

For a full description of the model, refer to [13].

### 3.2.3 Simulating the Swarm Taxis Behaviour

The swarm taxis behaviour is simulated in ARGoS [19], which also has a built-in model of the e-puck. It is a 3D simulator. The simulated space can be divided into several sub-spaces that run different physics engines in parallel. The arena size is  $400 \times 400 \text{ cm}^2$ . There is one beacon located in the right of the arena, and the robots are randomly initialized in the left region of the arena. Each robot is equipped with light sensors (to detect the beacon) and range-and-bearing sensors (to detect other robots nearby). The length of control step is set to 0.1 s. Note that in the model shown in Fig. 7 we did not attempt to optimize the value of each parameter (such as the time threshold and avoidance radius).

We performed 10 simulation trials with 20 robots, and in each trial the robots can successfully move towards the beacon while maintaining a coherent group. Figure 8 shows snapshots from a simulation trial, using the automatically generated controller code in RoboChart. A video showing the simulation of the two case studies and the automatically generated C++ code of the controllers can be found in the online supplementary materials [13].

## 4 Conclusion

In this paper, we have presented a state-machine based framework RoboChart for modelling the controllers of autonomous robots, combined with the automatic gen-

eration of C++ code. We believe that this is the first framework that allows for both automatic code generation for robotic simulation, deployment and formal verification. The applicability of our approach has been demonstrated through modelling two case studies (self-organized aggregation and swarm taxis) in swarm robotics. The automatically generated code of the robot's controller was run in two different simulators, which again, demonstrates the flexibility of our approach.

Our vision is to significantly reduce the gap between the high-level reasoning and low-level implementation through the use of formal methods and automatic code generation. The work presented can be seen as a first step towards the goal of verifying emergent behaviour, which is a potential application of our work to be investigated in the future. Our current focus is, however, to enrich the state machine specification of RoboChart by adding time and probability constructs, so that the framework can be applied to model a wide variety of robotic control systems. The formal semantics of RoboChart will also be enriched to make the verification feasible. In RoboChart, we focus on modelling the controller of a single robot, but we are investigating the possibility of using RoboChart models to simulate and analyse robotic swarms.

Currently, the generated controller code is a direct mapping from the elements in RoboChart to simulation. In the future, soundness of the simulation will be established by verifying the code generator. This can be realized using various software engineering techniques. In particular, we envisage that the CSP model generated from the RoboChart specification is a basis for establishing the correctness of the generated code using refinement. Practical verification can be carried out using a model checker like FDR (which also provides a facility to animate the model, and thus perform some validation), or using a theorem prover.

In this paper, we only automatically generate the code of controllers, however the simulation configurations (e.g. length of control step) in the case studies are manually defined. We intend to define simulations in an extended notation, from which the simulation configurations can also be specified. The simulation notation will be independent of specific programming languages such as C++ and Java, and of specific robotic platforms.

Possible avenue for future work is the integration of RoboChart in other tools [3, 9]. For example, in [9], an automatic design method was used to tune the free parameters of a predefined parametric architecture (e.g. probabilistic state machine) for the individual robot controller of a swarm. In this case, the controller architecture can be modelled in RoboChart, so that the obtained solution can be formally verified. In [3], a property-driven approach was proposed to design the controller of swarming robots. The designed controller can also be modelled in RoboChart to support both formal verification and code generation.

Finally, we intend to model the environmental stimuli and generate code for physical robots.

**Acknowledgements** The authors would like to acknowledge the support from EPSRC grant EP/M025756/1.

## References

1. Bergenti, F., Poggi, A.: Exploiting uml in the design of multi-agent systems. In: Omicini, A., Tolksdorf, R., Zambonelli, F. (eds.) *Engineering Societies in the Agents World: First International Workshop*, pp. 106–113. Springer, Berlin, Germany (2000)
2. Bjerknæs, J.D., Winfield, A.F.T.: On fault tolerance and scalability of swarm robotic systems. In: Martinoli, A., Mondada, F., Correll, N., Mermoud, G., Egerstedt, M., Hsieh, A.M., Parker, E.L., Støy, K. (eds.) *Distributed Autonomous Robotic Systems: The 10th International Symposium*, pp. 431–444. Springer, Berlin, Germany (2013)
3. Brambilla, M., Pinciroli, C., Birattari, M., Dorigo, M.: Property-driven design for swarm robotics. In: *Proceedings of 2012 International Conference on Autonomous Agents and Multiagent Systems*, pp. 139–146. IFAAMS, Richland, SC, USA (2012)
4. Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M.: Swarm robotics: a review from the swarm engineering perspective. *Swarm Intell.* **7**(1), 1–41 (2013)
5. Chen, J., Gauci, M., Li, W., Kolling, A., Groß, R.: Occlusion-based cooperative transport with a swarm of miniature mobile robots. *IEEE Trans. Robot.* **31**(2), 307–321 (2015)
6. Dhoubi, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Noda, I., Ando, N., Brugalí, D., Kuffner, J.J. (eds.) *Simulation, Modeling, and Programming for Autonomous Robots*, pp. 149–160. Springer, Berlin, Germany (2012)
7. Dixon, C., Winfield, A.F.T., Fisher, M., Zeng, C.: Towards temporal verification of swarm robotic systems. *Robot. Auton. Syst.* **60**(11), 1429–1441 (2012)
8. Feiler, P.H., Gluch, D.P.: *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley, Boston (2012)
9. Francesca, G., Brambilla, M., Brutschy, A., Garattoni, L., Miletitch, R., Podevijn, G., Reina, A., Soleymani, T., Salvaro, M., Pinciroli, C., Mascia, F., Trianni, V., Birattari, M.: AutoMoDe-Chocolate: a method for the automatic design of robot swarms that outperforms humans. *Swarm Intell.* **9**(2–3), 125–152 (2015)
10. Gauci, M., Chen, J., Li, W., Dodd, T.J., Groß, R.: Clustering objects with robots that do not compute. In: *Proceedings of 2014 International Conference on Autonomous Agents and Multiagent Systems*, pp. 421–428. IFAAMS, Richland, SC, USA (2014)
11. Gauci, M., Chen, J., Li, W., Dodd, T.J., Groß, R.: Self-organized aggregation without computation. *Int. J. Robot. Res.* **33**(8), 1145–1161 (2014)
12. Konur, S., Dixon, C., Fisher, M.: Analysing robot swarm behaviour via probabilistic model checking. *Robot. Auton. Syst.* **60**(2), 199–213 (2012)
13. Li, W., Miyazawa, A., Ribeiro, P., Cavalcanti, A., Woodcock, J., Timmis, J.: Online supplementary material (2016). <http://www.york.ac.uk/robot-lab/dars2016/>
14. Liu, W., Winfield, A.F.T.: Modeling and optimization of adaptive foraging in swarm robotic systems. *Int. J. Robot. Res.* **29**(14), 1743–1760 (2010)
15. Lopes, Y.K., Trenkwalder, S.M., Leal, A.B., Dodd, T.J., Groß, R.: Supervisory control theory applied to swarm robotics. *Swarm Intell.* **10**(1), 65–97 (2016)
16. Magnenat, S., Waibel, M., Beyeler, A.: Enki: the fast 2D robot simulator (2011). <http://home.gna.org/enki/>
17. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J., Woodcock, J.C.P.: RoboChart: a state-machine notation for modelling and verification of mobile and autonomous robots. Technical report, University of York, Department of Computer Science, York, UK (2016). [www.cs.york.ac.uk/circus/publications/techreports/reports/MRLCTW16.pdf](http://www.cs.york.ac.uk/circus/publications/techreports/reports/MRLCTW16.pdf)
18. Mondada, F., et al.: The e-puck, a robot designed for education in engineering. In: *Proceeding of the 9th Conference on Autonomous Robot Systems and Competitions*, vol. 1, pp. 59–65. IPCB: Instituto Politécnico de Castelo Branco (2009)
19. Pinciroli, C., Trianni, V., O’Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Birattari, M., Gambardella, L.M., Dorigo, M.: ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intell.* **6**(4), 271–295 (2012)

20. Rouff, C.A., Hinchey, M.G., Pena, J., Ruiz-Cortes, A.: Using formal methods and agent-oriented software engineering for modeling NASA swarm-based systems. In: 2007 IEEE Swarm Intelligence Symposium, pp. 348–355. IEEE, Honolulu, Hawaii (2007)
21. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: from code-driven to model-driven designs. In: Proceedings of the 14th International Conference on Advanced Robotics, pp. 1–8. IEEE, Munich, Germany (2009)
22. Schultz, U.P., Christensen, D.J., Stoy, K.: A domain-specific language for programming self-reconfigurable robots. In: Proceedings of the 2007 Workshop on Automatic Program Generation for Embedded Systems, pp. 28–36. ACM, Salzburg, Austria (2007)
23. Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, K.L., Dautenhahn, K., Saez-Pons, J.: Toward reliable autonomous robotic assistants through formal verification: a case study. *IEEE Trans. Hum. Mach. Syst.* **46**(2), 186–196 (2016)
24. Winfield, A.F.T., Sa, J., Fernandez-Gago, M.C., Dixon, C., Fisher, M.: On formal specification of emergent behaviours in swarm robotic systems. *Int. J. Adv. Robot. Syst.* **2**(4), 363–370 (2005)