Dragonfly: a Tool for Simulating Self-Adaptive Drone Behaviours

Paulo Henrique Maia*, Lucas Vieira*, Matheus Chagas*, Yijun Yu†, Andrea Zisman†, and Bashar Nuseibeh†‡
*State University of Ceará, Fortaleza, CE, Brazil
†The Open University, Milton Keynes, United Kingdom
‡Lero - The Irish Software Research Centre, Limerick, Ireland

***Abstract*—Drone simulators can provide an abstraction of different applications of drones and facilitate reasoning about distinct situations, in order to evaluate the effectiveness of these applications. In this paper we describe *Dragonfly*, a simulator of the behaviours of individual and collection of drones in various environments, involving random contextual variables and different environmental settings. *Dragonfly* supports the use of several drones in applications and evaluates the satisfaction of requirements under normal and exceptional situations. It simulates adaptive behaviours of drones due to exceptional situations. The adaption of drones is based on the use of wrappers implemented using aspect-oriented programming.**

***Index Terms*—Drone, Adaptation, Aspects**

I. Introduction

The use of drones to support different types of applications ranging from search-and-rescue, to goods delivery and surveillance tasks has become a reality [1], [2]. However, it is possible to encounter uncertainties and exceptional situations, not initially predicted, during the use of drone-based applications. Drones are designed to satisfy pre-defined requirements, following pre-defined specifications, and are not necessarily intended to change their behaviour during their execution in order to support requirements of specific applications, or even to support new situations that may appear during the execution of these applications.

Many applications, including drone-based applications, are developed by the composition of independently created components into a single application. These applications allow the execution of certain functionalities that cannot be achieved by individual participating components on their own. Therefore, it is necessary to support emergent behaviours that appear due to the combination of existing components into new larger applications, or even due to new requirements or context changes [3]. The participating components are not necessarily intended to change their behaviour during execution, nor to support some requirements of the new applications.

Self-adaptive approaches have been proposed to support uncertainties and new requirements during the execution of applications [4]–[8]. However, the evaluation of the applications during runtime, in real situations, is not always an easy task. This is the case when dealing with risky situations. For example, in the case of drone-based applications, it is necessary to avoid the lost of drones and their respective delivery goods due to drones landing on water because of low battery levels; or to avoid drones flying over prohibited areas because of bad communication with the controller or lost of visual line-of-sight (VLOS) of the pilots.

In this paper we present *Dragonfly*, a simulator for self-adaptive drone behaviours in drone-based applications. *Dragonfly* is an extensible open-source tool that simulates both normal and adaptive behaviours of drones at runtime. The tool provides a graphical interface from which the user can create different environments composed of pre-defined entities and variables (e.g., drones, river, hospitals, communication antennas), and can monitor the status of the drones' resources and activities. The simulator also evaluates the implementation of self-adaptive behaviours of drones due to exceptional situations. The behavioural adaptation is based on the use of wrappers implemented in aspect-oriented programming. *Dragonfly* supports the comparison of the execution of drones in normal situations and when using the wrappers.

The remainder of the paper is structured as follows. Section II discusses related work. Section III gives an overview of the approach to support self-adaptation of drones, underpinning the simulator. Section IV presents the *Dragonfly* simulator, detailing its architecture and design decisions. Section V describes an example of using the simulator for a medical payload drone delivery application. Section VI concludes and presents future work.

II. Related Work

Simulation approaches evaluate the satisfaction of requirements by using a model similar to the real environment, at an acceptable level of abstraction, depending on the evaluation purpose. Some approaches have been proposed to support simulation of drones and drone-based applications. Evaluating functional requirements of drone surveillance systems requires the environment to be modelled as close to the real world as possible, whilst evaluating safety requirements of drone navigation systems, requires the environment to model unlikely events that could cause collision incidents.

Simulation of functional requirements: Microsoft AirSim[1] [9] can simulate the functionality of drones and their environments using a video game engine called Unreal Engine. The simulation allows users to immerse in an almost real-time experience in first-person view (FPV). This simulator has been used to validate algorithms that implement functionalities

[1]https://github.com/Microsoft/AirSim

such as Obstacle Detection in navigation [10]. These types of simulators are good for evaluating functional requirements of individual drones. However, they do not support the simulation of multiple drones or the simulation of drones with different behaviours.

Simulation of safety requirements: Although it is important to evaluate a swarm of drones to support safety requirements such as collision avoidance, the use of simulation can provide assurance of certain functionalities before physical real tests in the sky. Dronology approach [11] can simulate up to 100 drones for critical safety properties such as separation of distances. However, most of the properties supported by Dronology are concerned with the drones, while properties concerned with the environment (e.g., communication congestion, weather situations) are not taken into account.

Simulation of crosscutting requirements: In general, different quality requirements tend to crosscut multiple parts of large applications [12]. Aspect-oriented programming (AOP) [13] is one way to simulate drones to evaluate such crosscutting requirements, but not the only way. For example, dependency injection (DI) [14] may be applied when it is allowed to change the design of original system intrusively. AOP can be applied to both source and binary code, while DI is applicable typically when the programming languages support reflections [15].

In the following we present *Dragonfly* that complements existing simulators for drone-based applications. *Dragonfly* supports simulation of a large number of drones with distinct behaviours. It considers properties of the drones and of the environments. Moreover, *Dragonfly* is based on a non-intrusive approach to support adaptation of drones.

III. Our Approach

Figure 1 presents an overview of the proposed approach to identify and resolve exceptional situations in drone-based applications. As shown in the figure, the process is divided into two phases, namely: (i) *wrapper design and implementation* and (ii) *runtime adaptation*. The former occurs at design time, while the latter occurs at run time.

The *wrapper design and implementation* (phase 1) uses aspect-oriented programming (AOP) [13] to implement wrappers. The use of wrappers supports changes in the behaviour of a drone in order to satisfy global requirements of an application, while keeping the satisfaction of local requirements of the participating drones.

Wrappers specified in aspect-oriented programming support the introduction of changes into participating drones without requiring consent from the designer of the drones. The use of agent-oriented programming avoids the need to redesign a drone to satisfy emergent behaviours, as it is the case when using dependency injection techniques [14] or plugin-based approaches [16]. On the other hand, the use of aspect-oriented programming techniques can be risky, since changes introduced by the use of aspects may violate the original (local) requirements of the drones. Our work uses aspect-oriented techniques in a way that guarantees the original requirements of the drones under normal execution situations of the application.

In phase 1, the first step consists of *identifying exceptional situations* that may happen with a drone with respect to the application. This can be done by running brainstorming sessions with the development team of the application, or using a more systematic approach like scenarios to model both normal and exceptional behaviours of the drones. Scenarios can be used to represent both exceptional and normal situations of a system, and have been widely used for modelling *what-if* situations [17]. In general, scenarios assume the use of off-the-shelf software components, with predefined requirements and specifications.

The second step in phase 1 consists of *identifying monitorable context variables*. These variables can represent internal resources of drones (e.g. battery level and geographical positioning), and environmental features of the application (e.g. wind situation and temperature).

The identified exceptional situations indicate where changes in the system should occur. We refer to these locations of change as *joint points*. The third step in phase 1 consists of *analysis of exceptional situations as joint points*. The identified joint points are narrowed down to *point cuts* in the system (e.g., movement procedures of drones).

The fourth step in phase 1 aims at *planning adaptation actions as advices*. Aspect advices handle exceptions by either invoking existing functionalities of a drone or introducing new functionalities and situations to be executed. The approach uses precedence order among wrappers when more than one wrapper exists for the same exceptional situation.

Finally, after implementing wrappers as aspects, the fifth step consists of *deploying wrappers* in the drone's system so that the drone becomes a self-adaptive system during runtime.

In the *runtime adaptation* (phase 2), the system is *weaved* with wrappers and the respective functionalities are executed. This phase is based on the MAPE-K loop approach [18]. In this phase, monitors are used to verify contextual variables based on data from the environment (e.g., wind situation) or from sensors (e.g., Global Positioning Systems, accelerometer, battery level). The approach analyses exceptional situations by executing the *point cuts* defined in phase 1. When exceptional situations are satisfied, the respective wrappers for those situations are invoked and exceptional functionalities implemented by the aspects are executed.

IV. The *Dragonfly* tool

Dragonfly simulator is an open source and extensible Java tool available at GitHub [2]. It supports creating environments for simulating the behaviour of a set of drones in drone-based applications. In this section, we describe the interface and architecture of *Dragonfly*. We also describe how a simulation can be executed and how the tool can be extended.

[2]https://github.com/DragonflyDrone/Dragonfly

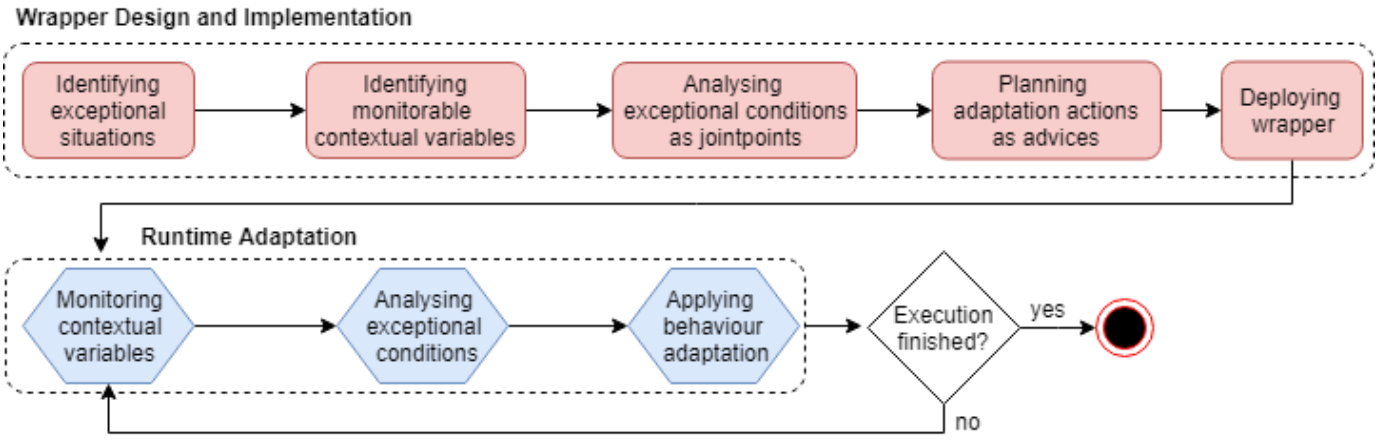

Fig. 1. Overview of the drone behaviour adaptation approach

A. Interface

Figure 2 shows the interface of the *Dragonfly* tool. As shown in the figure, the interface is divided into four main panels, namely: *graphical elements panel* (1), *drone flight environment panel* (2), *drone properties panel* (3), and *trace log panel* (4). In Figure 2 each panel is expanded for better visualisation.

The graphical element panel (1) provides a set of graphical elements that can be used to represent different environments of various applications. The current version provides elements to represent rivers, hospitals, communication antennas, and drones. Other elements can be easily added to the tool and is part of a future extension of *Dragonfly*.

The user can insert the graphical elements in the drone flight environment panel (2), which is used to create the environment in which the tool will simulate drones in an application. This panel consists of a grid layout in which each cell can contain one or more graphical elements.

The example in Figure 2 illustrates an environment with two hospitals: one on the far left and the other one on the far right, representing origin an destination of a flight. The pathway of this scenario consists of a river. The environment also has two communication antennas to simulate the emission of signals that may interfere with the journey of the drones. There are four drones flying in the environment: one drone (a) with its original specifications (i.e., without having any wrapper), and three drones ((b), (c), and (d)) weaved with wrappers representing different adaptive behaviours in case of exceptional situations during their respective journeys.

The drone property panel (3) allows setting of initial values of some of the resources of the participating drones. It also associates each participating drone with one or more wrappers representing behaviour adaptation functionalities, in case of exceptional situations. Examples of initial resource values are initial battery level and battery consumption rate.

The trace log panel (4) shows the current status and activities of each participating drone (identified by a number), during runtime simulation. An example of a current status is concerned with a drone's battery level, while examples of activities are concerned with take off, fly, move aside, and land.

B. Architecture

The architecture of *Dragonfly* is structured following the Model-View-Controller (MVC) architectural pattern [19]. As shown in Figure 3 the architecture is composed of three layers, namely: *View*, *Controller*, and *Model* layers.

The View layer contains classes representing the graphical elements and the environment itself. For each graphical element, it is necessary to create a view class (illustrated in Figure 3 by the generic class *EntityView*). A view class implements the interface *SelectableView*, which declares methods for inserting visual effects when an element is (or not) selected in the environment.

A particular case is of the graphical element representing a drone. In this case, the view class is abstracted and should be implemented by each specific type of drone available in the environment. For example, it is possible to have simple drones, which provide users with simple features such as taking off, moving, and landing; or more sophisticated ones that implement actions such as "return to home" or "follow me". In Figure 3 we assume one type of drone and represent this by class *DroneViewImpl*).

The class *CellView* contains shared functionalities for each cell of the grid of the drone flight environment. This class is associated with class *EnvironmentView*.

The Controller layer contains classes that associate events related to each graphical element in the environment with their correspondent model classes. Each element has its own controller class. The drone is the only element that has two controller classes: one for handling generated events when the drone is controlled by a user representing a pilot via a keyboard (*DroneKeyboardController*), and one for handling generated events when the drone is in automatic pilot mode (*DroneAutomaticController*).

The class *MainController* identifies all possible events in the environment and transfers the events to the correct controller

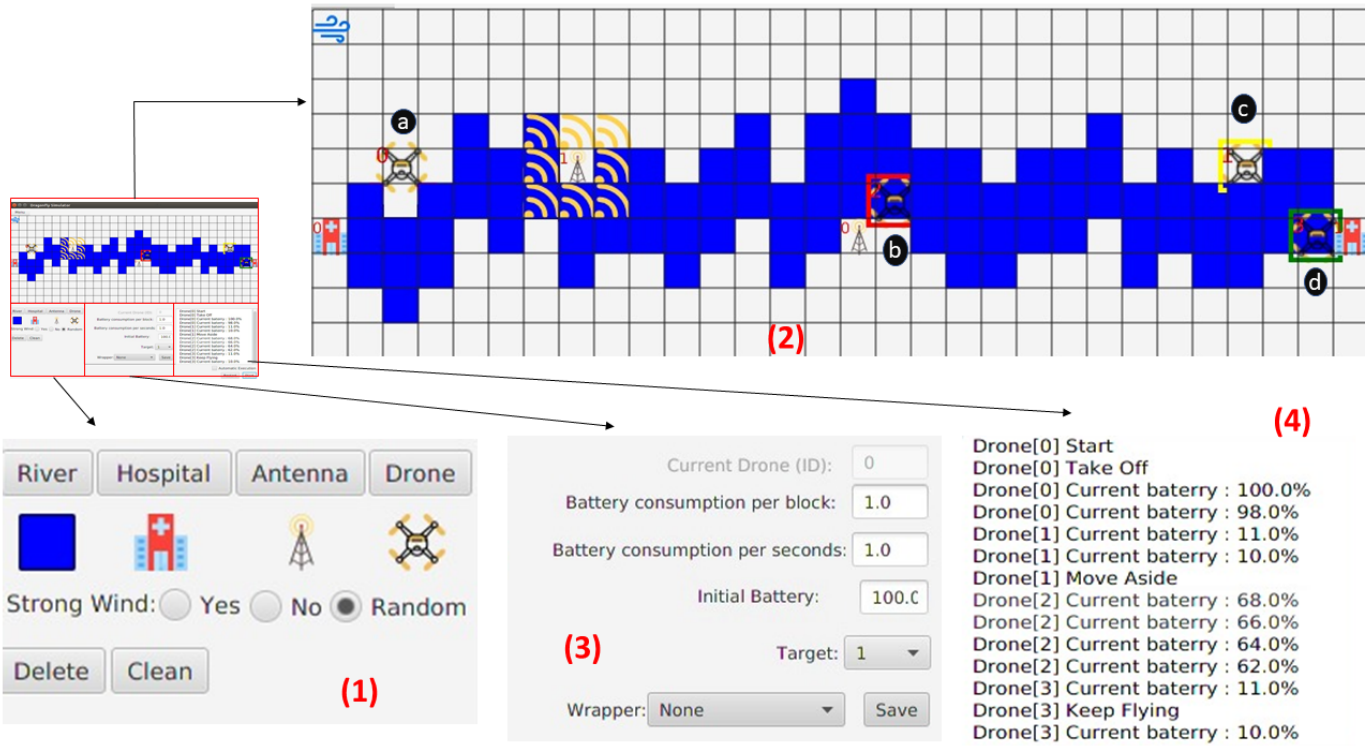

Fig. 2. Screenshot of the *Dragonfly* simulation tool

class that can handle them. In addition, the class *Logger-Controller* is responsible for printing the trace logs execution during the drone flight in the trace log panel.

The Model layer contains entity classes for each graphical element (illustrated by the generic class *Entity*). These classes implement the behaviour of elements. For instance, object *Drone* can execute several activities (e.g., take off, fly, and land), while object *Antenna* has only one associated activity (signal emission).

C. Execution of flight simulation

When using the simulator, the first step consists of constructing the environment of an application. In this case, the user inserts graphical elements by selecting an element from the graphical element panel (Figure 2 (1)) and choosing a specific position in the grid of the drone flight environment (Figure 2 (2)), where the element should be placed.

The user needs to configure the following properties for each drone inserted in the environment: battery consumption rate per block and per second, initial battery level, and target element (i.e., the place to where the drone will fly). In addition, he/she can associate a drone with an available wrapper. Afterwards, the user chooses the mode that the drone should fly. In the case of automatic pilot mode, when the simulation starts, the drone will execute the shortest path to reach the target destination. In the case of user pilot mode, the user will manoeuvre the drone by using the keyboard. The commands to manoeuvre a drone are available at the *Dragonfly's* GitHub repository.

The final step consists of starting the simulation by clicking the "Start" button, which triggers the execution of each inserted drone simultaneously. The currently implementation of the tool supports up to 400 drones flying at the same time, with and without wrappers. If the user has chosen to pilot the drone manually, the available commands to be executed are: turn on/off the drone, take off, move (up, right, down, and left), and land.

D. Tool extension flow

The current version of *Dragonfly* is extensible in terms of new graphical elements to represent other environment settings and in terms of new wrappers to represent new exceptional situations. For the creation of new graphical elements, it is necessary to create one class in each layer of the architecture for each new element, following the structure described in Section IV-B. It is also possible to associates an image with the new element in its correspondent view class. For the creation of new wrappers, it is possible to implement the behaviour represented by the wrapper in aspect-oriented programming to represent new exceptional situations, and associate drones with the new wrappers.

V. An Example

In order to illustrate the *Dragonfly* simulator, and its capability to simulate multiple drones under emergent environment

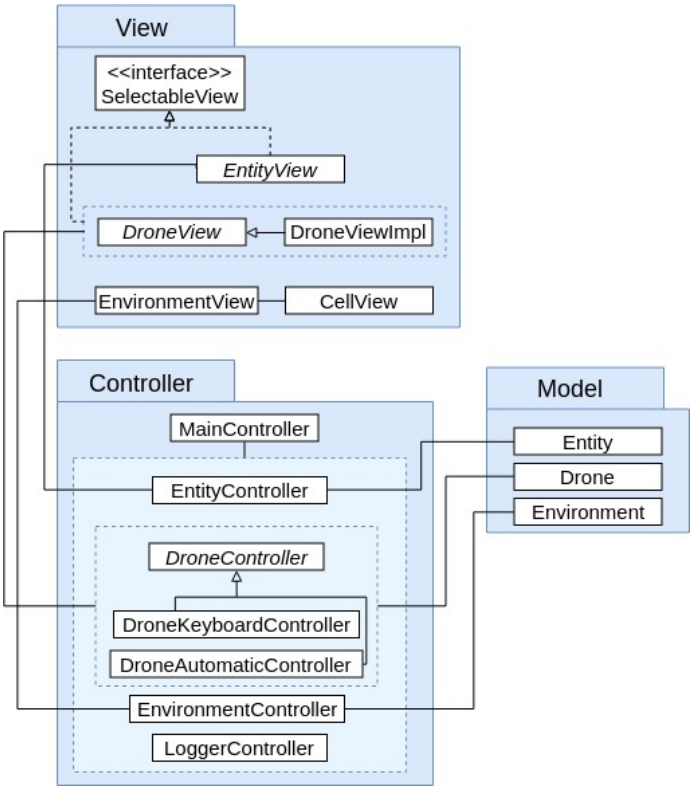

Fig. 3. Architecture of the *Dragonfly* tool

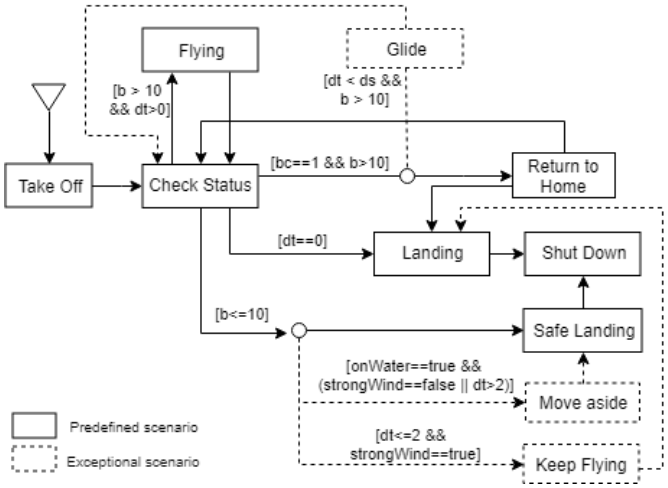

Fig. 4. Drone example scenario

situations, consider an example of a *medical payload drone delivery application* (organs or blood bags) between two hospitals.

This example has been chosen since it illustrates safety concerns involving multiple drones, with different battery levels, to deliver medical payload along a route across a river. The example also shows a complex environment, with different types of elements such as river and communication antennas, and different wind conditions.

Assume two global requirements for the example as:

- GR1: the drone must take a payload organ from the sender to the receiver hospital.
- GR2: in the case in which it is not possible to deliver the payload, the drone should not lose it (e.g., lost of the payload by landing on water).

Figure 4 depicts an overview of scenarios representing possible behaviours of a drone, and when the various behaviours are executed depending on sequence of actions and environment conditions. In Figure 4, the functionalities of normal specifications of a drone are represented by full rectangles and the transitions by full arrows, while the representation of exceptional situations are represented by dashed rectangles and dashed transitions.

In the scenario, the pilot can control the drone to take off (*Take Off*). During a flight, a drone periodically checks the status (*Check Status*) of its internal devices and sensors such as battery level and distance from the destination, represented by *b* and *dt*, respectively, in the guard conditions over the transitions. For the example, if the battery level is above the expected threshold of $\theta = 10\%$, and the drone is not yet in its destination, the pilot can keep manoeuvring the drone (*Flying*).

In a normal behaviour, when a drone arrives at its destination, the pilot sends a landing command to the drone (*Landing*), that is acknowledged when the drone lands on the ground, and after landing the drone shuts down (*Shut Down*). If the battery is below the expected threshold, the drone performs a safe landing (*Safe Landing*) in accordance to its specification, and than it shuts down.

The "return to home" functionality (*Return to Home*) is executed whenever a drone bypasses a bad connection area (for instance, near communication antennas) and loses its connection with the pilot. This is a common safety procedure currently found in most existing drones. As shown in Figure 4, *Return to Home* happens when the contextual variable *bc* (bad connection) is true. When the drone arrives at the depart point (contextual variable *ds* equals to 0), it lands normally.

Wrapper Design and Implementation: This phase initiates with the identification of exceptional situations for the medical payload drone delivery application, which are represented as dashed boxes in Figure 4. In the example, the exceptional situations are added in *interception points* represented by circles in the figure. The exceptional situations state that if the distance to the expected destination is less than 2km and the wind is strong (condition *strongWind==true*), then the pilot can keep the drone flying (*Keep Flying*), even in a low-level battery situation, and land afterwards. In this case, the drone is able to complete its overall goal of delivering the organ payload successfully (global requirements GR1, above). In the case in which the battery level is low, the drone is flying over the river (condition *onWater==true*), but the drone is more than 2km away or the wind is not strong, then the action is to move the drone aside (*Move Aside*) in order to land it safely on the ground and, therefore, satisfy global requirement GR2.

The exceptional situation for the automatic return to home

functionality consists of allowing the drone to *glide* while waiting for the connection to return and, therefore, maximizing the chances of the payload delivery. The exceptional behaviour is represented by *Glide*, which is performed when a bad connection is detected, the distance from the target hospital (contextual variable *dt*) is less than the distance from the source hospital, and the battery level is greater than 10%.

The next step consists of *identifying monitorable context variables* for analysing global requirements. In the payload organ delivery example, the monitorable context variables are: battery level, distance to destination, position of drone in relation to water, and wind condition.

The exceptional situations are mapped to corresponding concepts in the aspect-oriented paradigm [13]. The main goal is to *analyse exceptional conditions* and identify when they are triggered in order to associate them with joint points when mapping into aspect-oriented technique. The interception points in Figure 4 represent the point in which exceptional situations should be analysed and they are mapped to joint points.

For instance, considering the medical payload drone delivery example, the only interception point is the one between *Check Status* and *Safe Landing*, when it is checked whether the battery level has reached 10%. Subsequently, we have to define point cuts in which the code will be intercepted. For our example, we defined that method *safeLanding()* needs to be intercepted when it is called.

The code in Figure 5 shows an excerpt of the *DroneAspect*. We defined point cut *checkExceptionalConditions()* that intercepts the call of the drone's original *safeLanding()* method (when the battery level reaches 10%).

The code shows two kinds of syntactic advices: *before* and *around*. The *before* clause (Line 6) is executed when the drone is over the water, and either the distance to the target hospital is no less than 2km or the wind is not strong (Lines 7-9). In this case, the drone executes a *moveAside()* method (Line 10), which moves the drone to fly over the ground and to execute the original *safeLanding()* method.

The *around* clause (Line 13) is executed when the drone is no more than 2km away from the destination, is flying over the water, and the wind is strong (Lines 14-16). Unlike the previous *before()* clause, new behaviour *manoeuvre()* (Line 17) replaces the *safeLanding()* method, which is no longer executed. When the drone reached the destination it performs the original *Landing()* method.

For the medical payload drone delivery example, we have created three wrappers W1, W2, W3, specified as follows. Wrapper W1 addresses the safe landing procedure including exceptional situations *Move Aside* and *Keep Flying*. Wrapper W2 addresses the bad connection situation by forcing the drone to glide for a while instead of automatically returning to home. Wrapper W3 implements the above two exceptional conditions at the same time.

Runtime Adaptation: This phase executes the weaved wrappers (W1, W2, and W3) due to exceptional situations. The

```
1  public aspect DroneAspect {
2
3      pointcut checkExceptionalConditions():
4          call (void safeLanding());
5
6      void before(): checkExceptionalConditions() {
7          if (isOverWater()
8           && (getDistanceTargetHospital() >=2
9            || !isStrongWind()))
10             getDrone().moveAside();
11     }
12
13     void around(): checkExceptionalConditions() {
14         if (isOverWater()
15          && getDistanceTargetHospital() <=2
16          && isStrongWind())
17             getDrone().manoeuvre();
18     }
19 }
```

Fig. 5. Example of DroneAspect advice

steps of this phase are derived from the activities of the MAPE-K [18] control loop.

During their executions, the drones monitor (M) their respective contextual variables by either checking internal components and resource levels, or by receiving environmental data provided by sensors. The contextual information is analysed (A) and, when applicable, the respective wrappers intercept the execution of the system in the defined point cuts. In the example, this happens after the interception of *checkExceptionalConditions()* point cut. In the planning (P) and execution (E) activities of the MAPE-K loop, when an exceptional situation is satisfied, the wrapper applies the behaviour adaptation by executing the exceptional scenario according to the rules implemented in the advices. The process continues while the execution of the application is not finished.

VI. Conclusions and Future Work

In this paper we presented a simulator tool called *Dragonfly* to support the simulation of drones and drone-based applications in distinct environments. *Dragonfly* supports the simulation of up to 400 drones at the same time, and the evaluation of these drones during normal and exceptional situations. The simulator provides ways to specify distinct environments and different ways of dealing with exceptional situations represented as wrappers, expressed in aspect-oriented programming. The wrappers implement adapted behaviours of the drones and support runtime adaptation. An example of a medical payload drone delivery application is described to illustrate the use of the simulator.

Currently, we are extending the simulator to support the representation of new elements (entities and conditions), and implementing an editor to help developers identify exceptional situations and create wrappers.

Acknowledgements

The work is supported in part by Royal Society, EPSRC, SFI grant 13/RC/2094, and CAPES (Brazil) grants.

REFERENCES

[1] M. Erdelj and E. Natalizio, "Uav-assisted disaster management: Applications and open issues," in *2016 International Conference on Computing, Networking and Communications (ICNC)*, Feb 2016, pp. 1–5.

[2] S. Lee, D. Har, and D. Kum, "Drone-assisted disaster management: Finding victims via infrared camera and lidar sensor fusion," in *2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE)*, Dec 2016, pp. 84–89.

[3] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness requirements for adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 60–69. [Online]. Available: http://doi.acm.org/10.1145/1988008.1988018

[4] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 341–350. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985840

[5] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. R. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Cámara, R. Calinescu, M. B. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J. Jézéquel, S. Malek, R. Mirandola, M. Mori, H. A. Müller, R. Rouvoy, C. M. F. Rubira, É. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. M. Villegas, T. Vogel, and F. Zambonelli, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*, 2013, pp. 3–30.

[6] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Softw. Syst. Model.*, vol. 15, no. 1, pp. 31–69, Feb. 2016.

[7] D. M. Barbosa, R. G. de Moura Lima, P. H. M. Maia, and E. C. Junior, ": A tool for runtime monitoring and verification of self-adaptive systems," in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 24–30.

[8] P. Arcaini, E. Riccobene, and P. Scandurra, "Formal design and verification of self-adaptive systems with decentralized control," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 4, pp. 25:1–25:35, Jan. 2017.

[9] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[10] E. Bondi, A. Kapoor, D. Dey, J. Piavis, S. Shah, R. Hannaford, A. Iyer, L. Joppa, and M. Tambe, "Near real-time detection of poachers from drones in airsim," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 5814–5816. [Online]. Available: https://doi.org/10.24963/ijcai.2018/847

[11] J. Cleland-Huang, M. Vierhauser, and S. Bayley, "Dronology: an incubator for cyber-physical systems research," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 109–112. [Online]. Available: https://doi.org/10.1145/3183399.3183408

[12] Y. Yu, J. C. S. do Prado Leite, and J. Mylopoulos, "From goals to aspects: Discovering aspects from requirements goal models," in *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, 2004, pp. 38–47. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/RE.2004.23

[13] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 313–, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/503271.503260

[14] M. Fowler, "Inversion of control containers and the dependency injection pattern," http://www.martinfowler.com/articles/injection.html, 2004, accessed: 2015-07-23.

[15] S. Chiba and R. Ishikawa, "Aspect-oriented programming beyond dependency injection," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 121–143.

[16] M. Wermelinger and Y. Yu, "Analyzing the evolution of eclipse plugins," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 133–136. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370783

[17] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Trans. Softw. Eng.*, vol. 29, no. 2, pp. 99–115, Feb. 2003. [Online]. Available: https://doi.org/10.1109/TSE.2003.1178048

[18] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., Jun. 2005.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.