Separation of Concerns Within Robotic Systems Through Proactive Computing

Alexandre Frantz
University of Luxembourg
2, avenue de l'Université, 4365 Esch-sur-Alzette
Email: alexandre.frantz.001@student.uni.lu

Denis Zampunieris
Faculty of Science and Telecommunications
2, avenue de l'Université, 4365 Esch-sur-Alzette
Email: denis.zampunieris@uni.lu

Abstract—**In this short paper, we first introduce a possible new model for designing and implementing software in robotic systems. This model is based on proactive scenarios, coded through dynamic sets of condition-action rules. Each scenario embeds the required rules and can be assembled dynamically with others, allowing the proactive system to achieve a unique objective or behavior and instruct the robot accordingly. Furthermore, a scenario is not aware of the existence of the other scenarios. In fact, it only contains information about a predefined central scenario, which oversees global decision making. In addition, each scenario knows where to enter its suggestions, thus allowing for a high degree in terms of separating concerns and modularity of code. Consequently, allowing easier development, testing and optimization of each scenario independently, possible reuse in different robots, and finally, a faster achievement of robust and scalable robotics software. We then show how to apply this programming model and its functionalities during runtime, by a proof of concept consisting of a virtual robot deployed in the Webots™ simulator. This simulator is controlled with four proactive scenarios (plus the central one), in charge of three different objectives.**

I. Introduction

Robotic software systems are growing in complexity and it should be noted that most current systems mix their different functionalities together. Data acquisition, sensor fusion, decision-making for several objectives and reading commands are all handled within the same location. Therefore, preventing scalability and extendibility, fundamentals each large system should strive for. Moreover, this bundling of functionalities results in large, complex and even close to obfuscated pieces of code that are not easily optimizable, maintainable or reusable in any optimal way.

In this paper, we will present how we addressed this problem, through the example of a robot simulation. Initially the simulation started as simple as possible, which was a robot that can be user controlled or autonomously move around its environment. Already with these functionalities, the logic was becoming more and more complex. Data acquisition from sensors, reading user input and computing the next action, were all handled together. This produced the following question: How could we safely extend the robot's behavior easily in the present and future, without having to make major changes in each part of our code? This question will be answered in the following sections, as we present a new model of programming that solves the issue at stake. With that said, the introduction is followed by the problem statement section, where we present the problem at hand and the procedure taken to overcome it. Next, we move on towards the tools and software used during our work, before continuing towards the proof of concept and implementation of our new programming model. Lastly, but certainly not least, we will present our results before ending with a conclusion and possible future work applications.

II. Problem Statement

The problem we will be addressing in this paper is concerned with the lack of separation of concerns within current robotics software systems. As time progresses, demand for more powerful and well-equipped software is mandatory for us to keep up with the exponential progression of technology. Software systems are bound to be scaled and extended; however, without proper separation of concerns, these systems will be limited. Our objective is to tackle this issue, by applying fundamental principles of Proactive Computing. We will use proactive behavior and scenarios to separate concerns, consequently promoting scalability and extendibility. In fact, the goal is to handle one concern at a time per module, as we will prove with this new model of programming. Moreover, the above was accomplished with the use of a proactive engine at hand, which is a rule-running system. It was implemented within a preprogrammed robot simulation, in order to extend it with proactive scenarios, serving as our main example in this paper.

Furthermore, we also provide a different way to fuse Computer Science with Robotics, giving them a way to interact and work together. This alternative method facilitates the fusion between these two domains in a new way, through the implementation of the above-mentioned proactive engine and the definition of proactive scenarios, allowing us to extend the simulation with them. The simulator will support the different scenarios, whereas the engine will handle them accordingly.

III. Tools and related software

A. Proactive Computing

The notion of proactive computing was initially introduced by David Tennenhouse in 2000 [11] and refers to the idea of transitioning from interactive systems, towards proactive systems (PS). In fact, until now interactive computing is done from a human-centered perspective, whereas proactive computing will be human-supervised. The difference between the two forms of computing, is that on one side we have the user incorporated within the execution loop (i.e. interactive computing), whereas on the contrary, the user is “above the loop” (i.e. proactive computing). The main motivation is to provide a system that can proactively determine our needs and dynamically act. Thus, allowing the PS to act for and on behalf of the user on their own initiative [2,3,7], excluding them from the “loop”. The main advantage of this transition is faster computation time, because computers will handle decision making themselves. With this idea in mind, Professor Zampunieris and his team developed a rule-based proactive engine at the University of Luxembourg. This engine has been used in many projects, focusing mainly in the domains of e-Learning, cognitive science and eHealth [1].

B. Proactive Engine

The main topic of this paper relies on the use of the above-mentioned proactive engine (PE). The PE, which is coded in Java, strongly encompasses the principles of Object-Oriented Programming and event-driven programming as well [1]. Therefore, making it a perfect middleware for our robot simulation. The engine can be directly attached on other systems or be used together with a database, which acts as the middleware and handles communication. With these aspects introduced, it is equally important to provide a description of the PE’s structure which consists of rules, scenarios and the database.

The engine consists of two queues, called currentQueue and nextQueue. The first contains the rules that will be executed in the current iteration, whereas the second contains the rules that are to be executed in the next iteration step [6]. The engine inserts rules in the queues and executes them in a First in First Out manner. Once a rule has been iterated over, it is removed from the currentQueue and the engine will fetch the next one in order (i.e. located in nextQueue), however there is a way to keep the rule executing in later iterations, which we will explain shortly. The detailed explanation behind the engine’s iteration can be found in references [6,4].

C. Proactive Scenarios

A collection of rules with a common goal define a certain scenario. These so-called scenarios create context for a specific situation, that the PE will handle and execute accordingly. For example, we could have a data acquisition scenario, which dynamically collects data from sensors and determines when to ask for more. A proactive system may have more than one scenario, and ideally these scenarios must not know more information than their own. Each scenario is initialized by the meta-rule, which determines based on context, when to activate all its rules during execution [6].

D. The Database

It acts as the middleware between two systems, linking one system with the proactive engine. Namely, the engine may also be used to read/write data between the two systems, such that we can achieve communication. However, another important use of the database, is to save the state of the proactive system, acting as a failsafe that contains crucial information of the engine. The information can be historical data from sensors and results obtained by executing the rules before a crash [6].

E. Webots™ Simulation Software

The simulation environment that we chose as is Webots™ and we believe that it is a great candidate for our work. Webots™ is a professional robot simulator that is used for not only R&D (research and development), but also for educational purposes. The simulator provides a complete development environment, which allows us to primarily model, program and simulate the use of robots. Each robot is controlled by a specific file, called the “controller”. The compiler will execute this file, which contains the behavioral logic behind our robot. The simulator offers a great GUI (graphical user interface) with which we can construct our environment, by adding/removing objects, robots etc. Furthermore, each controller can be programmed with a multitude of languages, such as Java, Python, C, C++ and more [9, 10]

IV. Proof of Concept

Before reading the following paragraphs, it is recomended to get a first view of the simulation structure beforehand which is represented in Figure 1.

A. Robot Side

The robot side consists of two main java class files, the fist being the RobotSuperclass and the second being the ProactiveController. These two files are called controllers and are compiled and executed within Webots™. The RobotSuperclass controller, acts as a common superclass for the controllers that have been defined for our robot simulation. In fact, there are more controllers that are attached to different robots; however, they will not be covered in this paper. The inheriting class (i.e. ProactiveController), is the main controller file we are interested in. The superclass defines common methods and variables that are used in the hierarchy, whereas as the inheriting class in our diagram, implements the methods needed to run our proactive robot. The two-colored arrows, blue and red, represent the writing and reading actions on the database.

The Proactive Controller is in charge for controlling our robot, given some instruction from the PE. Within the controller, we initialize all the sensors and variables needed to execute instructions and send data to the PE. For example, the GPS, the Inertial Unit (IU), the Distance Sensors (DS) and the variable mode (enumeration) that defines the robot’s next action. Furthermore, we write and read data on the database for specific reasons. The sensor values are written on their corresponding tables, for the PE to read and determine the next action. In turn, the controller continuously reads the database for the next instruction and switches the value of the mode accordingly. The different values are stored in an enumeration, and the mode is updated within a switch during runtime.

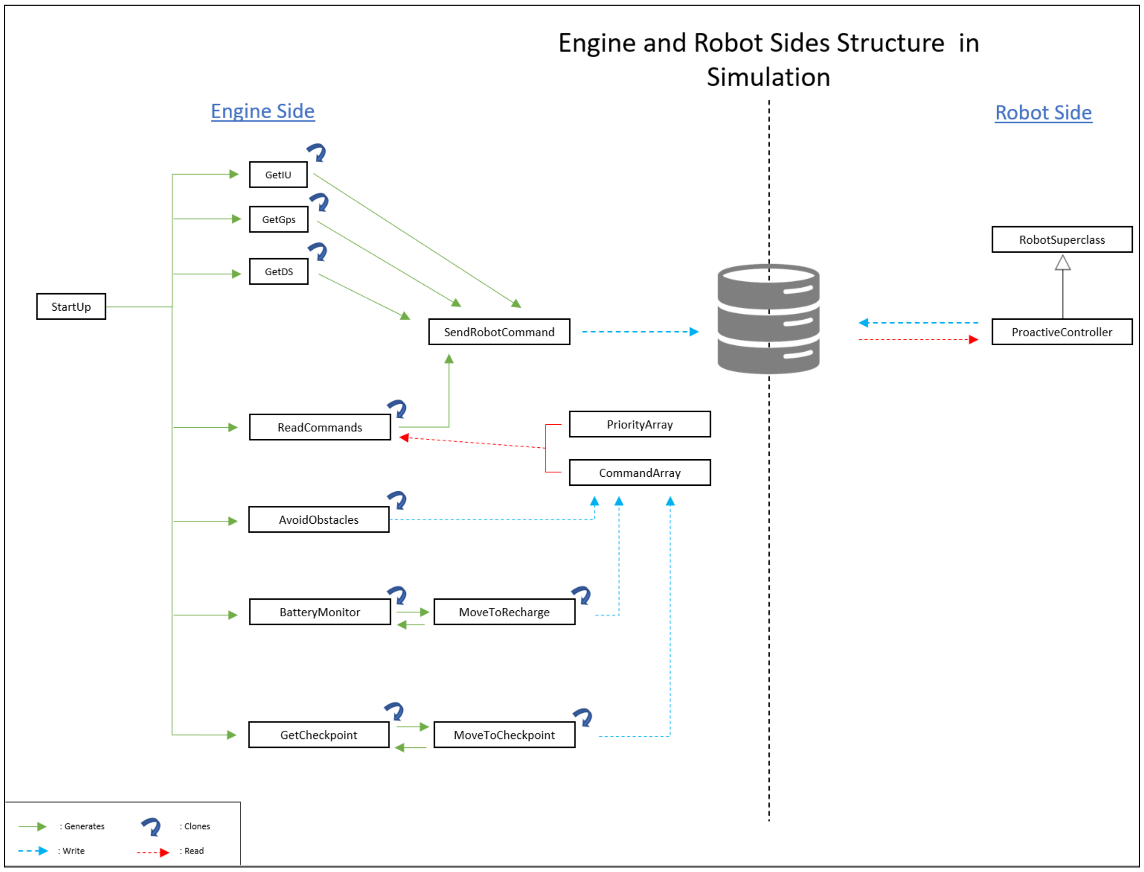

Fig. 1. Engine and Robot side Structure

B. Engine Side

The engine side consists of five main scenarios that define our robot simulation. Each scenario performs the necessary computations to achieve their goal, and is comprised of rules that define them. These rules perform the necessary computations to achieve their scenario's goal. The computations are dependent on the values read from the database. Once a rule has decided to propose an instruction to the robot, it writes on a common 1 x n array called: CommandArray (see Figure 1). Each scenario has been given a default index, which defines its priority, to enter their proposal in the array. Moreover, once a command has been selected to be sent to the robot, the corresponding rules generate the "SendRobotCommand" rule, that writes into the database.

Scenario 1: Data Acquisition

This scenario is defined by three main rules: GetGPS, GetDS and GetIU. Each of these rules check that there exists a value from the corresponding sensors (GPS, DS and IU) within a certain interval. Once the time passed is greater than the interval, the rules directly write on the database that they require new data on a given sensor. It is important to note, that these rules are not required to write on the CommandArray, since they do not have a specific priority.

Scenario 2: Reach Destination

Here, we have defined a scenario that moves the robot to one or more checkpoints. The rules in charge of this process are: GetCheckpoint and MoveToCheckpoint. As the names suggest, the first one retrieves the first non-reached destination from the database and generates the second rule. MoveToCheckpoint in turn, instructs the robot to move towards the checkpoint, by using the GPS and IU values. It will clone itself until the checkpoint has been reached. Furthermore, after reaching the current destination, the process restarts and the scenario retrieves, if any, the next non-reached destination. Finally, all the suggested commands that the scenario wants the robot to execute, are written on the common array.

Scenario 3: Obstacle Avoidance

The PE is also able to determine whether there is always an obstacle in the path of the robot. In the case that the robot is about to collide with one, the Obstacle Avoidance Scenario intervenes. The scenario is defined by only one rule: AvoidObstacles, whose job is to determine, based on the values of the DS if an obstacle is too close to the robot. If the case is true, the scenario enters a suggestion in the common array, which stops the robot if it is accepted.

Scenario 4: Battery Monitor

Our robot is consuming battery while it has moved a certain distance and it is the goal of this scenario to determine whether it is time to recharge its batteries. This scenario works almost identically to the 2nd one, with one small difference. The process starts with the BatteryMonitor rule, that reads the battery levels from the database and decides if the robot can reach its destination. If it is, the rule clones itself and restarts the same process. In the case it that the robot does not have enough battery, the same rule generated the MoveToRecharge rule. It should be noted that this rule is identical to the MoveToCheckpoint rule of the 2nd scenario. It instructs the robot to move to the recharge point (defined by GPS coordinates within the database), and once it has been reached it recharges the robot's batteries. The rule clones until the recharge point is reached.

Scenario 5: Decision Making and Strategy

The last and most important scenario of our PE oversees the decision-making mechanism of the whole system. It is defined by one rule, ReadRobotCommands and two important arrays: PriorityArray and CommandArray. The rule determines which command to write to the database, and consequently send to the robot, based on the priority of each scenario. The process starts by retrieving highest interger (priority) within the PriorityArray. The value is stored, and then the rule retrieves the string in the CommandArray, with the given index (priority). If a value exists, the rule writes it to the database, else, we retrieve the second highest priority and its command. The process repeats until the engine is terminated. Lastly, within the PriorityArray we can define the strategy we wish to follow during runtime. This strategy constraint defines the priority of each scenario and can be dynamically changed during execution to fulfill different goals.

C. Database Side

The database was created using the MySQL workbench™ and contains eight schemas. As mentioned previously, the database's role is to act as middleware, establishing communication between robot and engine. Each side reads and write data on the database schemas, which are the following:

- **Battery_level:** Contains the battery levels that the robot inserts
- **Battery_recharged:** Contains the level we wish to recharge the battery to
- **Checkpoints:** Contains the checkpoints the robot must reach
- **Distance_sensor_data:** Contains data on the DS
- **Engine_comments:** Contains the instructions the robot must execute, sent from the PE
- **Gps_Stored_values:** Contains the GPS coordinates of the robot
- **Iudata:** Contains the IU values during iteration
- **Recharge_station:** Contains the coordinates of the Recharge Station

D. Relation between scenarios

The scenarios are running in parallel within the PE and are initiated in the following order:

Scenario 1 $\rightarrow$ Scenario 4 $\rightarrow$ Scenario 2 $\rightarrow$ Scenario 3 $\rightarrow$ Scenario 5

They will work together, without explicit interaction, to instruct the robot towards the destination, given the current strategy setting. Firstly, the StartUp rule will initialize all the scenarios, while at the same time, the robot awaits instructions from the PE and initializes its battery levels to the max value (that value is sent to the database). Scenario 1 will repeat itself and ask for the robot to update its sensor values in the database, whereas Scenario 3 will determine whether the battery levels are too low to reach our destination. Next, Scenario 2 will retrieve the first non-reached destination and generate the rule MoveToCheckpoint, which will instruct the robot towards the destination (i.e.. insert its suggestion in the CommandArray). While this process is executed, Scenario 4 makes sure that there is no risk of collision by checking the DS values from the database. Finally, Scenario 5 will read the suggestions of all the rules and determine which one to send to the robot, based on the highest priority which is defined by the current strategy. Once the robot has reached its destination, the PE has achieved its goal.

V. Results-Discussion

We mentioned in previous sections that we created a robot simulation using Webots™. In fact, we created a world which consists of a circular arena that the robot traverses during execution. The arena itself contains certain obstacles, such that we can apply our obstacle avoidance. We should also note, that on the arena we are free to define multiple destinations and one recharge point. Furthermore, to test our battery consumption, obstacle avoidance and the rest of our scenarios, we simulated multiple instances of our world with different values for our parameters. For example, different destination coordinates, battery levels, and obstacles.

With the implementation of the proactive engine within the robot simulation, we can argue that we have successfully addressed and solved our problem.With the addition of the PE we have separated the functionalities with the definition of scenarios and rules, each contributing to the robot's objective. The decision making, data acquisition, and the computations behind the robot's goal, have been moved to the engine side. Consequently reducing the controller's complexity and only containing code that defines how the robot executes commands. Whereas, the engine handles all the necessary computations to control the robot.

Moreover, the defined scenarios do not interact explicitly with each other. However, each scenario contains information only about itself, and works independently towards the same objective. Since they can work in parallel, each concern is handled in its own location and together they create a working system that can successfully instruct the robot. For example, the data acquisition scenario requests sensor information from the robot, the scenario dedicated to move the robot to its objective, reads the information and determines how to instruct the robot, and finally the decision making scenario determines which instruction has the highest priority in the end. In other words, we achieved separation of concerns and proved that we can address our problem in a different way, namely through proactive behavior and rule-based systems.

Finally, the communication is achieved with the use of the database, acting as the middleware between the two sides, and without it we would not be able to implement the PE. The database schemas contain all the data needed to exchange information and allow the two sides to communicate. In fact, the use of the database is the default design for our middleware, to link a proactive system with another one. The main advantages are, that we do not need to modify both systems to great extent, to facilitate communication, as well as allowing the developer to program the extended system in a different language. Furthermore, one could imagine another way of directly linking the robot environment with the PE, theoretically this would be possible by defining a different communication layer between the two sides.

VI. Conclusion

We have presented our work regarding the extension of a robot environment, by implementing a proactive engine within the simulation. We stated our problem, which is the lack of separation of concerns in robotic systems and proposed a viable and effective solution for it. The proactive engine at hand allowed us to use proactive behavior and rule-driven programming, to define proactive scenarios and rules, therefore achieving separation of concerns. We managed to move the functionalities that were originally handled on the robot side, towards the engine side, such that we can implement them in separate scenarios which run in parallel. Therefore, the robot is explicitly directed by the engine and only executes the commands it has been sent. On the other hand, the engine handles the original functionalities, such as data acquisition, decision-making and moving from one objective to the next. In addition, the strategy feature that is embedded in our decision-making scenario, allow the users to change and define new strategies, and therefore context, before or during runtime. Moreover, the database serves as the middleware that handles data exchange and communications between the robot and engine sides. Lastly, but certainly not least, not only did we solve the problem at stake, we also provided a way to link Computer Science with Robotics through our simulation environment and the proactive engine, therefore breaking barriers and creating links.

VII. Future Work

A possibility for future work applications of our proactive engine, is to extend outside of the confined spaces of a simulation, towards a real-life robot. We could argue that the same principles of our work, could be implemented in a simulator that supports ROS (Robot Operating System), such as Gazebo [8]. Once the simulations return the results we expected, we will be one step closer to implementing the PE in a physical robot, and test how it performs. This idea provides a new way of controlling and managing robots, through proactive behavior, and opens doors to more possibilities that could be scaled to a large variety of applications, such as smart homes, medicine, and teaching. Furthermore, one might ask how we can negate critical errors within the system, such as a rule failing to execute and needing to clear the CommandArray, such that no inappropriate command executes. In fact, this could be handled by simply extending the proactive engine with the creation of fail-safe dedicated scenarios which handle exactly this concern.

References

[1] Gilles Neyens, *Confidence-Based Decision-Making Support for Multi-Sensor Systems* Doctoral Thesis, University of Luxembourg, Luxembourg 2019

[2] Gilles Neyens, Denis Zampunieris, *Proactive Model for Handling Conflicts in Sensor Data Fusion Applied to Robotic Systems* Proceedings of the 14th International Conference on Software Technologies (ICSOFT), 2019 Prague, Czech Republic, 26 - 28 July, 2019

[3] Gilles Neyens, Denis Zampunieris, *Proactive Middleware for Fault Detection and Advanced Conflict Handling in Sensor Fusion* Proceedings of the 18th International Conference, ICAISC 2019 Zakopane, Poland, June 16–20, 2019, Part I & II

[4] Gilles Neyens, Denis Zampunieris, *A rule-based approach for self-optimisation in autonomic eHealth systems* Workshop Proceedings ot the 6th International Workshop on "Self-Optimisation in Autonomic & Organic Computing Systems" in ARCS 2018 - 31st International Conference on Architecture of Computing Systems, Braunschweig, Germany, 09 - 12 April, 2018

[5] Dobrican Remus-Alexandru, Denis Zampunieris, *A Proactive Solution, using Wearable and Mobile Applications, for Closing the Gap between the Rehabilitation Team and Cardiac Patients* Proceedings of the IEEE International Conference on Healthcare Informatics 2016 (ICHI 2016)

[6] Denis Zampunieris, *Implementation of a Proactive Learning Management System* Proceedings of "E-Learn - World Conference on E-Learning in Corporate, Government, Healthcare & Higher Education" 2006

[7] Salovaara, A. , Oulasvirta, A. *5)Six modes of proactive resource management: auser-centric typology for proactive behaviors* Proceedings of the Third Nordic Conference on Human-Computer Interaction 2004, Tampere, Finland, October 23-27, 2004

[8] Wikipedia, *Robot Operating System, ROS : https://en.wikipedia.org/wiki/Robot_Operating_System*

[9] Cyberbotics Ltd, *Webots User Guide: https://cyberbotics.com/doc/guide/index*

[10] Wikipedia, *Webots: https://en.wikipedia.org/wiki/Webots*

[11] David Tennenhouse, *Proactive Computing* Communications of the ACM, May 2000