2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) | 978-1-6654-0289-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEAMS51251.2021.00037

Body Sensor Network: A Self-Adaptive System Exemplar in the Healthcare Domain

Eric Bernd Gil
University of Brasília
ericbgil@gmail.com

Ricardo Caldas
Chalmers | University of Gothenburg
ricardo.caldas@chalmers.se

Arthur Rodrigues
University of Brasília
arthur.farias@aluno.unb.br

Gabriel Levi Gomes da Silva
University of Brasília
gabrielevigomes@gmail.com

Genaína Nunes Rodrigues
University of Brasília
genaina@unb.br

Patrizio Pelliccione
Chalmers | University of Gothenburg
Gran Sasso Science Institute (GSSI)
patrizio.pelliccione@gssi.it

***Abstract*—Recent worldwide events shed light on the need of human-centered systems engineering in the healthcare domain. These systems must be prepared to evolve quickly but safely, according to unpredicted environments and ever-changing pathogens that spread ruthlessly. Such scenarios suffocate hospitals' infrastructure and disable healthcare systems that are not prepared to deal with unpredicted environments without costly re-engineering. In the face of these challenges, we offer the SA-BSN – Self-Adaptive Body Sensor Network – prototype to explore the rather dynamic patient's health status monitoring. The exemplar is focused on self-adaptation and comes with scenarios that hinder an interplay between system reliability and battery consumption that is available after each execution. Also, we provide: (i) a noise injection mechanism, (ii) file-based patient profiles' configuration, (iii) six healthcare sensor simulations, and (iv) an extensible/reusable controller implementation for self-adaptation. The artifact is implemented in ROS (Robot Operating System), which embraces principles such as ease of use and relies on an active open source community support.**

***Index Terms*—Body sensor network, self-adaptive systems, healthcare exemplar, cyber-physical systems, control theory.**

I. Introduction

In spite of various research efforts in IT for the healthcare domain in the last years, e.g. [1], [2], the development of healthcare self-adaptive cyber-physical systems is still quite challenging due to its safety critical nature. The provision of assurance evidences for the compliance of quality attributes such as safety [3], reliability [4], and cost [5] has a vital importance in this domain. Therefore, equipping these systems with self-configuration, self-healing, and self-management competences has become paramount for healthcare and assisted living applications.

The evolution of self-adaptive systems can be explained from a historical perspective through a set of complementary stages, namely *waves* [6]. These waves encompass concepts like: (i) automation of management tasks; (ii) architecture-based adaptation; (iii) use of models at runtime; (iv) adaptation based on goals; (v) provision of guarantees under uncertainties; (vi) control-based approaches; and (vii) use of learning techniques in adaptation. Although some of the concepts in *vi* and *vii* are well-established for at least five years [7], [8], and the inspiration from machine learning and control theory on self-adaptive systems goes way beyond that, we are just starting to grasp the benefits of adopting their fundamentals in the engineering of self-adaptive cyber-physical systems.

The SEAMS community has steadily supported in self-adaptive systems [9]. Although there are proposals available in the healthcare domain [10] and in the engineering of safety- and mission critical adaptive systems [11]–[13], to the best of our knowledge, there is no particular exemplar of a cyber-physical system based on control-theoretical principles in the domain of adaptive body sensor network systems. Driven by the need of having a control theory-based prototype to explore the effects of uncertainties on quality attributes of safety critical applications, we present the Self-Adaptive Body Sensor Network exemplar (SA-BSN) [14]. The SA-BSN is designed as a software exemplar to monitor and analyse the health statuses of patients individually, through a set of sensors in tandem with a centralized processor. Moreover, as an exemplar for the self-adaptive domain, the adaptation goal of the SA-BSN is to keep a target reliability level while accounting for a target energy consumption management. In the past few years, our artifact has been not only under constant evolution but also adopted in the evaluation of some previous works of our research group [15]–[18], proving itself as an artifact that is synergistic with the ideas presented in the aforementioned waves *vi* and *vii*. The use of our exemplar is straightforward, and we have made it available online, together with a virtual machine, supporting a set of scenarios and adaptation policies[1]. The executable scenarios we propose demonstrate how our artifact adapts itself to cope with three distinct classes of uncertainty: the system itself, the system goals, and the environment.

The rest of the paper is structured as follows. In Section II, we present an overview of SA-BSN, provided with adaptation scenarios and the quality attributes involved. Section III describes the exemplar from the architecture and implementation perspectives, including implementation details as well. In Section IV, we guide the reader to use the provided artifact for experimentation. Finally, Section V concludes the paper with the next steps.

[1]https://github.com/lesunb/bsn

2157-2321/21/$31.00 ©2021 IEEE
DOI 10.1109/SEAMS51251.2021.00037

2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) | 978-1-6654-0289-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEAMS51251.2021.00037

II. SA-BSN Exemplar and Adaptation Overview

The SA-BSN is an exemplar of a healthcare application implemented in ROS [19]. The goal of the SA-BSN is to detect emergencies by continuously monitoring the patient's health status. Furthermore, the SA-BSN is equipped to adapt itself in order to maintain the desired QoS levels with minimal human intervention, while accounting for classes of uncertainty. Hereafter we stand upon the well-known architectural view of managed and managing system as means of seeking for separation of concerns [6], [20] to refer to the corresponding SA-BSN Managing and Managed System modules and their responsibilities. In this section, we describe the SA-BSN exemplar requirements in a goal-oriented perspective.

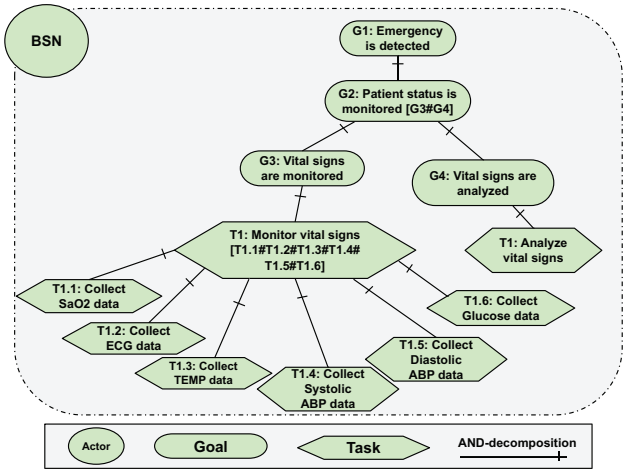

Fig. 1: The Goal Model used to represent the BSN

According to the goal-oriented view of the SA-BSN's functional requirements shown in Figure 1, the Managed System has six sensors to continuously monitor the patient's vital signs and achieve its goal of analysing the vital signs and detecting an emergency when it occurs. A range of vital signs is periodically collected from the patient through a set of distributed sensors: electrocardiograph sensor (ECG) for heart rate and electrocardiogram curve; a pulse oximeter (SaO2) for measuring blood oxygen saturation; a thermometer (TEMP), that collects the body temperature in Celsius; a sphygmomanometer for measuring and systolic arterial blood pressure (ABP); there is also a Glucose sensor for measuring blood glucose levels. The collected data is then forwarded to the Central Hub: a component in the Managed System to fuse the vital signs and classify the overall health situation of the patient into *low*, *moderate*, or *high* risk status. In addition to that, the Central Hub can stack other responsibilities like preprocessing the incoming data, filtering the redundancy, and translating communication protocols.

The Managing System module of the SA-BSN is, in turn, responsible for continuously assuring the fulfillment of the desired QoS attributes related to the values of reliability and battery consumption (i.e. cost). For the evaluation of the quality of the adaptation we use control theory metrics following the terminology proposed by Camara et al. [21]. The chosen QoS constraint is attributed to a setpoint, which is set by the user before the system execution. For example, if the concerned QoS attribute is the reliability, one could set it to 95%, within an acceptable error range. This is called setpoint tracking, which can be measured by the steady-state error (SSE) metric. In addition to this requirement, the user can verify other control theoretic metrics related to the transient behavior, which can be evaluated at the end of the system execution. For instance, the user could set the threshold for the adaptation overshoot, specifying that it should not exceed 10% of the target value (setpoint). Another example would be choosing the settling time in a way that the adaptation should not take more than 3 minutes to converge. We further illustrate these metrics in Section IV, where we present a running scenario for the BSN.

While trying to meet its requirements, the system is prone to a range of uncertainties. Thus, the controller is activated to mitigate the effects of unexpected events in quality attributes. Table I presents the uncertainties and the adaptation goals of three scenarios that can be executed in the SA-BSN. The first scenario, S1, focuses on uncertainties related to the overflow of sensed data into the the Central Hub queue and also to the possible data uncertainties in sensors, which are related to the reliability of the system. The second scenario, S2, focuses on the uncertainty related to the operational frequencies of the components, which can lead to a battery consumption that exceeds what is needed to satisfy the requirements. In the third scenario, S3, depending on the patient profile, the operator may not want to use certain sensors; with fewer components to manage, less uncertainty in the system is expected and, consequently, a more stable adaptation process.

III. SA-BSN Implementation Details

In this section we further delve into the architecture perspective (Section III-A) and the implementation details (Section III-B) of the SA-BSN exemplar.

A. SA-BSN architecture on ROS

The SA-BSN artifact is composed of four main modules: Managing System, Managed System, Knowledge Repository and Simulation, as depicted in Figure 2. Below we further detail these architecture modules of the SA-BSN and their functionalities.

1) The Managing System: This SA-BSN module comprises the Strategy Manager and Strategy Enactor and is in charge of implementing the controller to deal with the adaptation issues. The Strategy Manager is responsible for estimating the reliability and cost setpoints for the components of the Managed System module, given a system desired setpoint and the system's reliability and cost estimated via a parametric formula available in the knowledge repository [18]. The Strategy Enactor is where the controller is implemented; it is responsible for applying the adaptation strategies to achieve the previously estimated setpoints for each component.

Figure 3 shows interaction between these two components and the other parts of the system and the closed feedback

Scenario	Uncertainty Class [17] : Type	Impact	Adaptation Policy	Affected QoS Attribute
S1	SI: unexpected number of users	Delays on message processing	Adjust Central Hub service time rate	Reliability
	SG: uncertain sampling rate	Uncertain mean time to failure	Adjusting sensors' sampling rate	Reliability
S2	SG: uncertain sampling rate	High battery consumption	Adjusting sensors' sampling rate	Cost
S3	ENV: uncertain sensor availability	Unwanted sensors activated	Disable unwanted sensors	Cost, reliability

TABLE I: SA-BSN adaptation scenarios under different classes of uncertainty (SI: System Itself (S1); SG: System Goals (S1-S2); ENV: execution context (S3))

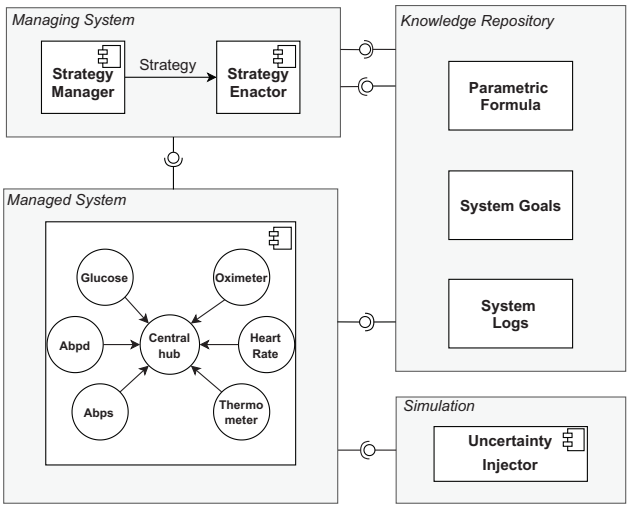

Fig. 2: Architectural perspective of the SA-BSN as a self-adaptive exemplar.

loop for Strategy Enactor, which works for both the Managed System components' and system setpoints. This component uses the setpoints estimated by the Strategy Manager and compares them to the actual value of the desired QoS attribute (i.e., reliability or cost) for each component, applying the adaptation policy according to the analyzed status. In our case, the adaptation policy consists in adjusting the components frequencies according to the calculated error using the controller in order to simulate the sensors' sampling rate and the central hub processing rate. These knobs directly impact the overall system reliability since: (i) more data points collected by a sensor per time unit is expected to render a more precise measurement of the vital signal and (ii) messages lost in the Central Hub due to queue problems may impact the reliability of the system. Also, they impact the energy consumption (cost) of the components since more executions are performed per time unit. We should note that the BSN is not limited by actuation through such knobs or control based on reliability or cost. These are interchangeable with other actuation mechanisms and QoS attributes.

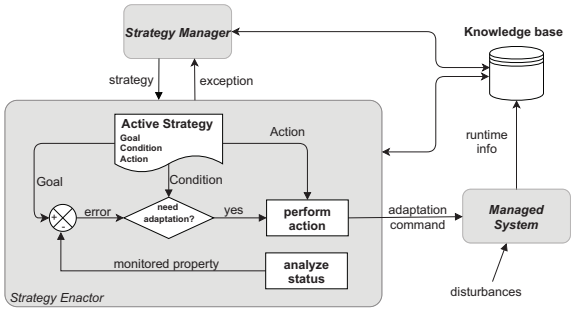

Fig. 3: System feedback loop and its interactions [18]

2) The Managed System: The Managed System comprises the sensors for vital signs monitoring and the Central Hub. The components responsible for the communication between this module and the other SA-BSN modules are the Probes and the Effectors. The Probes are responsible for gathering data from the Managed System components and sending them to the Knowledge Repository and the Managing System. The Effectors are responsible for receiving adaptation commands from the Strategy Enactor and changing the Managed System components parameters accordingly.

3) The Knowledge Repository: The knowledge repository comprises (i) the parametric formulas to compute the reliability and energy consumption of the Managed System, which were generated using our Pistar-GODA MDP artifact [17], (ii) the goals to be achieved, in the form of the goal model, and (iii) the System Logs where knowledge about the system's execution is persisted.The System Logs which consist of 5 different types of logs: Adaptation, Status, Event, Uncertainty and EnergyStatus. The Adaptation log is where the Strategy Enactor adaptation commands are persisted. The Status log is where information about Managed System components status is persisted. The Event log is where information about activation and deactivation of Managed System components is persisted. The Uncertainty log is where information about injected uncertainty is persisted. Finally, the EnergyStatus log is where cost information is persisted. We should note that the data persistence in the logs as well as the interface between the Knowledge Repository and the other SA-BSN modules is carried out by the Data Access component.

4) The Simulation Module: This fourth SA-BSN module comprises the Uncertainty Injector component to simulate the uncertainties envisioned for the Managed System. As such, this component is responsible for injecting uncertainty into the Managed System sensors in order to induce failure on the data collection process. One important comment to make here is that the sensors will not fail unless the uncertainty

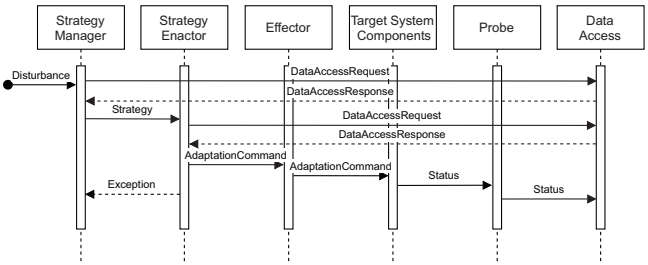

Fig. 4: Sequence diagram of a normal cycle of adaptation

injector is active. However, if it is desirable that a specific group of sensors do not fail under any circumstances, one must configure the injector to not inject uncertainty into them; this feature is explained in the next sections.

B. SA-BSN operation on ROS

The four SA-BSN modules are coordinated through ROS messages exchanged in a publish/subscribe architecture, using the TCP/IP communication protocol. The dynamic view of the adaptation process is shown in the sequence diagram in Figure 4, where we present the messages exchanged between components when an adaptation is required. The need for adaptation is detected when there is a disturbance in the attribute of interest, i.e., when the error is bigger than the setpoint times a stability margin, which is fixed in 0.02. If the adaptation is needed, the Strategy Manager sends a *DataAccessRequest* message to the Data Access component, which will fetch the log entries regarding failure rates or battery consumptions of the Managed System components. Then, the Strategy Manager estimates the setpoints for all Managed System components, and send them as a *Strategy* message to the Strategy Enactor. In its turn, the Enactor uses these setpoints to estimate the frequency of the components, sending an *AdaptationCommand* message to the Effector that redirects it to the target component. If any exception is verified, the Strategy Enactor sends an *Exception* message to the Strategy Manager, which acts accordingly. Finally, the Managed System components receive the *AdaptationCommand*, change their frequencies, and continue to send their current condition in periodic *Status* messages to the Probe, which forwards them to the Data Access node right after;

IV. Hands on the SA-BSN

In this section, we present a guide to aid the reader in setting up an experimental environment and execute an adaptation scenario as a demonstration[2] of the replicability of our exemplar.

A. Customizing the Strategy Enactor

We provide a default controller implementation into the Strategy Enactor component. We should note, however, that the researcher using the SA-BSN can implement its own controller. The objective of our default controller is to calculate frequency values for components according to the desired set-point values for the adaptation metric calculated in the Strategy Manager. Furthermore, the user-defined controller builds the *AdaptationCommand* message in order to communicate with the Effector. Similar to the other component in the system, the configuration of the controller and its parameters are defined in a launch file. Our default controller is a proportional controller, that consists in a proportional gain K_p, which acts directly on the error $e(t)$ between a desired setpoint and the current value of a given metric (e.g. reliability), generating a control output $u(t)$.

$$u(t) = K_p \times e(t) \tag{1}$$

For the case of a user-defined controller, the user must redesign the methods *setUp*, *apply_reli_strategy*, *apply_cost_strategy*, and *receiveEvent*, which are defined in the Controller source and header files. The *setUp* method is responsible for reading the launch file and assigning values to the respective controller parameters as well as receiving the QoS attribute to be adapted, which is defined in the Strategy Manager. The *apply_reli_-strategy* and *apply_cost_strategy* are the methods responsible for calculating the module's frequency values, depending on the QoS attribute used, building then the *AdaptationCommand* message. Finally, the *receiveEvent* method is responsible for receiving activation and deactivation signals, generated by the Managed System modules, and setting the parameters to default values.

B. System Configuration and Setup

The setup required to build the runtime environment for the SA-BSN, comprises five stages: (i) experimental environment setup, (ii) vital signs generation setup (iii) sensors setup (iv) building the System Manager and (v) configuring the Uncertainty Injection mechanism. Below we further detail each stage.

1) Experimental Environment Setup: There are two ways of obtaining the local version of the system, through the download of a virtual machine or the download of the source code. Further instructions to access both can be found in a Github repository[3], where we provide an installation guide.

In order to configure SA-BSN components parameters we use ROS features called launch files. They are XML-based files in which the tags are divided in three types: `launch`, `node` and `param`. The `launch` tag indicates the scope of the launch file. The `node` tag is responsible for setting up the component. The `param` tag[4] is responsible for the configuration of the components parameters and contains a `name` attribute, which represents the name of the parameter, a `value` attribute and an optional `type` attribute. When configuring the SA-BSN we mostly change the attribute value of the `param` tags.

[2]Check our demonstration video in Youtube. Link provided in the github repository.

[3]https://github.com/lesunb/bsn

[4]http://wiki.ros.org/roslaunch/XML/param

```
<launch>
    <!-- Blood Oxigenation Measurement Sensor -->
    <node name="patient" pkg="patient" type="patient" output="screen" />

    <param name="frequency" value="10" />

    <param name="vitalSigns" value="oxigenation, heart_rate, temperature, abps, abpd, glucose" />

    <!-- Frequency for changes in states of each markov in Hertz -->
    <param name="oxigenation_Change" value="0.2"/>
    <param name="heart_rate_Change" value="0.1"/>
    <param name="temperature_Change" value="0.1"/>
    <param name="abps_Change" value="0.1"/>
    <param name="abpd_Change" value="0.1"/>
    <param name="glucose_Change" value="0.1"/>

    <!-- Offsets for each changes, in seconds -->
    <param name="oxigenation_Offset" value="10"/>
    <param name="heart_rate_Offset" value="10"/>
    <param name="temperature_Offset" value="10"/>
    <param name="abps_Offset" value="10"/>
    <param name="abpd_Offset" value="10"/>
    <param name="glucose_Offset" value="10"/>

    <!-- Markov chain for oxigenation -->
    <param name="oxigenation_State0" value="0,0,0,0,0" />
    <param name="oxigenation_State1" value="0,0,0,0,0" />
    <param name="oxigenation_State2" value="0,0,90,8,2" />
    <param name="oxigenation_State3" value="0,0,75,10,5" />
    <param name="oxigenation_State4" value="0,0,5,35,60" />

    <!-- Risk values for oximeter -->
    <param name="oxigenation_HighRisk0" value="-1,-1" />
    <param name="oxigenation_MidRisk0" value="-1,-1" />
    <param name="oxigenation_LowRisk" value="65,100" />
    <param name="oxigenation_MidRisk1" value="55,65" />
    <param name="oxigenation_HighRisk1" value="0,55" />
```

Fig. 5: Patient module excerpt.

2) Vital Signs Generation Setup: For the vital signs generation setup the user needs to configure the launch file for the Patient module, where an example of a partial configuration is shown in Figure 5. This module is responsible for the generation of vital signs following a configured discrete-time markov chain model, so that sensors use ROS services to request for data in each execution. In the configuration file, we set the name of the vital signs to be generated (one for each sensor), as in line 7, the frequencies for each change (change rate), as in lines 10-15, and the offset (in seconds), as in lines 18-23, where the data changes only if it has passed *period* plus *offset* seconds since the last change. We also define the transition probabilities for each of the five states the sensors can assume, as in lines 26-30, and the range of risk values for each of these states, as in lines 33-37.

3) Sensors setup: For the sensors setup, the user needs to configure one launch file for each sensor. See Figure 6, for an example of the Oximeter's configuration. As shown in lines 10-12, the probability occurrences are configured for

```
<launch>
    <!-- Blood Oxigenation Measurement Sensor -->
    <node name="g3t1_1" pkg="component" type="g3t1_1" output="screen" />

    <param name="start" value="true" />

    <param name="frequency" value="1.3" /> <!-- 0.2 Hz  -->

    <!-- Defines the percentages to consider low, moderate or high risk -->
    <param name="lowrisk" value="0,20" />
    <param name="midrisk" value="21,65" />
    <param name="highrisk" value="66,100" />

    <!-- Risk values for oximeter -->
    <param name="HighRisk0" value="-1,-1" />
    <param name="MidRisk0" value="-1,-1" />
    <param name="LowRisk" value="65,100" />
    <param name="MidRisk1" value="55,65" />
    <param name="HighRisk1" value="0,55" />

    <!-- accuracy in percentage -->
    <param name="accuracy" value="99" type="double" />

    <!-- instant recharge parameter-->
    <param name="instant_recharge" value="true" type="bool" />

</launch>
```

Fig. 6: Sensor module excerpt

```
<launch>
    <node name="injector" pkg="injector" type="injector" output="screen" />

    <param name="frequency" value="6" type="int" /> <!--Hz-->

    <param name="components" value="g3t1_1,g3t1_2,g3t1_3,g3t1_4,g3t1_5,g3t1_6" />

    <!--Parameters for g3t1_1 uncertainty injection-->
    <param name="g3t1_1/type" value="random" />
    <param name="g3t1_1/offset" value="0" />
    <param name="g3t1_1/amplitude" value="0.025" />
    <param name="g3t1_1/frequency" value="0.5" />
    <param name="g3t1_1/duration" value="3" type="int" />
    <param name="g3t1_1/begin" value="0" type="int" />
```

Fig. 7: Uncertainty injector excerpt

low, moderate and high risks. In lines 15-19, the range of values are defined for the vital signs risks. The third and fourth parameters are the sensor accuracy, defined in line 22, and the instant_recharge parameter, defined in line 25. The instant_recharge specifies whether the sensor simulates the recharging of the battery or not. Finally, the last parameter is the start parameter, shown in line 5, which defines whether the sensor will be active during the execution or if it will be shutdown right at the beginning.

4) Building the System Manager: The System Manager components configuration entail the Strategy Enactor and the System Manager launch files.

To configure the Strategy Enactor, the user has to configure the frequency and K_p, since we provided a proportional controller as default. In case of a user-defined controller, we could have as many parameters as needed.

There are several parameters to be defined to configure the Strategy Manager: *monitor_freq*, *setpoint*, *actuation_freq*, *info_quant*, *offset*, *gain*, and *qos_attribute*. The *monitor_freq* is the frequency in which the Strategy Manager will monitor the values of the chosen adaptation metric. The *actuation_freq* is the frequency in which the Strategy Manager will calculate values for the setpoints of the Managed System components. The *setpoint* parameter is the system setpoint to be achieved, where the system reliability and cost are estimated via their corresponding parametric formula. The *info_quant* parameter specifies how many data points will be used to calculate the reliability. The *offset* and *gain* parameters are related to the search algorithm, together they delimit the size of the adaptation space. Finally, the *qos_attribute* parameter is name of the QoS attribute of interest. In order to use the engine for adaptation of reliability one must have to set the node parameter name and type attributes to "reli_engine" while for cost one must have to set them to "cost_engine".

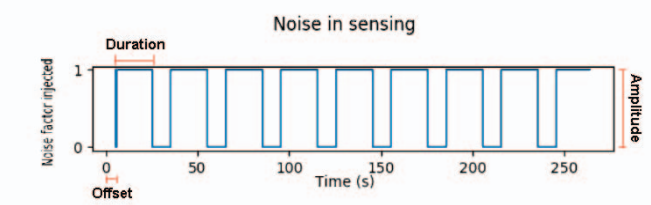

Fig. 8: Example of a step uncertainty signal waveform

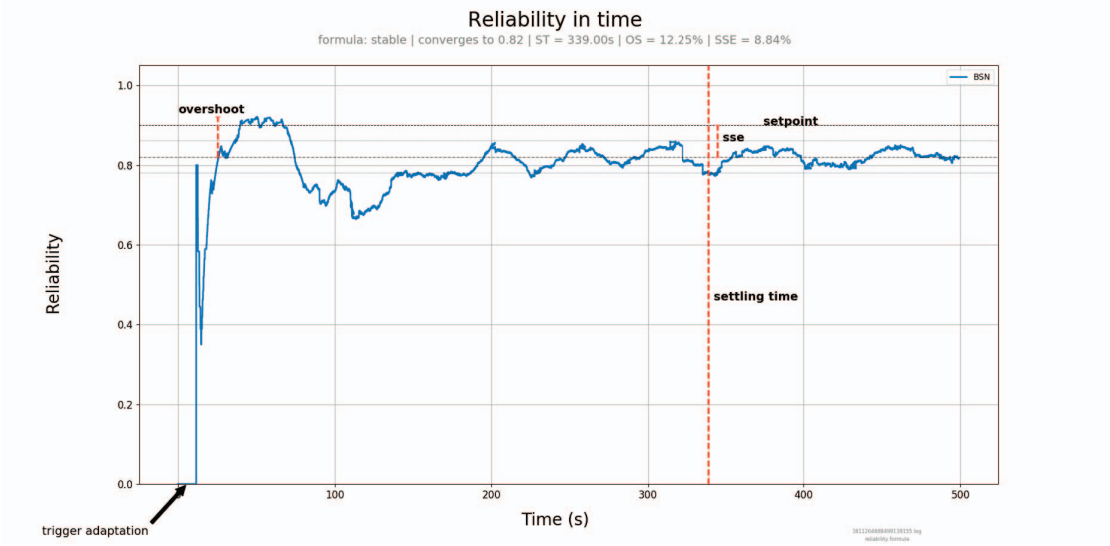

Fig. 9: Reliability curve obtained after the execution

5) Configuring the Uncertainty Injection Mechanism: The last setup stage required to run the SA-BSN is the configuration of the uncertainty injection mechanism, where noise is introduced into the sensors to induce failure by data inconsistency. An example of a partial configuration for this module is shown in Figure 7. The configurable parameters for this module are the injection frequency (line 4) and the sensors in which uncertainty shall be injected (line 6).

In lines 9-14, we refer to the configuration of a few other parameters that can be also considered for each sensor: type, offset, amplitude, frequency, duration, and begin. The *type* parameter defines the kind of wave the uncertainty will follow, which can be either `step`, `ramp` or `random`. An example of a `step` waveform is shown in Figure 8, where the wave parameters set in the configuration are indicated. It is important to notice that each component can be stimulated by a different type of noise. The `offset` parameter refers to the uncertainty offset to be injected. The `amplitude` parameter is the magnitude of the uncertainty value, which has implications on how much the noisy data will differ from the original collected data. The value of such a difference is what defines if the sensor failed or not due to the noisy input. The `frequency` parameter is the rate in which the uncertainty must be injected in the Managed System sensors. It is worth mentioning that, for sensors with instantaneous recharge (instant_recharge parameter), the `frequency` parameter defines an upper bound of the number of failures that can happen.

C. Example Scenario Evaluation

The user must execute the ready-to-go bash script (*run.sh*) to run the exemplar. The run command can be configured by maximum execution time (seconds) as an argument (e.g. "bash run.sh 30"). The default duration of the execution is 300 seconds, in case that no argument is passed.

Once the SA-BSN runs, a series of terminals will pop up on the screen, each corresponding to a component. By these means, the user can keep track of the execution progress. After the execution time has elapsed, the terminals will close, and the user can check the logs or run the *analyzer.py* script (inside the "src/simulation/analyzer" folder) to obtain a graph describing the monitored QoS attribute curve, i.e., reliability or cost. Including the aforementioned control-theoretic metrics of interest. Instructions for the use of the *analyzer.py* script are given in the SA-BSN GitHub repository.

To illustrate our exemplar, we exercise scenario S1 (cf. Table I) and plot the results in Figure 9. In scenario S1, we simulate the SA-BSNs reliability degrading due to processing message delays: an unexpected number of patients are simultaneously using the system, flooding the communication channel of the Central Hub. The analyzer.py script generates Figure 9 that contains 540 seconds of execution with all the six sensors active, which was obtained by the artifact's default configuration. Furthermore, we can verify that the setpoint, in this case, was 90% reliability and the convergence value was 82%, which resulted in a steady-state error (SSE) of 8.84%. The maximum value reached was 112.25% of the convergence value that results in an overshoot of 12.25%. Finally, the settling time was 339 seconds.

V. Conclusion

In this paper we provide SA-BSN, an exemplar of the self-adaptive system in the healthcare domain. Our exemplar sheds light on control theoretical solutions for SAS by providing an environment with disturbance injection in the components that encode vital signs monitoring sensors and a central hub. Henceforth, we present the requirements that guided the SA-BSN implementation and the uncertainty scenarios available in the current version. Then, we discuss the implementation details in ROS and provide a walk-through for interested users.

ACKNOWLEDGMENT

The authors express their utmost gratitude to Carlos Eduardo Taborda Lottermann and to Léo Moraes for supporting the implementation of the SA-BSN and to the members of the LADECIC research group for the fruitful discussions. This study was financed in part by CAPES-Brasil – Finance Code 001, through CAPES scholarship, by CNpq under grant number 306017/2018-0, by University of Brasilia under Call DPI/DPG 03/2020, by the Wallenberg Al, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] B. Abidi, A. Jilbab, and M. E. Haziti, "Wireless sensor networks in biomedical: Wireless body area networks," in *Europe and MENA cooperation advances in information and communication technologies*. Springer, 2017, pp. 321–329.

[2] K. Guk, G. Han, J. Lim, K. Jeong, T. Kang, E.-K. Lim, and J. Jung, "Evolution of wearable devices with real-time disease monitoring for personalized healthcare," *Nanomaterials*, vol. 9, no. 6, p. 813, 2019.

[3] A. Reyes-Muñoz, M. C. Domingo, M. A. López-Trinidad, and J. L. Delgado, "Integration of body sensor networks and vehicular ad-hoc networks for traffic safety," *Sensors*, vol. 16, no. 1, p. 107, 2016.

[4] R. Gravina, P. Alinia, H. Ghasemzadeh, and G. Fortino, "Multi-sensor fusion in body sensor networks: State-of-the-art and research challenges," *Information Fusion*, vol. 35, pp. 68–80, 2017.

[5] A. H. Sodhro, L. Chen, A. Sekhari, Y. Ouzrout, and W. Wu, "Energy efficiency comparison between data rate control and transmission power control algorithms for wireless body sensor networks," *International Journal of Distributed Sensor Networks*, vol. 14, no. 1, p. 1550147717750030, 2018.

[6] D. Weyns, *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, 2020.

[7] A. Filieri, M. Maggio, K. Angelopoulos, N. d'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein *et al.*, "Software engineering meets control theory," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015, pp. 71–82.

[8] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *IEEE transactions on software engineering*, vol. 39, no. 11, pp. 1467–1493, 2013.

[9] Self-adaptive systems artifacts and model problems repository. Accessed 24-January-2021. [Online]. Available: https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/

[10] D. Weyns and R. Calinescu, "Tele assistance: A self-adaptive service-based system exemplar," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015, pp. 88–92.

[11] J. Wuttke, Y. Brun, A. Gorla, and J. Ramaswamy, "Traffic routing for evaluating self-adaptation," in *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2012, pp. 27–32.

[12] S. Gerasimou, R. Calinescu, S. Shevtsov, and D. Weyns, "Undersea: an exemplar for engineering self-adaptive unmanned underwater vehicles," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 83–89.

[13] P. H. Maia, L. Vieira, M. Chagas, Y. Yu, A. Zisman, and B. Nuseibeh, "Dragonfly: a tool for simulating self-adaptive drone behaviours," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2019, pp. 107–113.

[14] R. Caldas, G. Levi, S. Couto, E. Gil, C. E. Taborda, L. Moraes, G. Rodrigues, and J. Mendes, "SA-BSN first official release," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4620112

[15] A. Rodrigues, R. D. Caldas, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "A learning approach to enhance assurances for real-time self-adaptive systems," in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2018, pp. 206–216.

[16] A. Rodrigues, G. N. Rodrigues, A. Knauss, R. Ali, and H. Andrade, "Enhancing context specifications for dependable adaptive systems: A data mining approach," *Information and software technology*, vol. 112, pp. 115–131, 2019.

[17] G. F. Solano, R. D. Caldas, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "Taming uncertainty in the assurance process of self-adaptive systems: A goal-oriented approach," in *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '19. IEEE, 2019, pp. 89–99.

[18] R. D. Caldas, A. Rodrigues, E. B. Gil, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "A hybrid approach combining control theory and ai for engineering self-adaptive systems," in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 9–19.

[19] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[20] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "Morph: A reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, 2015, pp. 9–16.

[21] J. Cámara, A. V. Papadopoulos, T. Vogel, D. Weyns, D. Garlan, S. Huang, and K. Tei, "Towards bridging the gap between control and self-adaptive system properties," ser. SEAMS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 78–84.