Noname manuscript No.
(will be inserted by the editor)

Automatic Generation of Test System Instances for Configurable Cyber-Physical Systems

Aitor Arrieta · Goiuria Sagardui · Leire Etxeberria · Justyna Zander

Received: date / Accepted: date

Abstract Cyber-Physical Systems (CPSs) are ubiquitous systems that integrate digital technologies with physical processes. These systems are becoming configurable to respond to the different needs that users demand. As a consequence, their variability is increasing, and they can be configured in many system variants. To ensure a systematic test execution of CPSs, a test system must be elaborated encapsulating several sources such as test cases or test oracles. Manually building a test system for each configuration is a non-systematic, time-consuming and error-prone process. To overcome these problems, we designed a test system for testing CPSs and we analyzed the variability that it needed to test different configurations. Based on this analysis, we propose a methodology supported by a tool named ASTERYSCO that automatically generates simulation-based test system instances to test individual configurations of CPSs.

To evaluate the proposed methodology, we selected different configurations of a configurable Unmanned Aerial Vehicle, and measured the time required to generate their test systems. On average, around 119 seconds were needed by our tool to generate the test system for 38 configurations. In addition, we compared the process of generating test system instances between the method we propose and a manual approach. Based on this comparison, we believe that the proposed tool allows a systematic method of generating test system instances. We believe that our approach permits an important step towards the full automation of testing in the field of configurable CPSs.

Keywords Configurable Cyber-Physical Systems · Test System Generation · Test Automation

Aitor Arrieta
Goiru 2, 20500, Arrasate, Spain
Tel.: +34 685720978
E-mail: aarrieta@mondragon.edu

Goiuria Sagardui
E-mail: gsagardui@mondragon.edu
· Leire Etxeberria
E-mail: letxeberria@mondragon.edu
· Justyna Zander
E-mail: dr.justyna.zander@ieee.org

1 Introduction

Cyber-Physical Systems (CPSs) are defined by Lee & Seshia (2015) as "an integration of computation with physical processes whose behavior is defined by both physical and cyber parts of the system." The *physical layer* is composed of physical processes, which are a set of many parallel processes (Derler *et al.*, 2011) running in continuous time according to laws of physics (Lu *et al.*, 2015). The *cyber layer* is composed of computational platforms and networks which monitors and controls the physical processes usually with feedback loops (Derler *et al.*, 2011). *Computational platforms* consist of several sensors, actuators, embedded computers and embedded software. The *network fabric* provides communication mechanisms for the computer platforms.

Validation of CPSs is a difficult and time-consuming task. Simulation-based test systems combined with "in-the-Loop" test levels play an important role in the systematic validation of embedded systems (Zander-Nowicka, 2008). These systems contain algorithms (such as test oracles or test data generators) that help reduce the effort of the verification and validation stages. Furthermore, test systems may be implemented from the beginning of the product life cycle, allowing system testing at early development stages. However, proposals of test systems for embedded systems validation need to be adapted for the context of CPSs. CPSs are concurrent, which means that several tasks are performed simultaneously (Lee, 2008)(Derler *et al.*, 2011). Physical processes are parallel processes and their integration with computing requires a simultaneous composition of computing tasks and physical tasks (Lee, 2008). From the validation perspective this implies that the different parallel tasks have to be observed by the test oracle. Moreover, in the context of CPSs the physical world is not predictable and thus CPSs will not be operating in a controlled environment (Lee, 2008). This means that CPSs must be robust enough to withstand unexpected conditions (Lee, 2008), and must have high adaptive capabilities to dynamically reconfigure themselves (Shi *et al.*, 2011). This unpredictability has to be efficiently managed during the validation stages.

When CPSs have to be customized to clients demands, variability must be efficiently managed during all the development stages, which considerably increases the complexity of the system development and validation.[1] Configurable CPS development processes can be similar to those processes employed in product line engineering. In product line engineering, two main layers are considered: the domain engineering layer and the application engineering layer. The domain engineering layer involves different engineering tasks that consider variability of the product (i.e., variability in the product, in requirements, etc.). The purpose of the domain engineering layer is to create assets that will be reused in the application engineering layer. In the application engineering layer, the variability is resolved to create a specific system variant. The assets developed in the domain engineering layer are reused to create a variant, reducing the time for the development of different configurations.

Highly configurable CPSs can be configured into thousands or even millions of system variants and it is impracticable to test all possible configurations.[2] Thus, cost-effectively testing these systems is important to save time while achieving a high overall test quality. Simulation models that represent aspects such as system behavior, environment, structures and properties of CPSs are capable of rasing the level of abstraction at which testing is performed (Briand *et al.*, 2016). Employing simulation models permits software engineers to (1) execute more test cases, (2) develop test methods to select scenarios that should also be executed on the deployed system based on the risk level (e.g., fault revealing capability) and (3) specify test oracles for the automatic fault detection (Briand *et al.*, 2016). Moreover, simulation permits testing scenarios that could be dangerous, expensive or even impossible to reproduce employing a real prototype

[1] With variability we refer to configurability, i.e., variability in product space (Thiel & Hein, 2002).

[2] We denominate this term as system variant, configuration or system configuration, across the whole article.

(e.g., safety related scenarios). However, although the use of simulation methods permits several advantages, testing CPSs is still expensive. One of the reasons of testing a specific system variant in the CPSs domain is expensive is because manually generating a test system for a configuration is an error-prone, non-systematic and time consuming process. The use of a test system permits a systematic validation of simulation models by automating the execution and evaluation of test cases, reusing the most relevant test cases across the different development stages or employing test optimization algorithms to improve the test execution time while maintaining the overall test quality.

We highlight two main contributions: (1) a novel test system for testing CPSs employing simulation models and (2) a method for the automatic generation of test system instances for each configuration of configurable CPSs. For the first contribution, we have extended the test system proposed by Zander-Nowicka (2008) for configurable CPSs by supporting different methods for testing particularities of CPSs (i.e., concurrency, timing behavior and unpredictability). The test system for each configuration is automatically generated by the proposed tool, which is considered the second contribution of the paper. The tool obtains information of the variability parsing a feature model. After performing a set of model transformation the test system for testing a specific configuration is obtained. The tool allows full automation when generating the test system, what permits reducing the test system generation time as well as the error proneness.

This article extends our previous work-in-progress paper (Arrieta *et al.*, 2014) by (1) providing a completed tool together with a metodology that fully automates the generation of a test system, (2) providing a detailed metamodel of the designed test system specific for CPSs and (3) performing an evaluation of the tool with a case study.

The paper is structured as follows: after the introduction we give an overview of the background in Section 2. Section 3 proposes a case study of a quad-copter that will be used as an illustrative example, and to perform a validation of the proposed test system. Section 4 gives a general overview of the proposed methodology for the automatic generation of test systems in the context of configurable CPSs. The meta-model of the designed test system for the validation of CPSs is explained in detail in Section 5. Section 6 presents our approach for the systematic generation of test system instances for configurable CPSs. A complete evaluation of the proposed approach is presented in Section 7. Section 8 positions our work with other previous studies in the current state of the art. Finally, conclusion and future work are outlined in Section 9.

The reader is encouraged to visit the following webpage for further information of the developed tool: https://sites.google.com/a/mondragon.edu/asterysco/

2 Background

This section provides a brief background about CPS testing and basic concepts of configurable CPSs.

2.1 Cyber-Physical Systems Testing

Simulation is one of the driving forces for testing system models in domains where software interacts with physical processes and objects such as CPSs (Matinnejad *et al.*, 2016). As the development of a real prototype is often costly, the use of simulation for early validation and prototyping of CPSs has become a standard de-facto. Simulation models in the CPS domain are heterogeneous encompassing software, network and physical parts, and can represent an accurate representation of the real world and continuous dynamics (Mosterman & Zander, 2016)(Matinnejad *et al.*, 2016).

Figure 1 depicts the typical structure of a CPS involving the parts that have to be considered when simulating the system. In this case the example corresponds to an Unmanned Aerial Vehicle (UAV) we adapted from (Mosterman *et al.*, 2014), which is a model of a commercial drone. The *physical layer* of the system involves different physical processes: such as the dynamics of the UAV, its global position and the battery level.

As for the *cyber layer*, the example involves three different computational platforms connected by a network fabric. Models of computations, sensors, actuators and physical processes can be incorporated in simulation of CPSs (Jensen *et al.*, 2011). Regarding the example presented in Figure 1, each computational platform controls the drone at a specific level. Platform 1 performs the highest level control, which is planning the path of the UAV. Its sample time (i.e., when the embedded computer executes its tasks) is 100 ms. Platform 2 performs the low level control, whose objective is to maintain the UAV flying. It obtains data from the gyroscope every 10 ms and sends their speed to platform 3. Platform 3 is in charge of controlling the speed of each rotor, and it performs computations every 10 μs. The network that connects the three platforms is a CAN bus.

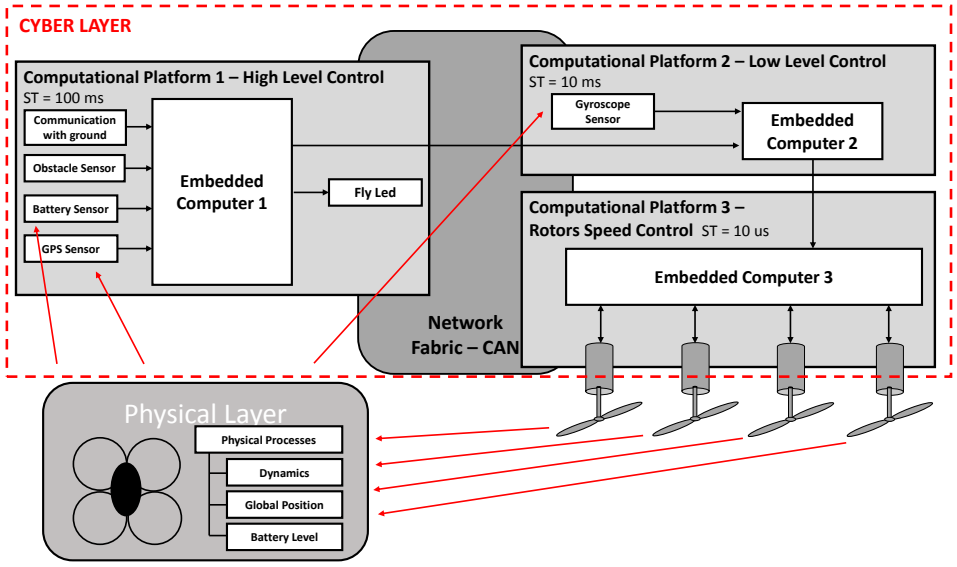

Fig. 1 Overview of the typical structure of a CPS, corresponding to an example of an UAV (based on (Lee & Seshia, 2015)). ST refers to the sample time of the computations to obtain data from sensors and compute their control tasks in each level

Apart from using commercial tools (such as MATLAB/Simulink, or LabView) for simulation and testing of CPSs, current studies also provide methodologies based on open source tools for virtual prototyping. For instance, Mueller *et al.* (2012) virtualized the cyber layer of the CPS employing C/C++ and System C whereas the physical layer of the system was modeled with Open Dynamics Engine. Other studies also focus on methodologies to design methods and simulation tools for the design of CPSs employing Model Driven Development techniques (Lu *et al.*, 2015).

When using simulation based testing, different testing configurations and test levels can be adopted (Shokry & Hinchey, 2009): Model-, Software-, Processor- and Hardware-in-the-loop (MiL, SiL, PiL and HiL) tests. The MiL configuration tests the model of the control system

together with the physical plant. The model of the control system is replaced with its software code for the SiL configuration. This software is compiled and embedded in the target processor for detecting errors related to the compiler in the PiL configuration (Shokry & Hinchey, 2009). The last step is to integrate the object code with the real Electronic Control Unit (ECU) and the real time infrastructure (e.g., Real-Time Operating Systems (RTOS), drivers, etc.), with the objective of validating performance requirements and timing constraints in a HiL configuration (Shokry & Hinchey, 2009). As temporal constraints are critical in the context of CPSs, HiL simulations are being adopted to test CPSs in real time. For instance, Eidson *et al.* (2011) proposed a programming and simulation model called PTIDES that allows CPSs co-design and co-simulation, where HiL simulation is facilitated. In the automotive domain, Kane *et al.* (2014) used test oracles to monitor high level critical properties employing a HiL simulation.

Test systems are often needed so that the verification and validation process of CPSs can be systematic. A test system can include several options, such as test oracles, components that generate input data in the form of test cases, and other testing resources. Kane *et al.* (2014) use external runtime monitors to detect violations employing a runtime verification technique in an automotive system. They obtain system data employing a passive bus monitor, and their oracle checks properties that were previously specified with a temporal logic language and state machines. Wan *et al.* (2011) presented a low cost hardware-based test platform for testing autonomous vehicles including wireless-sensor networks. A system that supports real-time operations for CPSs testing is provided by Tidwell *et al.* (2009). Their system allows for (1) specification of different experiment designs, (2) mapping of the execution of simulation model onto hardware resources, (3) interface with input and outputs of digital and analog devices and (4) support for strict temporal restrictions. Zander-Nowicka (2008) designed a test system composed of three main elements for conformance testing of embedded systems from the automotive domain, using a MiL configuration. In this case, the test system elements are the test data generator, test oracle and test control.

The use of formal verification techniques has improved considerably and might be effective for certain tasks, such as identifying potential deadlock conditions (Lee, 2007). However, if the properties are not formally specified, they cannot be verified (e.g., timing behavior of software) (Lee, 2007). According to Matinnejad *et al.* (2016), formal specifications are rare in practice, as it is challenging to capture continuous dynamics of CPSs. In addition, formal verification is not carried out by system designers, but by verification experts. Moreover, scalability to realistic systems remains a major issue (Lee, 2007).

2.2 Variability in Configurable CPSs

In Figure 2 we provide a taxonomy with the variability that our approach can deal with. Consider the UAV presented in the previous section (Figure 1) as an example of a configurable CPS. The example consists of a model of a commercial drone (Mosterman *et al.*, 2014), and we include some variability points. This UAV has different variability points based on user needs.

As for the physical layer, we consider variability in several points. The mechanical elements are highly exposed to variability, which has to be taken into account since a change in a mechanical element might change the dynamics of the system. For instance, the UAV might have different shapes and sizes, which has a direct impact on its weight as well as on the center of gravity of the system. Changing both these factors in turn affects the dynamics of the CPS, which can lead to a completely new behavior. Another variability point in the physical layer consists of the energy supply system. Energy supply in CPSs is a challenge and often the system must optimize its consumption (Wan *et al.*, 2011). For instance, in the example of the UAV two battery models

are provided: a short duration battery and a long duration battery. However, note that a long duration battery might be bigger and heavier, which can have a direct influence in the dynamics of the system.

Regarding the cyber layer, we consider variability in several points too. In the software of the CPS variability can appear in different functionalities of the configurations. In the example provided, an optional functionality is the obstacle detection system. Another variability that has to be considered is related to the sensors (variability in points such as sensitivity, ranges they can measure, response time, parametric values, etc.) and actuators (such as communication, ranges they can work in, response time or robustness properties, etc.). In addition to sensors and actuators, variability in the number of platforms and in the network fabric in charge of communicating the computational platforms is also supported in our approach. Although the provided example does not consider variability in the platforms, variability can appear for instance in the number of platforms, the way they are interconnected and the sample time in which each platform performs computations. Concerning the network fabric, different types of communications can be considered (e.g., Ethernet, CAN bus, 802.11b WLAN, etc.).

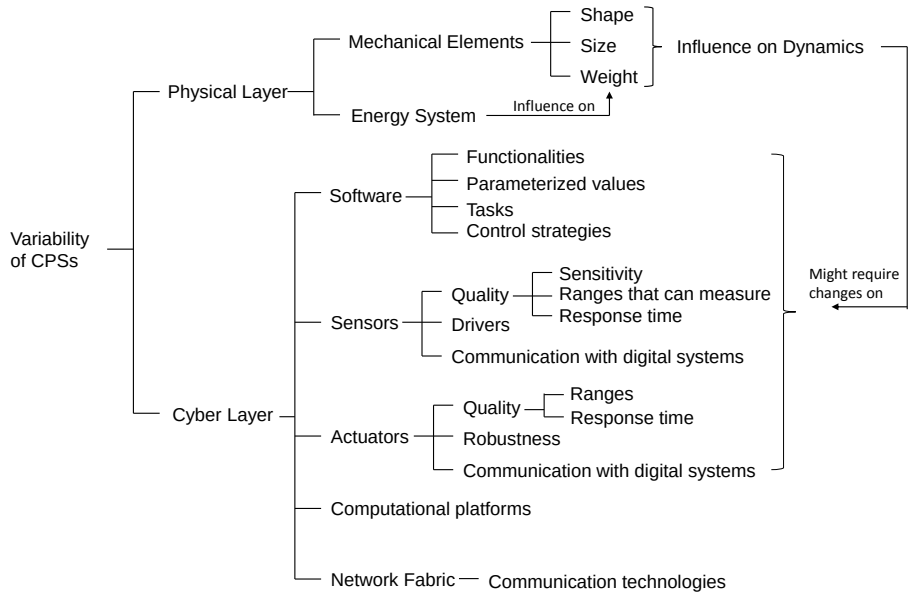

Fig. 2 Classification of the variability of configurable CPSs

It is important to highlight that the variability of some parts might require considering variability in other parts of the system. For instance, the variability related to the physical layer might have a direct influence on the dynamics of the system. This issue might also require some changes in the cyber layer, where configurations in the software (such as new functions or changes in certain parametric values), or even in the hardware (such as sensors with better sensitivity or actuators that can cope with heavier weights) might be needed. Some functionalities, such as the example of obstacle detection, might also require changes in both the software as well as the hardware. Moreover, there are specific hardware elements that require the software to be adapted. For instance each specific sensor needs its driver and each actuator might require

a specific controller. In the case of the rotors in our example, there are some software control parameters that must be adapted depending on the type of rotors used in the configuration (i.e., high or low speed rotors).

When testing variability-intensive systems, the number of configurations that the system can be set to is a factor that has to be considered at validation stages. This number is usually too high, and as a result, it is infeasible to test all the configurations. For this reason, it is important to optimize as much as possible the whole test and validation process. The following steps need to be taken when testing variability-intensive systems, such as configurable CPSs:

- Generation of relevant configurations to test: as testing all the configurations is infeasible, it is important to select relevant configurations that can ensure certain test coverage for the whole variability-intensive system. The most common approach for selecting these configurations is employing Combinatorial Interaction Testing (CIT) algorithms (Kuhn *et al.*, 2009). These algorithms usually parse a feature model and generate configurations that cover all possible feature interactions using SAT solvers (Perrouin *et al.*, 2010). For instance, a pairwise configuration generation criteria will ensure that all the feature pairs of the variability-intensive system are covered. This way, it can be ensured that the system does not fail due to the interaction of two features.
- Prioritization of the order in which the configurations are tested: once the relevant configurations are generated, it is also important to prioritize the order in which these are tested. For instance, Sánchez *et al.* (2014) demonstrated that in the context of Software Product Lines, the product complexity helped increase the fault detection rate as compared to other prioritization criteria (e.g., configuration size or dissimilarity of configurations).
- Generation of the test system: Once selected the configuration that must be tested, the test system that includes test cases, test oracles and other sources must be generated. Manually generating the test system can be a time consuming and error-prone process, and thus, an automatic test system generation is needed to ensure a systematic generation of the test system. The contribution of this paper is focused on this point.
- Test case selection and prioritization: Taking into account the selected configuration it is important to select which are the test cases that must be executed (Wang *et al.*, 2013a)(Wang *et al.*, 2016). Studies in the field of Software Product Line engineering have also proposed several search algorithms for test suite minimization (e.g., (Wang *et al.*, 2013b)(Wang *et al.*, 2015)). In addition, after selecting relevant test cases, test case prioritization helps improve the fault detection rate or reduce simulation time (Wang *et al.*, 2014)(Arrieta *et al.*, 2015).

2.3 Basic Concepts

Let $CCPS$ be a configurable CPS with a set of configurations named *cps*, i.e., $CCPS = \{cps_1, cps_2, ..., cps_{ncps}\}$, where *ncps* is the number of possible configurations that the system can be set to (Wang *et al.*, 2015). F is the feature model corresponding to the $CCPS$, which is composed of nf features (f): $F = \{f_1, f_2, ..., f_{nf}\}$ (Wang *et al.*, 2015). The inputs and outputs of a system play an important role in black-box testing, as the test system usually stimulates the inputs and a test oracle monitors the outputs to see the expected behaviours with respect to the inputs. The $CCPS$ will have a set of Inputs, $I = \{i_1, i_2, ..., i_{ni}\}$, and a set of Outputs, $O = \{o_1, o_2, ..., o_{no}\}$, where *ni* and *no* are the number of inputs and outputs, respectively. Usually, depending on the functionalities needed by the user, a set of features will be selected from the feature model, and the inputs and outputs will be associated with these features. R is a set of requirements assigned to $CCPS$, composed of nr requirements, $R = \{r_1, r_2, ..., r_{nr}\}$. The relation

among features, inputs, outputs and requirements are managed with cross-tree constraints in the variability model.

The features corresponding to a specific configuration from $CCPS$ are represented by $F_{cps_i} = \{f_1, f_2, ..., f_{nf_{cps_i}}\}$, where F_{cps_i} is a subset of F, f_j can be any feature in F ($f_j \in F$) and nf_{cps_i} is the number of features representing cps_i, where $1 \leqslant nf_{cps_i} \leqslant nf$ (Wang *et al.*, 2015). $I_{cps_i} = \{i_1, i_2, ..., i_{ni_{cps_i}}\}$ and $O_{cps_i} = \{o_1, o_2, ..., o_{no_{cps_i}}\}$ are a subset of inputs and outputs corresponding to cps_i, where $i_j \in I$, $o_j \in O$ and $1 \leqslant ni_{cps_i} \leqslant ni$, $1 \leqslant no_{cps_i} \leqslant no$. In addition, cps_i will have a subset of requirements, $R_{cps_i} = \{r_1, r_2, ..., r_{nr_{cps_i}}\}$, where r_j can be any requirement from R ($r_j \in R$), and $1 \leqslant nr_{cps_i} \leqslant nr$.

When testing cps_i, the test system has to be configured to test the requirements corresponding to the configuration, i.e., R_{cps_i}. In addition, the software, the inputs (I_{cps_i}) and outputs (O_{cps_i}) have to be configured for cps_i. The test system has to be automatically configured to stimulate the inputs and to observe the outputs of cps_i, which may vary from configuration to configuration.

3 Case Study

The AR.Drone is an Unmanned Aerial Vehicle (UAV) developed by Parrot for the market of video games and home entertainment (jean Bristeau *et al.*, 2011). Mosterman et al. modelled the dynamics of the AR.Drone using experimental input-output data, the structure of the vehicle, equations of motion and system identification techniques (Mosterman *et al.*, 2014). The mathematical models of the AR.Drone dynamics were previously validated with the real vehicle in (Mosterman *et al.*, 2014), where an accurate position estimation was demonstrated.

We reused the AR.Drone model developed by Mosterman *et al.* (2014) and added different variability points in terms of extra functionalities as well as variability points in hardware and software units (Subsection 3.2). In addition, as we added new functionalities, functional requirements were re-written (Subsection 3.3).

3.1 System Architecture

The cyber layer of the UAV is composed of three different platforms, as depicted in Figure 1. Each platform controls the UAV at a specific level. Platform 1 corresponds to the layer performing the high level control of the system. This platform is in charge of deciding the path that the UAV must follow. For this task, the platform integrates three sensors (obstacle sensor, battery sensor and GPS). It is also the platform in charge of communicating to the ground station. Based on the user's request, the embedded system obtains positional data via the GPS as well as information from the outside environment (e.g., if there is any obstacle) and the path to follow is plotted. It also integrates the "FlyLed" that is turned on while the UAV is flying. Platform 2 makes a low level control to ensure that the UAV keeps flying. It obtains data corresponding to the dynamics of the system with the gyroscope sensor and computes a set of control algorithms to regulate the speed of each rotor so that the system keeps stable. The speed of each rotor is stabilized by sending a set of speed commands to Platform 3, which is in charge of controlling the speed of each rotor. This platform performs the lowest level control.

3.2 Variability of the AR.Drone UAV

Feature modeling is an approach used to hierarchically capture commonalities and variabilities in a configurable system (Benavides *et al.*, 2010). The features can be categorized as mandatory

(if there is a black circle at the top of the feature) or as optional (if there is a white circle at the top of the feature). When there are sub-features, they can also be categorized as alternative or as "OR." The identified variability has been classified in the two main layers that a CPS can have, i.e., the cyber and physical layers. Figure 3 and 4 depicts the feature models used to manage the variability of the AR.Drone model.[3] As for the software, there are two mandatory features: (1) track system and (2) control. The former refers to how the UAV plans its path. As compared to the model developed by Mosterman *et al.* (2014), we included an extra path planning strategy, where the UAV can follow a person from different perspectives, i.e., from the left, right, front and back (as specified by the user). The latter refers to the control strategy. The control strategy proposed in (Mosterman *et al.*, 2014) consisted of a set of proportional controllers for the position, height, forward and lateral velocity and heading angle. In our case, we give the user the option to choose between a proportional or a proportional-integral control strategy.

In addition, we included some optional features in the software. We added some safety functions: collision avoidance equipment, wind avoidance algorithms, an emergency system and a strategy to independently return home. We also added some functionalities regarding the battery management. We developed two battery models: a battery lasting around 12 minutes, and another one lasting around 25 minutes. In addition, to prevent the drone from losing battery power, the vehicle could either launch the landing program, or it could find the closest battery station to recharge the battery. As for the RTOS, in our modified system, we gave the option of using RTOS or not. If the user chooses to use a RTOS, embedded linux or FreeRTOS (FreeRTOS, 2014) can be used. By default, the AR.Drone uses an embedded linux operating system (jean Bristeau *et al.*, 2011).

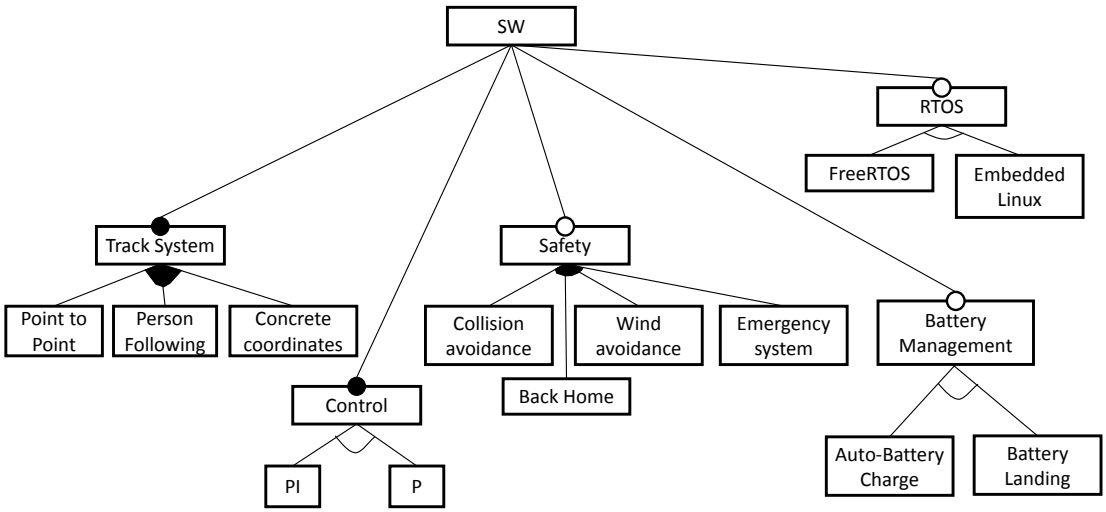

Fig. 3 Feature model for the configurable AR.Drone software

From the hardware perspective, we also proposed some variability points, allowing the user to choose different components. Concerning the sensors, four types of sensors could be used in

[3] For presentation purposes of the article, two independent feature models have been developed.

the drone: Gyroscope and GPS, which are mandatory for position and vehicle state estimations, Obstacle Sensor, which is only employed if the collision avoidance strategy is chosen, and battery sensor, which is mandatory if the user has chosen the automatic management of the battery. For each type of sensor, the user can choose among three different models. With regard to actuators, we proposed other rotors for higher speeds; thus, the user can choose between low or high speed rotors. In addition, a LED that indicates that the vehicle is flying was added as an optional feature.

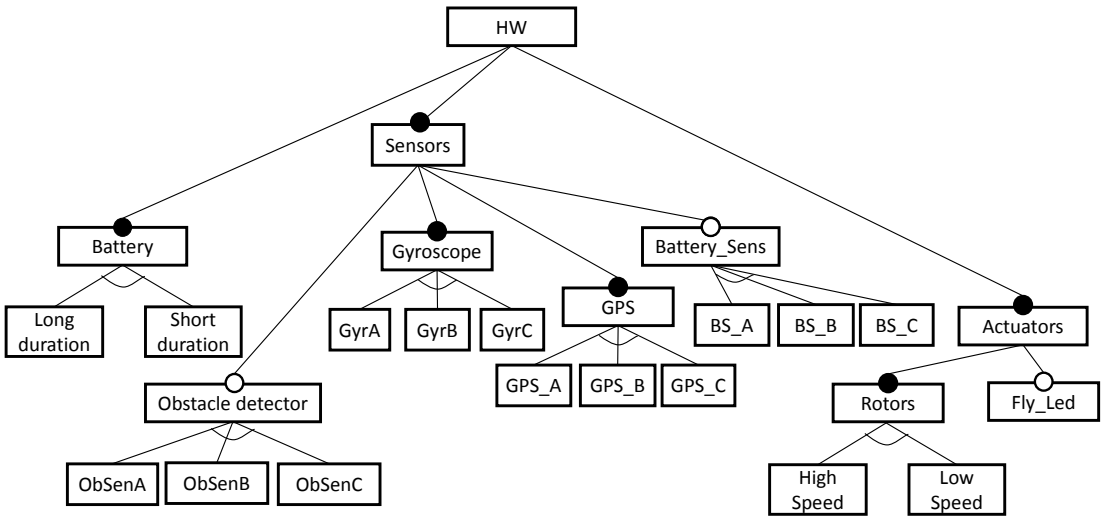

Fig. 4 Feature model for the configurable AR.Drone hardware

3.3 System Requirements

A total of 20 requirements were defined. There are certain requirements that are mandatory (Table 1) whereas others are optional requirements (Table 2). Mandatory requirements are those related to mandatory features. For instance, r_4 is related to the time that the UAV must fly, which is directly covered by the battery. In the same way, optional requirements are those related to optional features. As an example, r_6 is related to the wind avoidance, which is an optional functionality; as a result, if the chosen configuration does not include this functionality, r_6 will not be inside the requirements of the chosen configuration.

In addition, some requirements handled variability (i.e., r_1, r_2, r_3, r_4 and r_7). For instance, r_1 refers to the maximum vertical speed of the UAV. However, two kinds of rotors can be chosen by the user: high speed rotors and low-speed rotors. Thus, the maximum allowed vertical speed will not be the same for high or low-speed rotors. Therefore, $r_1 = \{r_{1a}, r_{1b}\}$, where r_{1a} specifies the maximum vertical speed for system variants using low-speed rotors, whereas r_{1b} specifies the maximum vertical speed when high speed rotors are chosen for the configuration.

Table 1 Mandatory functional requirements for the CPS involving the configurable UAV

Reqs	Sub-reqs	Features Involved	Requirement
r_1	r_{1a}	Rotor A	The maximum vertical speed for the drone shall not exceed 1 m/s
	r_{1b}	Rotor B	The maximum vertical speed for the drone shall not exceed 3 m/s
r_2	r_{2a}	Rotor A	The maximum horizontal speed for the drone shall not exceed 3 m/s
	r_{2b}	Rotor B	The maximum horizontal speed for the drone shall not exceed 5 m/s
r_4	r_{4a}	Battery A, Battery Control	The flight program shall last at least 12 minutes
	r_{4b}	Battery B, Battery Control	The flight program shall last at least 25 minutes
r_{10}	r_{10}	Gyroscope, Controllers	When the speed is around 0 m/s, the pitch and roll euler angles of the drone shall not exceed 20 degrees (deg)/100 meters height
r_{11}	r_{11}	Gyroscope, Controllers	The yaw angle shall be the one selected by the drone driver with a maximum error of 15 degrees (deg)/100 meters high
r_{16}	r_{16}	Communication	The drone shall not fly if the command for flying is not activated
r_{17}	r_{17}	gps	The drone shall not move horizontaly if it is placed in the floor, i.e., h ==0
r_{18}	r_{18}	gps	The drone shall not move rotationally if it is placed in the floor, i.e., h ==0
r_{19}	r_{19}	Communication	The drone shall land if the communication with the base station has been lost

4 Overview of the Approach

The overview of the approach considered in this study to automatically generate test system instances is highlighted in Figure 5. It represents several points that test engineer must consider when using the tool we propose. The first point corresponds to the variability management of both the configurable CPS and the test system. In our case, FeatureIDE (Thuem *et al.*, 2014) is used for this task. It is important to systematically document the requirements of the system, highlighting the test cases employed to test them. In our approach, this is addressed in the second point, where we employ the tool Doors to capture all the requirements of the configurable CPS and tracing them with test cases. Once the domain requirements of the system are specified, in a third step, system engineers develop a 150% model of the configurable CPS in Simulink. 150% models address all the variability that the system can have, which is later pruned by a system configurator. In the fourth point, engineers develop the infrastructure including test assets that will be reused at the application engineering layer for testing the configurable CPS. This infrastructure considers executable test cases, different requirement monitors and a context environment employed for simulating the environment in which the CPS operates. The fifth step corresponds to the selection of configurations to test. Most of the research in product line testing has focused selecting relevant configurations, taking into account the variability that the product

Table 2 Optional functional requirements for the CPS involving the configurable UAV

Reqs	Sub-reqs	Features Involved	Requirement
r_3	r_{3a}	Battery A, Battery Control	The drone shall begin the landing if the battery is less than 20%
	r_{3b}	Battery B, Battery Control	The drone shall begin the landing if the battery is less than 15%
r_5	r_5	Emergency system	The drone shall run the landing program if there is an emergency
r_6	r_6	Wind avoidance	The drone shall run the landing program if there is too much wind
r_7	r_{7a}	BatteryA, Battery Control, autobattery, GPS	The drone shall displace to the closest battery station if there is less than 30%
	r_{7b}	Battery B, Battery Control, autobattery, GPS	The drone shall displace to the closest battery station if there is less than 25%
r_8	r_8	AutoHome, GPS	The drone shall come back home if asked
r_9	r_9	Collision Avoidance	If there is an obstacle, the drone shall detect it in less than 0.5 seconds and be able to skip it independently
r_{12}	r_{12}	Point to point, GPS	The drone shall follow all the points with a maximum error of 40 cms
r_{13}	r_{13}	ConcreteCoordinates, GPS	The drone shall achieve the selected coordinates with a maximum error of 40 cms
r_{14}	r_{14}	PersonFollowing, GPS	The drone shall follow a person to 3 meters of distance from the horizontal coordinates, with a height indicated by the user
r_{15}	r_{15}	PersonFollowing	The drone shall land if the contact with the person to follow has been lost due to the distance
r_{20}	r_{20}	Flying Led	The drone shall turn on the led while it is on the air

can have (typically with a feature model). The objective is to reduce as much as possible the number of configurations to test (Lopez-Herrejon *et al.*, 2015). This is important due to the fact that testing each configuration is costly (Wang *et al.*, 2015). Current trends involve CIT techniques (e.g., t-wise testing) to generate relevant configurations.

Generation of test systems is performed in the sixth step in the proposed approach. This part is explained in detail in Section 6 of this paper. Manually generating a test system for each configuration is infeasible and thus, an automated solution is needed. For this we propose ASTERYSCO (Automatic Simulation-based TEst system geneRator for cYberphyscal Systems COnfigurations), a tool that automatically generates test systems to test the different system variants that a highly configurable CPS can be set to. ASTERYSCO (1) processes information of the feature model as well as information of each specific configuration to test, (2) obtains and configures the test cases, requirement monitors and the context environment functions needed to test the configuration and (3) generates the test system for the chosen configuration in Simulink. The generated test system is in accordance with a previously designed test system. The metamodel of the designed test system for testing CPSs is presented in Section 5. Our approach is

designed for early validation of configurable CPSs at system level as testing starts very early when simulation models are available (Matinnejad *et al.*, 2016). In a seventh step, the test cases for each configuration are executed in Simulink employing the test system that is automatically generated with our tool. The eighth step in Figure 5 consists of the analysis of the results of the tests by system engineers so that they can fix faults in the configurable CPS.

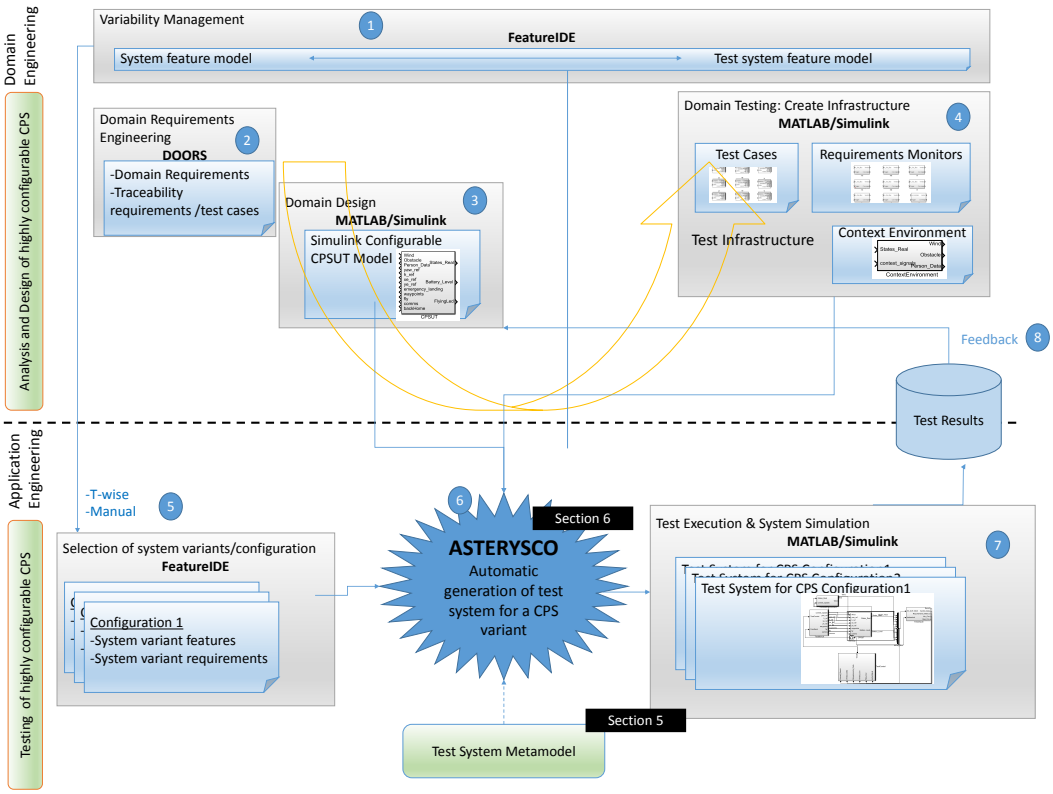

Fig. 5 Technological overview of ASTERYSCO and its dependencies

5 Test System for CPS Validation

This section explains the test system for the early validation of configurable CPSs. We took the test system proposed by Zander-Nowicka (2008) as a base. Nevertheless, this system was oriented for embedded systems from the automotive domain as well as for non-configurable systems. Thus, we extended that test system for testing configurable CPSs.

5.1 Test System Meta-model

The highest level of the metamodel for our test system, which is illustrated in Figure 6, is composed of five different elements:

- Cyber-Physical System Under Test (CPSUT): The system to be tested. In our case, the CPSUT is a model composed of the cyber and the physical layer.
- Test Stimuli: The source in charge of generating the necessary data to stimulate the CPSUT and to control part of the behavior of the context environment.
- Test Oracle: Observes the behavior of the CPSUT with respect to its inputs and decides whether the results fulfill functional requirements or not.
- Test Control: The source that control the execution of test cases and decides when the test must be finished.
- Context Environment Model: Simulates the physical world in which the CPS resides and has to interact with.

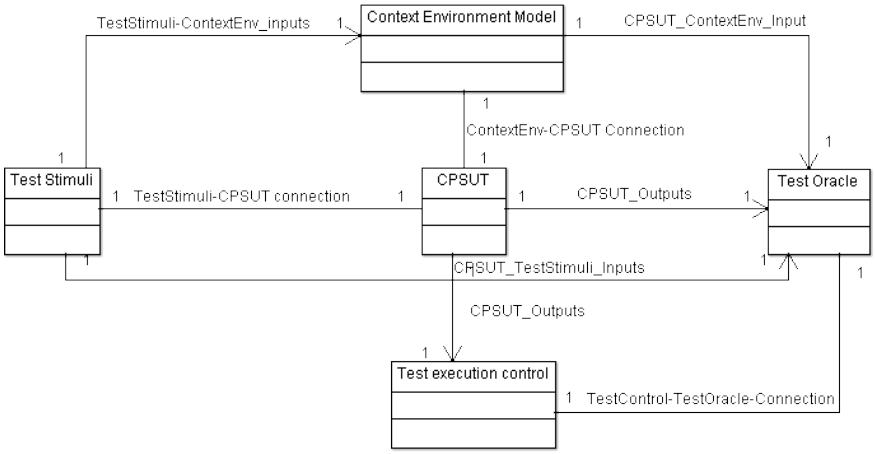

Fig. 6 Test system meta-model. The test stimuli source stimulates CPSUT, the context environment model simulates the context in which the CPSUT resides, the test oracle observes the behavior of the CPSUT and decides the test result and the test execution control sequences the test cases.

5.1.1 Test Stimuli

Figure 7 depicts the overview of the test stimuli block. This block encapsulates the different test cases for validating the system requirements. A test case refers to a set of signals that stimulate the inputs of the system. Each functional requirement is validated with a set of test cases (unlike in (Zander-Nowicka, 2008), where each requirement is validated with a single test case). An algorithm named Test Trigger Control sequences the test case execution for each requirement. We also provide support for reactive test cases, which are test cases that observe the state of the CPS and take decisions accordingly. Reactive test cases are very common in CPSs as they support the unpredictability of the physical world by observing the state of different parts of the CPS. "FeedbackSignals" contain signals related to the state of different parts of the CPS. Take as an example a test case that sets the speed of a direct current engine. The engine has to achieve 500 revolutions per minute (rpm) and once this speed is achieved, the speed has to be set to 100 rpm. The engine observes via the feedback signals the speed of the engine, and once it achieves 500 rpm, it sets the speed of the engine to 100 rpm.

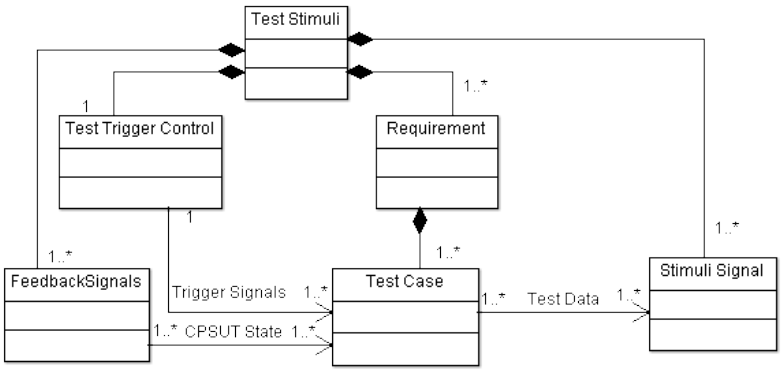

Fig. 7 Test stimuli meta-model of the test system employed to send the generated test data to the CPSUT

5.1.2 Test Oracle

The main structure of the test oracle is depicted in Figure 8. The test oracle observes the behavior of the CPSUT and decides whether the system fulfills functional requirements. The test oracle examines each requirement independently using one or more requirement monitors. For starting the validation of a specific requirement, the block named "RequirementTriggerControl" encapsulates an algorithm that triggers each requirement. A requirement monitor is a piece of software that determines the requirement status (pass, fail, inconclusive) taking into account a stream of significant input events (Robinson, 2010). We employ one or more requirement monitors to assess the concurrency of the CPS. A requirement monitor is composed of a precondition and a post-condition. The precondition specifies when a requirement monitor can evaluate a requirement, whereas the post-condition evaluates whether the requirement is valid or not. Each post-condition consists of a property verifier and a verdict generator. On the one hand, the property verifier observes that the state of the system is correct. On the other hand, the verdict generator evaluates whether the properties for the requirement meets the specified requirements. When the verdict generator evaluates the validity, it generates a verdict called Requirement-Monitor Verdict (RMVerdict), which is processed by the requirement validator. The data generated by the requirement validator is sent to the arbiter. The arbiter processes and analyses the verdicts of each requirement, and generates a verdict that represents the result of the executed tests. The arbiter is composed of the coverage calculator and the validation algorithm, which are employed to give the overall test result. First, the coverage calculator calculates the amount of test coverage the tests have achieved (i.e., in our context, the number of requirements that have been covered). This data is transferred to the validation algorithm to give the result of the overall test.

5.1.3 Context Environment

A CPS works inside an environment. Two kind of environments can be differentiated: (1) the context environment, and (2) the irrelevant environment. The former refers to the environment that directly affects the CPS, thus, engineers must take this environment into account when validating CPSs. For example, in the presented case study, the context environment includes a function that models the wind, another function that models the objects as well as other kinds of hazards that the drone can be exposed to and last, the position of the person that must follow

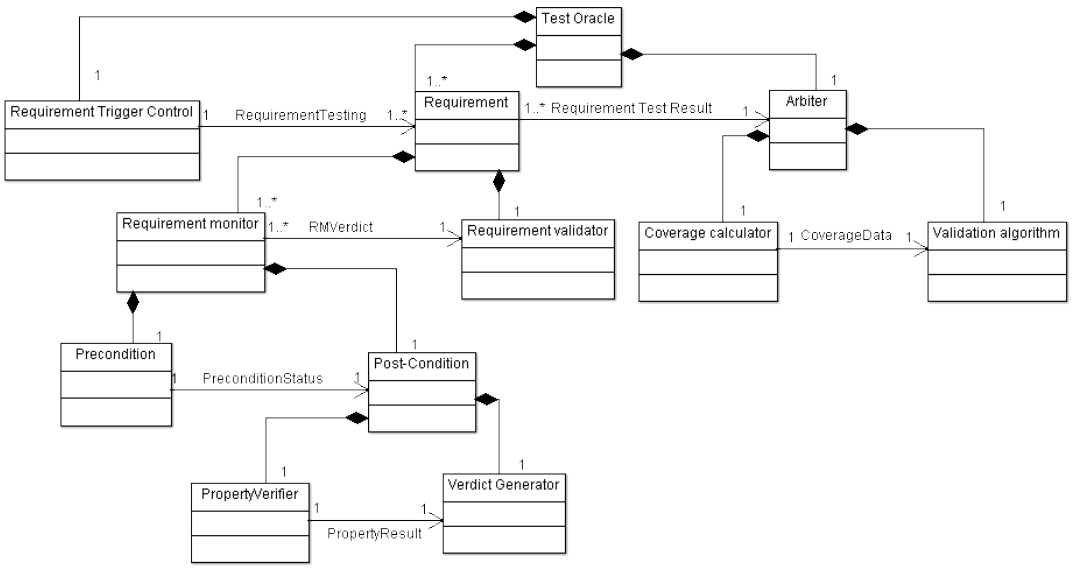

Fig. 8 Test oracle meta-model of the proposed test system for the decision of the test result

(if that feature is selected). The irrelevant environment refers to the environment that does not affect the CPS, even though it is operating within it.

The context environment is included in our test system to deal with the unpredictability of the physical world. Figure 9 depicts the metamodel of this part of the test system. The context environment models different scenarios, called context environment functions, in which the CPS can be involved. There can be one or more context environment functions. As mentioned in the previous paragraph, we model the wind, the obstacles that the CPS has to face and in the case that the UAV is configured to follow a person, the position of that person. The context environment communicates with the TestStimuli source (Stimuli Outputs). This is important as the test stimuli sends orders to the context environment (e.g., generate heavy wind). In addition, the context environment also obtains data related to the CPS via the CPSUT outputs (e.g., in the provided example, the altitude of the drone is obtained, so that as the drone gains height, the wind is heavier). After processing this data, the context environment sends data that can affect the CPSUT (such as wind or obstacle) via the "CPSUTInput."

5.1.4 Test Control

The test control algorithm we defined is quite different to Zander's test system. In (Zander-Nowicka, 2008) the test cases were executed following a time-, logical condition-, and verdict-triggered strategy. In contrast, to support the cost-effectiveness that the configurable CPSs demand when testing, we have divided it into three main parts: the Next Test Generator (NTG) algorithm, the Test ID Generator (IDGen) algorithm and the Test Ending Mechanism. The NTG algorithm decides **when** to execute a new test case, i.e., its main control is to detect when the test that is being executed has finished testing the CPS Under Test (CPSUT). When a new test case has to be executed, the NTG algorithm sends a boolean signal to the IDGen algorithm. The IDGen algorithm decides **which** test case to execute. Finally, a subsystem named "end mech-

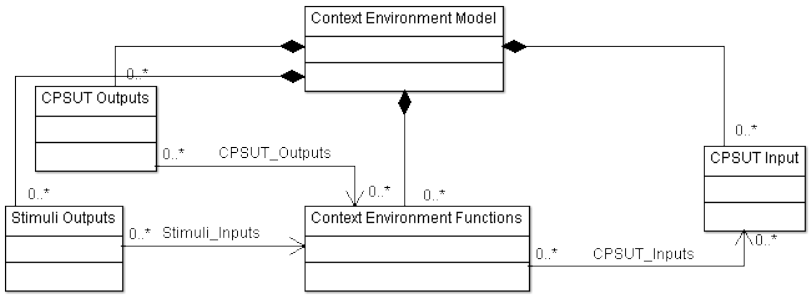

Fig. 9 Context environment metamodel of the test system for the simulation of the environment in which the CPS operates

anism" decides when to finish the test. Figure 10 depicts the architecture and the interactions of the different sources in charge of controlling the test at simulation level. For more information about different test control strategies for the cost-effective validation of CPSs refer to our previous work(Arrieta *et al.*, 2015).

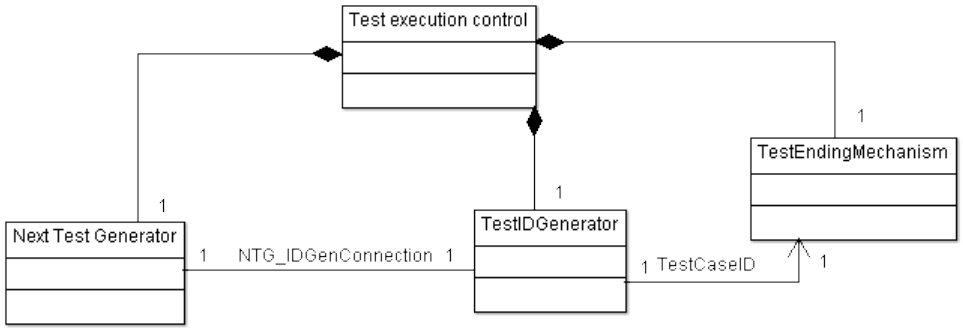

Fig. 10 Test execution control metamodel proposed to control the execution of the test cases, the block to the left belongs to the NTG algorithm, the one in the middle to the IDGen algorithm, and the one in the right to the "End mechanism"

5.2 Summary

Although some parts have been inspired by the previous work of Zander-Nowicka (2008), we extended different parts of the test system for testing particularities of configurable CPSs. CPSs are concurrent systems, which means that several tasks are performed simultaneously. To support concurrency, the test oracle we propose monitors each requirement with one or more requirement monitor. Each requirement monitor is capable of observing a specific task of the CPS. The unpredictability of the physical world is another particularity of CPSs. Our test system supports the testing of this issue by (1) including a new block named Context Environment and (2)

employing reactive test cases. The former models the physical world in which the CPS resides simulating different scenarios. The latter refers to the test stimuli block, which encapsulates different test cases capable of observing different states of the system and taking decisions based on them. Finally, one particularity of configurable CPSs is the need for cost-effectively testing. This is because many configurations have to be tested and simulating each configuration is expensive. Our test system supports cost-effectiveness by employing an efficient test control strategy that can be combined test suite minimization and test case prioritization.

6 ASTERYSCO: Automatic Test System Generator

This section explains the approach to automatically generate test system instances for each system variant. Figure 11 depicts the overview of the processes performed by the developed tool, named ASTERYSCO, for the automatic generation of the test system in a Software Process Engineering Metamodel (SPEM). The system takes a set of external files as input. The output result of the process is a test system ready to test the specified configuration of the CPS.

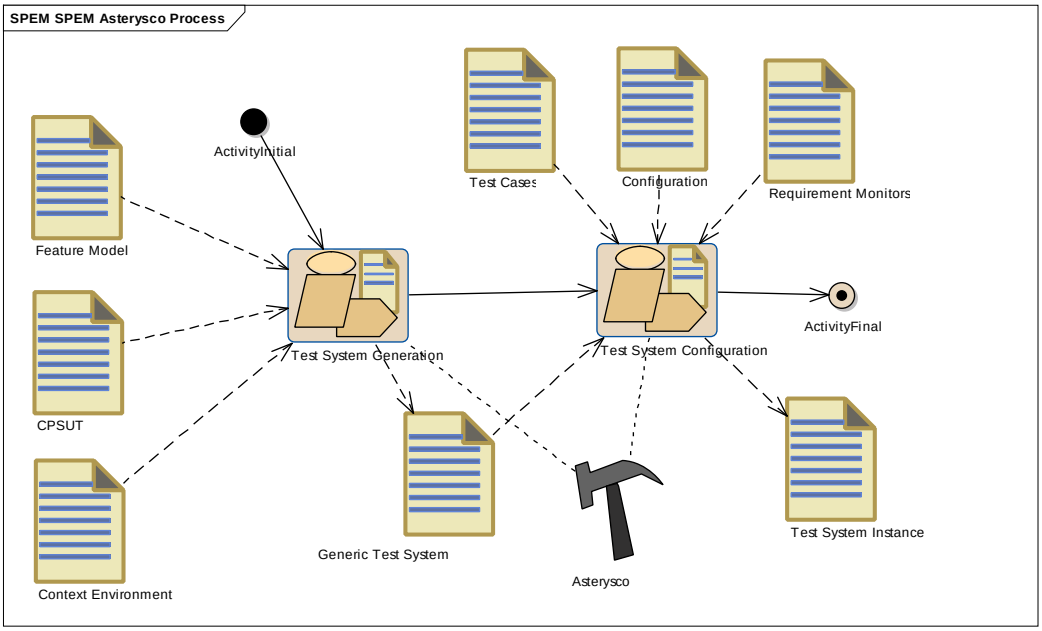

Fig. 11 Overview of ASTERYSCO's activities for the generation of a test system instances in SPEM. The arrows with the solid line indicate the control flow, whereas the arrows with the dashed lines indicate the object flow (i.e., inputs and outputs of each process). In this case, the first activity is the test system generation, and the second activity corresponds to the configuration of the test system for a specific system variant. In the first process the inputs are the feature model, the CPSUT and the context environment; the output of the first process is a Generic Test System, which is an input to the second process. The rest of the inputs for the test system configuration process are the test cases, the configuration file and the requirements monitors. The output of this process is the test system instance for the selected configuration.

The test system we choose is the one explained in Section 4. The generated test system is a Simulink model. Simulink was chosen because it is a prevalent modeling language for CPSs (Matinnejad *et al.*, 2016). It allows simulation of heterogeneous models, encompassing software, network and physical units. Moreover, it allows different variability mechanisms (Dziobek *et al.*, 2008)(Polzer *et al.*, 2012). In addition, Simulink supports CPSs modeling by employing different toolboxes for computer vision and signal processing, concurrency modeling of computing platforms, graphical state machine modeling, system-level physics modeling, communication and computation modeling and in-the-loop technologies (MathWorks, 2016). Feature modeling, which is widely used in industry (Berger *et al.*, 2013), was employed for variability management.

For generating different test system instances it is important to analyze the variability points of the test system. Depending on the chosen system variant or test objective, the test system has to be adapted and automatically configured. To achieve this goal, the different elements of the test system have to deal with variability.

The variability of the Test Stimuli strongly depends on the selected product configuration: variability can be found in the number and type of requirements, and as consequence in the test cases in charge of testing these requirements. Depending on the number of test cases needed to test a specific system variant, the test control source must also be adapted. The inputs of the CPSUT can vary from system variant to system variant, and therefore, the stimuli signals have to be updated. In addition, the test cases are reactive, which means that they act taking into account different signals and variables of the CPSUT. Depending on the system variants, these signals can also vary.

The test oracle also depends on the selected system configuration. Variability in this system can be found in the number and type of requirements. As a consequence, the signals of the test trigger control must be adapted. Furthermore, each requirement might also have one or more requirement monitor, which have to be configured for the selected system variant, thus, variability can also appear in the signals of the precondition or in the signals of the property verifier. The final verdict is given by the arbiter, which has to be adapted to the number of requirements in the system. Variability can also be found in this arbitration algorithm, as two algorithms were developed with different test objectives (i.e., validation based on the percentage of requirements covered or validation based on the obtained verdicts). In addition, the test oracle has to be adapted to the input and output signals present in the CPSUT system variant.

With regard to the context environment, depending on the selected system configuration, there are some functions that are inside the context environment and others inside the irrelevant environment. The functions that are inside the irrelevant environment of the selected configuration have to be removed from the test system when configuring it with the objective of saving simulation time.

Two main steps are taken to generate a test system for a specific configuration: (1) generation of the generic test system and (2) instantiation of the generic test system for a selected configuration. In the first step, the test system generator reads the variability management file, and a generic test system is automatically generated. This test system generator uses test system specific algorithms (e.g., arbitration algorithms, test control algorithms, etc.) that have been developed and stored in a Simulink library, and are the same for any system. In addition, it also takes the model of the CPSUT as well as the model related to the context environment in which the CPSUT operates. Once the generic test system is generated, this process is not needed to be performed again (unless there are modifications such as requirement changes, etc.).

In the second step, the generic test system is reused by the test system configurator, which automatically rebuilds it to obtain the test system compliant with the system variant. In this step, the test system configurator replaces the elements of the test system with the elements

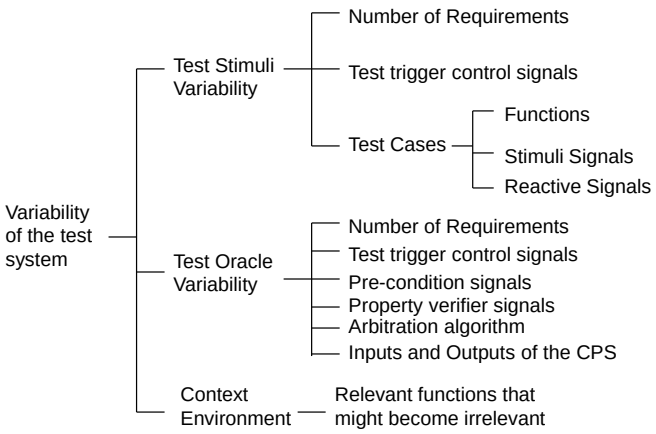

Fig. 12 Taxonomy of the variability for the chosen test system

necessary to test the configuration. Finally, it removes the elements of the test system that are not needed to test the selected system variant.

6.1 Input Files

As shown in Figure 11, a set of input files are needed for the generation of the test system. These inputs include:

- A variability management file which is manually developed by test and system engineers.
- A model of the CPSUT, which is developed by system engineers.
- A model related to the context in which the CPS operates, which is usually built by test engineers.
- A configuration file, which is related to a specific system variant that must be tested. This can be either automatically generated by combinatorial algorithms or manually, for testing specific user needs.
- A set of test cases, which can be either automatically generated from a behavioural SUT model or manually specified by test engineers.
- A set of requirement monitors, that are employed by the test oracle to verify that each requirement meets the specifications provided by the test engineers.

6.1.1 Variability management

For generating the test system instance, one of the most important files is the variability management file, which is in charge of specifying the variability of both the CPSUT, as well as the test system. The tool could clearly be used in a wide range of industrial applications. According to the survey conducted by Berger *et al.* (2013), feature modeling is the most used notation in industry to manage variability, where 74.3% of respondents reported using feature models for this task. Moreover, industrial practitioners are not usually familiar with modeling notations (Wang *et al.*, 2016). However, researchers in the field of testing have demonstrated that feature

models can be appropriate for industrial test engineers (e.g., test engineers from Cisco (Wang *et al.*, 2016)). In addition, we use Simulink models of the system, and thus the variability we consider is limited to the system model level. For the cases we have considered, feature models have supported our needs. Further, an industrial feature modeling tool named pure::variants (Pure-Systems, 2014) has demonstrated that feature models can deal with variability of Simulink models. The Feature Modelling tool chosen was FeatureIDE (Thuem *et al.*, 2014). The reason for choosing FeatureIDE was that it is a robust tool, which employs an intuitive user interface. Moreover, this tool is open source, which allows for the adaptation of the tool to our needs. It also supports both the automatic selection of system variants employing search algorithms as well as manual selection of products employing a user interface. Other variability notations can also be appropriate for the variability management of configurable CPSs (e.g., SimPL (Berger *et al.*, 2013), Clafer (Bak *et al.*, 2015) or Zen-CC (Lu *et al.*, 2016b)). We believe that ASTERYSCO can be adapted to support these tools.

On the one hand, system engineers develop the feature model related to the configurable CPS, taking into account the variability of both the Cyber and the Physical layers. On the other hand, test engineers build the feature model related to the test system, taking into account the variability of the test system. For the chosen test system, the variability is summarized in Figure 12. Note that when the test system for a specific configuration has to be generated, the tool must take into account several test related issues. The requirements of the selected configuration as well as the associated test trigger control signals have to be taken into account (for both the test stimuli and test oracle). The test cases associated to the configuration must also be allocated inside the test stimuli source, where the test stimuli signals (i.e., the ones that are employed to stimulate the CPSUT) corresponding to the configuration must be instantiated. This has an impact in the inputs and outputs that are evaluated by the test oracle, where the ones related to the specific configuration must be selected, and the not selected ones must be removed from the simulation model. The precondition and property verifier signals must also be allocated for the selected configuration (these ones in our test system are related to specific requirements). In addition, the arbitration algorithm used for the automatic evaluation must be specified (in our case this is specified beforehand). Finally, the different functions related to the context environment have to be taken into account, as not all the functions have an influence in all the configurations (e.g., if in a configuration of the UAV example the collision avoidance feature is not chosen, the function related to the obstacle is removed from the context environment).

When both feature models are developed (i.e., the one related to the system and the one related to the test system), test engineers integrate them, and trace the elements related to the CPSUT with the elements related to the test system. This traceability is done with the constraints of the feature modeling technique, i.e., with "requires" and "excludes." For this integration it is important to correctly interpret system requirements, and how each system requirement is associated with the features related to the system. It is important to highlight that in this stage there must be an efficient communication between the system engineer and the test engineer as some misunderstandings might appear. For instance, the name given to a specific system feature can be inappropriate so that the test engineer can interpret its association with a specific system requirement. The traceability is important when configuring a specific system variant, in this way the elements of the test system are automatically selected when a system variant is chosen.

Figure 13 depicts the test feature model used to manage the variability of the test system for the proposed case study.[4] The test system is divided into three main parts: Requirements, Stimuli Signals and Context Environment. In each part, child features are allocated in terms of optional (e.g., r_{20}, yaw_ref or Obstacle) or mandatory (e.g., r_1, context_signals or Wind). After

[4] For presentation reasons of the article, we could not allocate every single feature of the test system.

developing the test feature model for managing the variability of the test system, the following step would be to integrate it with the CPSUT's feature model.

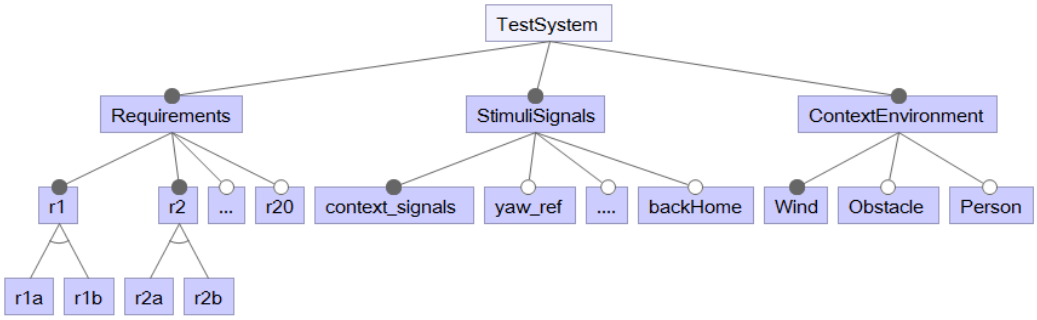

Fig. 13 Test system feature model without being integrated with the CPSUT

The integration with the CPSUT is mainly important for the automatic configuration of the test system. When the feature model that manages the variability of the CPSUT and the test feature model are integrated, the features of both need to be traced with cross-tree constraints. Thus, when a specific system variant of the CPSUT is chosen, the elements related to the test feature model are automatically selected. As an example, the requirement for the flying LED is r_{20}: "The drone shall turn on the flight LED while it is in the air." Therefore, to automatically select r_{20} when the flying LED is part of the system variant, the cross-tree constraint between the CPSUT feature model and the test system is the following: $Fly\_Led \Rightarrow r_{20}$. In addition, when the flying LED is not part of the system variant, r_{20} must be automatically unselected. For this case, the cross-tree constraints would be the following: $\neg Fly\_Led \Rightarrow \neg r_{20}$.

6.1.2 CPSUT

Another input when generating the test system is the CPSUT model, which is a 150% model of the CPSUT. 150% models integrate all the variability, i.e., the variability related to the whole product line; when a specific system variant needs to be selected, the variability of the 150% model is bound. The CPSUT is part of the input, as the test system needs to be integrated with it. Figure 14 shows the input model of the CPSUT related to the case study provided in Section 3. Note that the output "States_Real" is a bus with information related to the UAV.

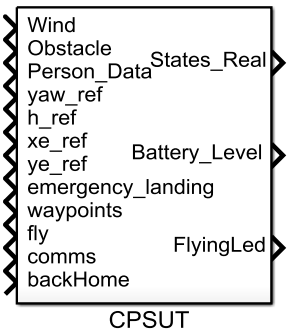

Fig. 14 CPSUT input model of the AR.Drone Simulink model

6.1.3 Context Environment

In Section 5.1.3 we described the context environment. The context environment model is taken as an input. This model simulates the behavior of the environment that directly affects the CPSUT. For the example provided in Section 3, the wind, obstacles and information related to the person that the UAV must follow (if the functionality is chosen) is modeled. Figure 15 depicts the context environment model for our case study.

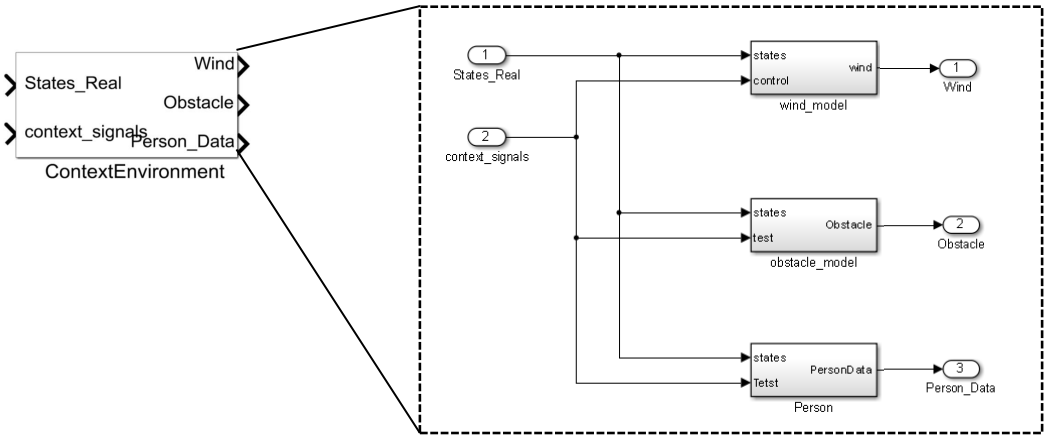

Fig. 15 Context environment model for the AR.Drone Simulink model

6.1.4 Configuration

The configuration file expresses the elements of both the CPSUT and the test system, for the selected system variant. System variants can be generated either automatically or manually. When generating system variants automatically, a very typical technique used in configurable systems testing is Combinatorial Interaction Testing (CIT). For example, the pairwise testing strategy generates a set of configurations so that two components interact between themselves at least once. The tool FeatureIDE allows the automatic generation of system variants using t-

wise techniques, specifically, three algorithms can be used: CASA (Garvin *et al.*, 2011), Chvatal (Chvatal, 1979) or ICPL (Johansen *et al.*, 2012).

The configurations can also be specified manually. FeatureIDE provides a user interface to perform this task. This can be useful when a customer wants a specific configuration, and before the configuration is built, it must be tested and validated by a simulation tool. The Test Engineer chooses a specific CPSUT configuration, and the elements of the Test System are automatically selected; this is possible as we have previously traced the components of the CCPS with the elements belonging to the test system. When all the features from the feature model are selected, the configuration file (which has a *.config extension) related to the chosen configuration is automatically generated by FeatureIDE.

As an example, we manually customized the C.AR.Drone. For this case, we chose a simple configuration, c_1, with the following features: $F_{c_1} = \{TrackSystem, Point\_To\_Point, PI, ConcreteCoordinates, BatteriesManagement, BatteryLanding, Short\_Duration, GyrA, GPS\_A, BS\_A, LowSpeed\}$. The elements of the test system needed to test c_1 are automatically selected:

- Requirements: $R_{c_1} = \{r_1, r_{1a}, r_2, r_{2a}, r_3, r_{3a}, r_4, r_{4a}, r_{10}, r_{11}, r_{12}, r_{13}, r_{16}, r_{17}, r_{18}, r_{19}\}$
- Stimuli Signals, $SS_{c_1} = \{context\_signals, yaw\_ref, h\_ref, xe\_ref, ye\_ref, waypoints, fly, comms\}$
- Context Environment, $CE = \{Wind\}$

6.1.5 Test Cases

Other inputs to ASTERYSCO are the test cases needed to test the CPSUT. Although this is not part of the contribution of this article, the test cases can be generated automatically using Model-Based Testing (MBT) techniques. There are several ways to generate reactive test cases. For instance, the tool Time Partition Testing (TPT), from piketec, is a MBT tool for testing embedded control systems (Piketec, 2015). This methodology allows, among others, graphical test modelling, automatic test execution, etc (Bringmann & Kramer, 2008). This tool also permits testing MATLAB/Simulink models, ASCET models, AUTOSAR software, C-code, etc. There are other approaches for the automatic generation of reactive test cases (e.g., (Zander-Nowicka, 2007), (Mjeda, 2013)).

6.1.6 Requirement monitors

To obtain the results of the executed test cases, the test oracle uses a set of requirement monitors. These requirement monitors observe several properties of the test system and generate a set of test verdicts. The requirement monitors are taken as an input to ASTERYSCO. These requirement monitors also encapsulate variability, which is bound by the test system configurator for the selected system variant.

6.2 Test System Generation

The first step consists of generating a generic test system, which is later used by the test system configurator to obtain the final test system. Figure 16 shows the tasks performed to generate the generic test system. The first task corresponds to the parser, which reads the variability management file. During the second task, the test system elements are automatically generated. To generate these elements, the data provided by the parser is processed and the required elements are taken from the outside libraries and allocated in a model. The CPSUT model is also allocated

in the same model. Finally, this model is sent to the test system integrator, which automatically integrates the input and output ports of the different elements allocated in the model.

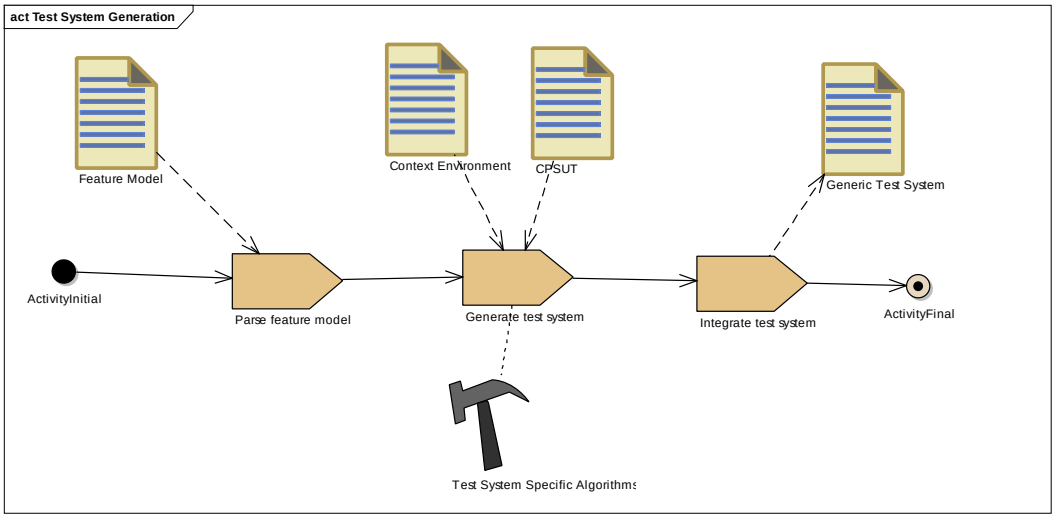

Fig. 16 Overview of the tasks of the test system generator for the generation of the generic test system model. The arrows with the solid line indicate the control flow, whereas the arrows with the dashed lines indicate the object flow. The first task parses the feature model, the second task generates the test systems' elements and the last task integrates these elements with the CPSUT. The input of the first model is a feature model that captures the variability. The second task takes as inputs the context environment and the CPSUT's model. The output of the third task is the generic test system.

ASTERYSCO was developed in MATLAB. The variability management tool is FeatureIDE using feature models. The models in FeatureIDE are saved in *.xml. For the chosen test system, when generating the Test Stimuli, the test system generator needs to process the name and number of the requirements and stimuli signals. When generating the test oracle, the name and number of each requirement is needed. Thus, the parser extracts all this data to send it to the test system elements generator. Once this data is extracted, the test system elements generator uses MATLAB scripts to generate the test system elements in accordance with the architecture explained in the Section 5.1. Finally, the test system integrator reads the test system elements model and joins all its ports.

Integrating the test stimuli, the test control and the test oracle is systematic, as the ports that have to be integrated do not vary. Nevertheless, the output signals of the test stimuli, the signals of the context environment and the signals of the test CPSUT vary from system to system. The automatic integration of these sources is possible when the names of the input and output ports to be linked are the same. Otherwise, a manual refinement process is needed for the generation of the whole test system. This integration is automatically performed with Algorithm 1. Figure 17 depicts the automatically generated generic test system model for the proposed case study by our test system generator.

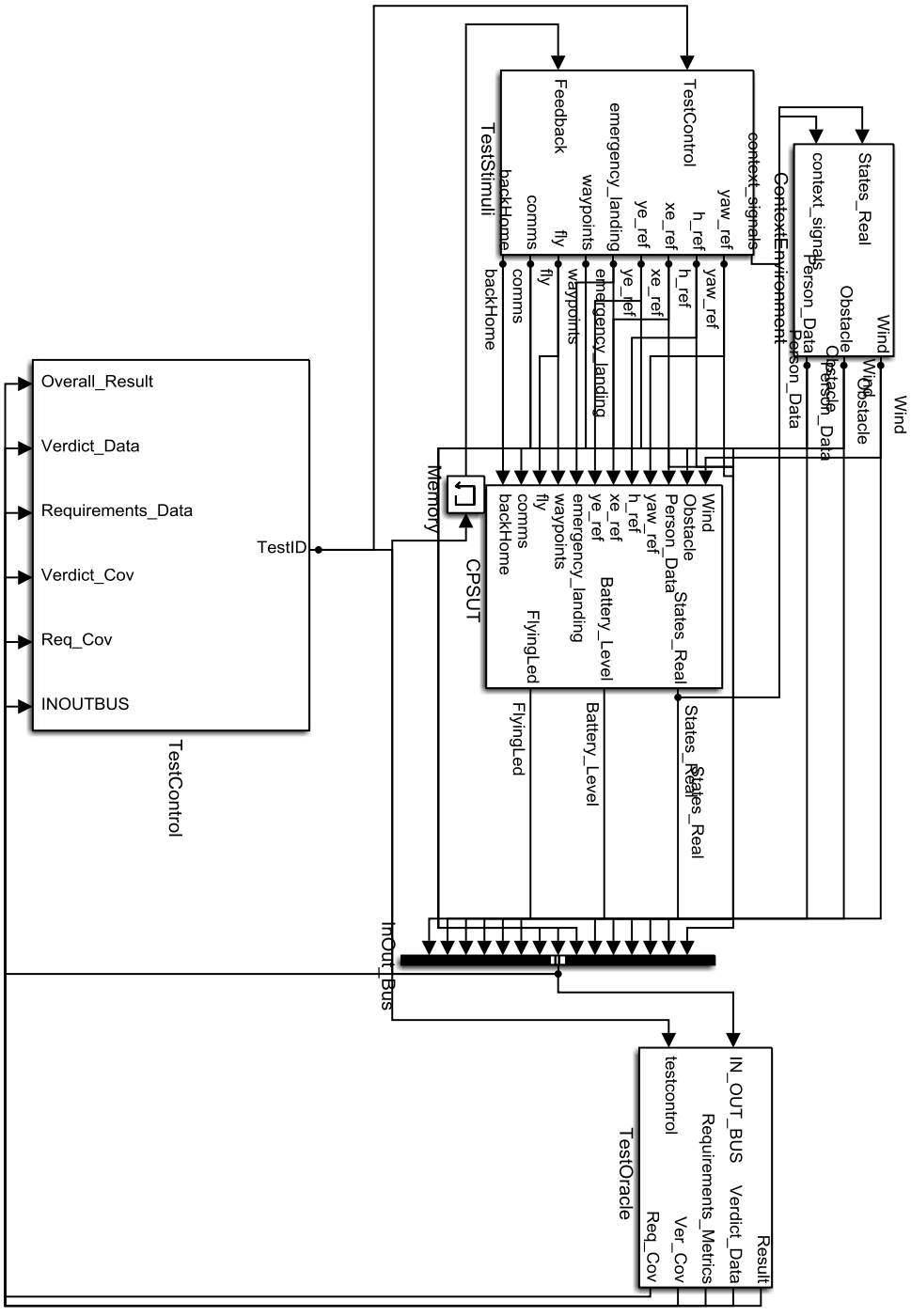

Fig. 17 The automatically generated generic test system model

```
Data: Test System Elements model
Result: Generic Test System
TestSystem = Read(Test System Elements model);
for i=1 to size(TestStimuli.Outputs) do
    for j=1 to size(ContextEnvironment.Inputs) do
        if TestStimuli.Outputs(i).Name == ContextEnvironment.Inputs(j).Name then
            addLine(TestStimuli.Outputs(i),ContextEnvironment.Inputs(j);
        end
    end
    for j=1 to size(CPSUT.Inputs) do
        if TestStimuli.Outputs(i).Name == CPSUT.Inputs(j).Name then
            addLine(TestStimuli.Outputs(i),CPSUT.Inputs(j);
            addLine(TestStimuli.Outputs(i),InOutBus);
        end
    end
end
for i=1 to size(ContextEnvironment.Outputs) do
    for j=1 to size(CPSUT.Inputs) do
        if CPSUT.Outputs(i).Name == ContextEnvironment.Inputs(j).Name then
            addLine(ContextEnvironment.Outputs(i),CPSUT.Inputs(j);
            addLine(ContextEnvironment.Outputs(i),InOutBus);
        end
    end
end
for i=1 to size(CPSUT.Outputs) do
    for j=1 to size(ContextEnvironment.Inputs) do
        if CPSUT.Outputs(i).Name == ContextEnvironment.Inputs(j).Name then
            addLine(CPSUT.Outputs(i),ContextEnvironment.Inputs(j);
        end
    end
end
```

Algorithm 1: Algorithm for the automatic integration of the Test Stimuli, CPSUT and Context Environment

6.3 Test System Configuration

Once the generic test system is generated, the test system configurator needs to rebuild it to test a specific system variant by generating the test system instance. The test system configurator works as follows: first, it parses the needed information from the variability management file as well as from the configuration file. The test system configurator relates the selected features in the configuration file to the features of the variability modelling file (which are generic files). After this relationship has been established, the elements related to the configuration file are replaced by the elements corresponding to the variability modelling file. Finally, the non-selected elements are deleted from the model.

FeatureIDE saves the models in *.xml format and the features of a specific system variant in *.config format. For the selected test system, we focused on requirements and stimuli signals. The configurator replaces the test cases related to the generic test system (which are empty subsystems), with the test cases for the specific system variant. The variability of the test cases is

also bound (e.g., a specific signal that is not needed for the system variant might be deleted, etc.). The same happens with the requirement monitors, where the generated requirement monitors are replaced by configuration-specific requirement monitors. Lastly, the generated empty test cases and empty requirement monitors that do not represent the chosen configuration are removed. Algorithm 2 shows the strategy followed for the configuration of the requirements.

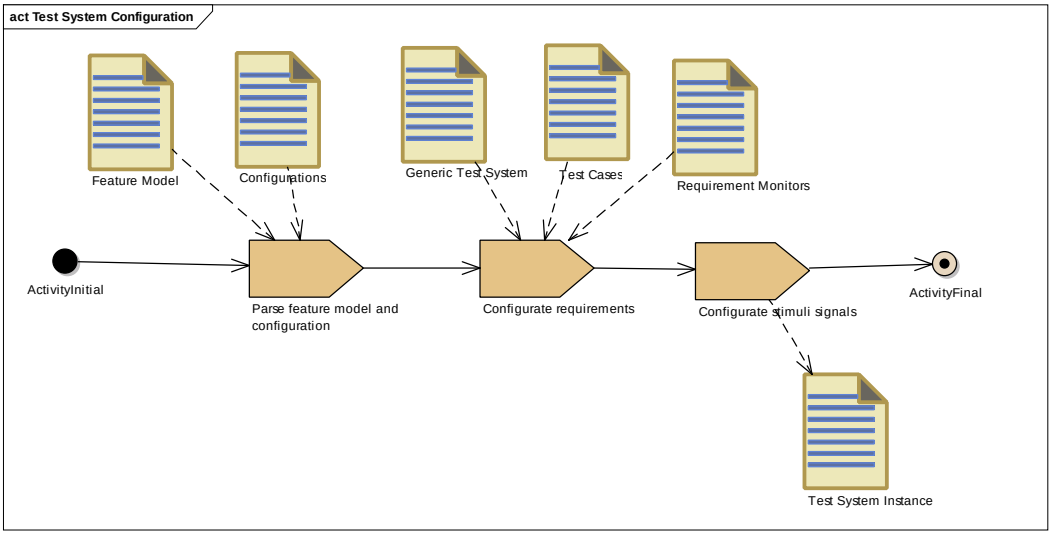

Fig. 18 Overview of the processes corresponding to the test system configuration for a specific system instance. The arrows with the solid line indicate the control flow, whereas the arrows with the dashed lines indicate the inputs and outputs of each steps. The first task parses information related to the feature model and the configuration. The second process configures the necessary elements for the selected requirements. Finally, the last process configures the test stimuli signals. The inputs to the first task are the feature model that captures the variability and the configuration file for a specific system instance. The second process obtains as inputs the generic test system, the test cases and the requirement monitors. The output of the last process is the test system instance, which is ready to test the selected configuration.

In the case of the stimuli signals, the process is easier. The stimuli signals that are not needed to test a specific system variant are automatically removed. Algorithm 3 shows the strategy for configuring the stimuli signals.

```
Data: GenericTestSystem, FeatureModel, ConfigurationFile
Result: TestSystemInstance
FM = parse(FeatureModel);
CONFIG = parse(ConfigurationFile);
for i=1 to size(CONFIG.Req) do
    Req = Config.Req(i);
    for j=1 to size(FM.Req) do
        if FM.Req(j).Children == 0 then
            //This means that it does not have children requirements
            if FM.Req(j) == CONFIG.Req(i) then
                req_id(j) = 1;
                ReplaceTStimuliReq(FM.Req(j).Name,CONFIG.Req(i).Name);
                ReplaceTOracleReq(FM.Req(j).Name,CONFIG.Req(i).Name);
            end
        else
            //Has children requirement;
            for k=1:size(FM.Req(j).Children) do
                if FM.Req(j).Children(k) == CONFIG.Req(i) then
                    req_id(j) = 1;
                    ReplaceTStimuliReq(FM.Req(j).Children(k).Name,CONFIG.Req(i).Name);

                    ReplaceTOracleReq(FM.Req(j).Children(k)Name,CONFIG.Req(i).Name);

                end
            end
        end
    end
end
for for i=1 to size(FM.Req) do
    if requirements_id(i) == 0 then
        RemoveTStimuliReq(FM.Req(j).Name);
        RemoveTStimuliReq(FM.Req(j).Name);
    end
end
```

Algorithm 2: Algorithm for the automatic configuration of the test system for the selected configuration requirements

Figure 18 shows the processes that ASTERYSCO performs to configure the generic test system for a specific test system instance. Note that the requirements configurations as well as the stimuli signals configurations are based on the chosen test system variability. In the case that another test system is chosen, these blocks should be replaced based on the variability points of the selected test system. In ASTERYSCO, after removing the requirement monitors of the test oracle, the verdict and coverage bus creator loses the signals of the removed requirement monitors. This is a problem for the arbitration algorithm when the different metrics have to be calculated. Thus, these two bus signals must also be adapted. Algorithm 4 configures these bus creators.

```
Data: GenericTestSystem, FeatureModel, ConfigurationFile
Result: TestSystemInstance
FM = parse(FeatureModel.xml);
CONFIG = parse(c_1.config);
for i=1 to size(FM.StimuliSignals) do
    RemoveStimuliSignal = true;
    for j = 1 to size(CONFIG.StimuliSignals) do
        if FM.StimuliSignals(i).Name == CONFIG.StimuliSignals(j).Name then
            //The stimuli signal is in the selected config;
            RemoveStimuliSignal = false;
        end
    end
    if RemoveStimuliSignal == true then
        RemoveStimuliSignal(FM.StimuliSignals(i).Name;
    end
end
```

Algorithm 3: Algorithm for the automatic configuration of the test system for the selected stimuli signals

```
Data: GenericTestSystem, FeatureModel, ConfigurationFile
Result: TestSystemInstance
FM = parse(FeatureModel);
CONFIG = parse(ConfigurationFile);
//Deletes buses for new number of inputs;
deleteBusCreator(VerdictBus);
deleteBusCreator(CoverageBus);
deleteUnconnectedLines();
//Adds bus creators with new number of inputs;
addBusCreator(VerdictBus, size(CONFIG.Requirements));
addBusCreator(CoverageBus, size(CONFIG.Requirements));
//Integration of bus with arbiter;
addLine(VerdictBus,Arbiter);
addLine(CoverageBus, Arbiter);
//Integrate Requirement monitors with buses;
for i=1 to size(CONFIG.Requirements) do
    addLine(CONFIG.Requirements(i).Name,VerdictBus);
    addLine(CONFIG.Requirements(i).Name,CoverageBus);
end
```

Algorithm 4: Algorithm for the automatic configuration of the test verdict and coverage bus from the test oracle for a configuration

6.4 Output

Figure 19 shows the final results of the test system to test the chosen configuration. The non-selected components and signals have been removed from the model. For instance, the ports corresponding to “emergency_landing” or “back_home” from the test stimuli have been removed. The same happens with the “Obstacle” and “Person_Data” from the context environment and the “FlyingLed” from the CPSUT.

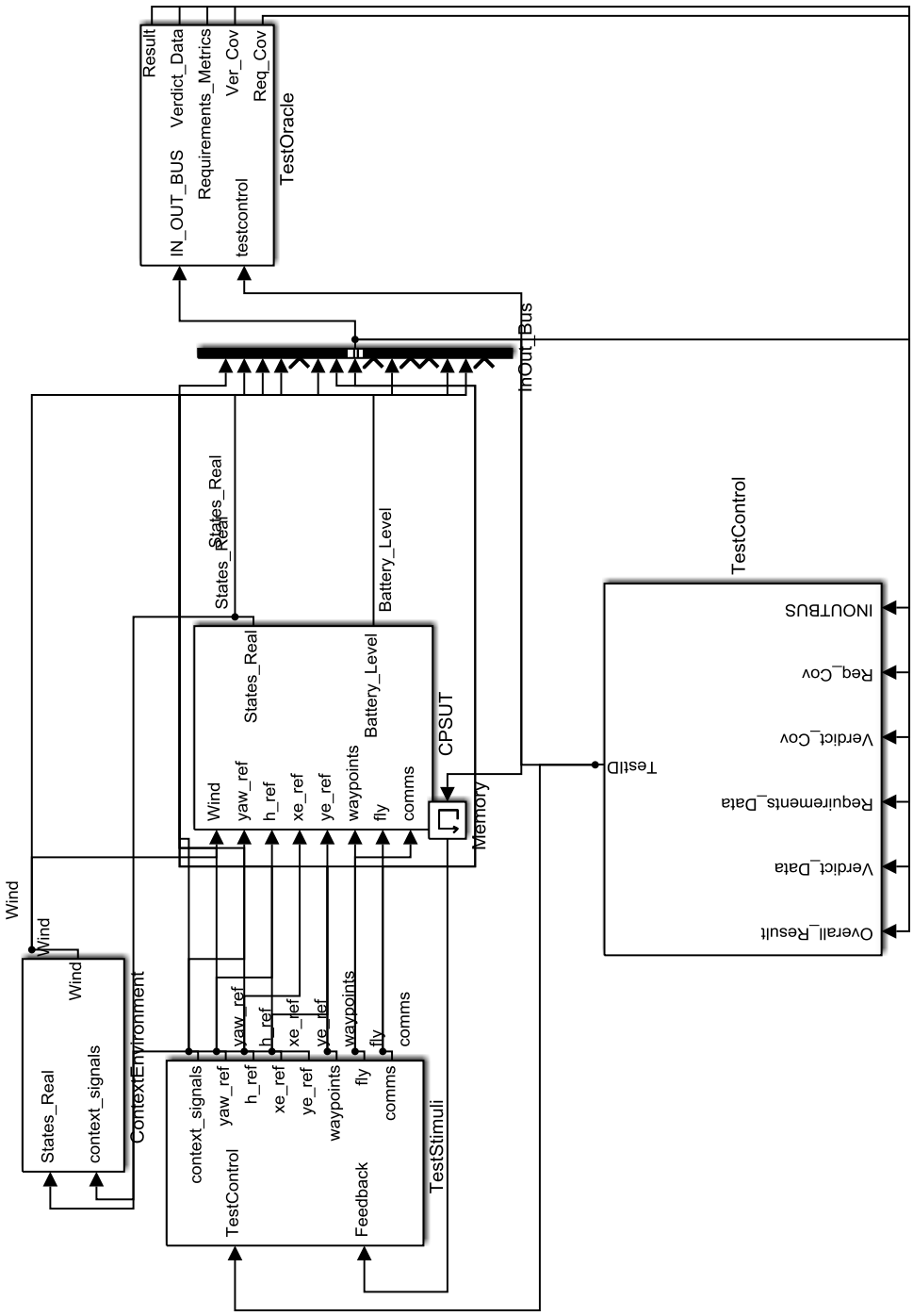

Fig. 19 Configured Simulink Model for the configuration c_i of the proposed illustrative example

6.5 Limitations

We have identified some limitations related to the current version of ASTERYSCO:

- Test System: We have used an extension of a test system that was designed for the validation of embedded systems from the automotive domain at system level. The developed tool is specific for the test system explained in Section 4.
- Simulation tool: Our prototype uses MATLAB/Simulink as the simulation tool. Other tools can be used for simulation, however, these tools must support scripts to generate the test system model. In addition, CPSs involve different technologies, and often, co-simulation is needed. We have not performed experiments using two independent simulation tools.
- Variability management tool: Our prototype used feature models employing the tool FeatureIDE (Thuem *et al.*, 2014). Feature model and the tool FeatureIDE may have some limitations (e.g., FeatureIDE does not support cardinality type variability) that we have not considered. Other tools, such as SimPL (Behjati *et al.*, 2013), Clafer (Bak *et al.*, 2015) or ZenCC (Lu *et al.*, 2016b) might be more adequate for variability modeling of CPSs. Although we did not performed experiments with other tools, we believe that the only changes to be performed would be the ones related to the parser.

7 Evaluation

This section evaluates the proposed approach for generating test system instances for configurable CPSs. We provide a case study and measure the time needed by ASTERYSCO to generate the test system instances for ten configurations. We also measure the time required by Simulink to simulate the executed tests. Later, we compare the difference of generating test systems employing our tool against to a manual process. Finally, the obtained results are discussed and some identified threats to validity of the performed evaluation are highlighted.

7.1 Case Study

The case study we employed for the evaluation of our approach is the one presented in Section 3. Before executing the case study, the test cases as well as the different requirement monitors for testing each functional requirement were developed. We selected a time-triggered execution strategy, i.e., a time slot was defined for each test case, and the test control measured the test execution time of each test case before executing the following one.

Regarding the test execution (i.e., the test stimuli), each requirement was tested by one or more reactive test cases. As for the test evaluation (i.e., the test oracle), each requirement was monitored by three requirement monitors. The first requirement monitor observed the tasks related to the high level control (i.e., if the position of the UAV was the correct one for the specified requirement). The second requirement monitor observed the tasks related to the low level control (i.e., if the stabilization of the UAV was correct). The third requirement monitor observed the specified property for the requirement. Figure 20 depicts the three requirement monitors for r_1. The test cases as well as the requirements monitor were stored in the Simulink library.

Once the test cases and the requirement monitors were developed the generic test system was generated. In a second step, we selected the configurations to test. These configurations were selected manually and automatically. We manually selected 10 different configurations ranging from c1 to c10. The simplest system variant is c1, which has minimum functionalities, whereas

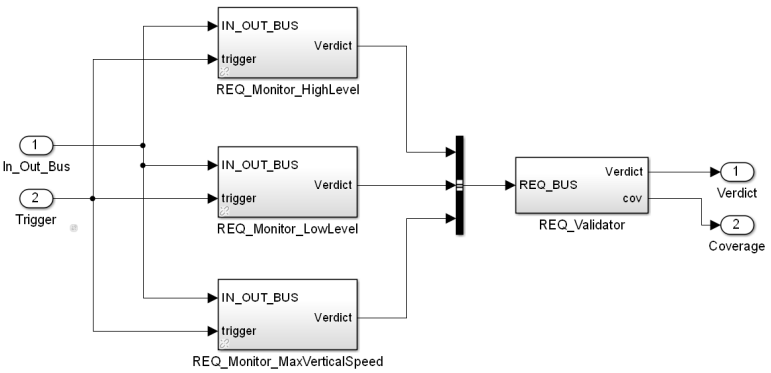

Fig. 20 Detail of the three requirement monitors for the validation of requirement r_1 and its requirement validator

c10 is the most complex configuration with all the functionalities. We selected configurations of different complexities for two main reasons. The first reason is that users usually select products based on their price and based on their features. Products with more features tend to be more expensive, and as a result, many users cannot afford them. On the contrary, cheap products tend to have few features, and while some users might consider the features of a cheap configuration to be enough, others could need a more sophisticated system. The second reason is related to the impact of the time required by ASTERYSCO to generate test systems based on the complexity of the configurations. We wanted to measure if our tool requires more time for configuring test systems with more requirements. We also selected configurations to test automatically. We employed FeatureIDE, which integrates different search algorithms for t-wise configuration selection. We employed the algorithm ICPL (Johansen *et al.*, 2012) as it is the fastest configuration selection algorithm. We configured the algorithm to select the system variants to test following a pairwise criteria. The reason for selecting this criteria is that at least the interaction of two features is ensured, and thus, it can be ensured that the system does not fail due to the interaction of two features. In total, the ICPL algorithm selects 28 system variants. The name of the configurations that were automatically selected start with "pw."

Table 3 shows the number of features, requirements and stimuli signals of each selected configuration. Later, we generated the test system for each selected configuration with our approach and measured the time. We repeated this process 100 times to account for random variations when generating the test system. It can be appreciated that as the number of requirements that a specific configuration has increases, the time required to generate the test system tends to increase. In addition, we measured the time required by Simulink to test each configuration.

7.2 Results

Results of the case study are reported in this Section. Table 3 shows the time required to generate the test system instance for each configuration. Each configuration has a different number of requirements. The generation of each test system was repeated 100 times to account for random variations, as recommended by Arcuri & Briand (2011). The test systems were generated with a computer running a Windows 7 operating system in an Intel Core i5, 2.5 GHz processor with 8 Gb of RAM memory. The column showing the generation time refers to the mean time required by our tool for generating each test system for the 100 repetitions. The standard deviation of the 100 repetitions is shown in the sixth column. The last column corresponds to the time

required by Simulink to execute the test cases. The mean time for generating the generic test system was of 3.85 seconds, with a standard deviation of 0.11 seconds. The fastest test system configuration was for the pw15, which had 11 requirements and seven stimuli signals. The mean time for configuring this test system was of 2.84 seconds. On the contrary, c10, which had 20 requirements, was the configuration for which ASTERYSCO employed more time for the test system configuration, specifically, 3.53 seconds. Once the test systems were generated, the tests were executed. In total, about five hours were needed to test the 38 configurations.

7.3 Comparison with Manual Test System Generation

The steps that have to be taken to generate a test system as well as the differences between employing a manual test system generation or employing ASTERYSCO are shown in Table 4. First, a feature model as well as the test infrastructure must be elaborated. Second, configurations must be selected. Third, the generic test system is generated.[5] Lastly, the test system for each configuration is generated.

7.3.1 Step 1: Elaboration of Feature Model and test infrastructure

The first step corresponds to the elaboration of the feature model as well as the test infrastructure. Regarding the feature model, in the case of manual configuration, just the feature model of the system would be needed to be developed for managing variability of the system. The elaboration of the feature model also allows traceability with the CPS model, which warrants a systematic configuration. In addition, in the manual process, the test feature model would not be needed. Regarding the test infrastructure, just the test assets would not be developed or generated. The main *benefit* of employing a manual approach is that just the feature model related to the system would be developed, not having to develop the test feature model nor the test assets. On the contrary, the *limitation* would be that just any information related to the variability of the system is available, and not information related to the test.

In the case of ASTERYSCO, the feature model of both the system as well as the test system has to be elaborated. In addition, in our proposed methodology, the dependencies between the features of the CPS and the features related to the test system must be traced with cross-tree constraints. As for the test infrastructure, different reusable test assets would be needed to develop. The *benefit* in this case is that information related to the variability of the test system is available in a feature model. However, the *limitations* are that the the feature model related to the test system must be manually elaborated. In addition, the reusable test assets must be generated. These tasks result in an extra investment of time.

T_FM_{sys} is the time needed to develop the feature model of the system, and is the same in both methodologies. Our proposed methodology includes the part related to the elaboration of the test feature model. T_FM_{test} is the time required to elaborate the test feature model and trace the features of the system feature model with the test feature model. T_ASSETS is the time required to the creation of reusable test assets.

7.3.2 Step 2: Selection of Relevant Configurations

The second step corresponds to the selection of relevant configurations. We previously explained that this step can be performed either automatically (i.e., employing CIT algorithms) or manually. Configurations are usually generated automatically when feature pairwise coverage needs

[5] This step is not mandatory for the manual configuration.

Table 3 Time required by ASTERYSCO to generate the generic test system as well as different configurations

	Features	**Reqs**	**Stimuli Signals**	**Generation Time (sec)**	**Standard Deviation**
GenericTS				3.85	0.11
c1	17	10	7	2.87	0.1
c2	20	11	8	2.93	0.07
c3	21	12	8	3.04	0.06
c4	22	13	9	3.11	0.06
c5	23	14	10	3.2	0.07
c6	24	15	10	3.26	0.07
c7	25	17	10	3.4	0.07
c8	28	18	10	3.49	0.06
c9	29	19	10	3.5	0.06
c10	31	20	10	3.53	0.14
pw1	16	11	3	3.23	0.26
pw2	26	13	8	3.27	0.32
pw3	26	13	5	3.00	0.06
pw4	26	16	8	3.18	0.05
pw5	22	15	8	3.12	0.05
pw6	27	15	6	3.19	0.22
pw7	30	17	8	3.29	0.07
pw8	27	16	10	3.18	0.06
pw9	24	14	5	3.06	0.05
pw10	28	14	8	3.05	0.05
pw11	26	12	4	2.92	0.00
pw12	28	15	5	3.16	0.00
pw13	23	12	4	2.92	0.00
pw14	26	14	5	3.07	0.00
pw15	22	11	7	2.84	0.00
pw16	26	15	5	3.14	0.00
pw17	25	13	7	2.98	0.00
pw18	23	13	3	3.01	0.00
pw19	22	12	3	2.92	0.00
pw20	21	13	3	3.02	0.00
pw21	19	12	3	2.94	0.00
pw22	24	13	3	3.16	0.00
pw23	24	13	3	3.00	0.00
pw24	24	13	3	3.01	0.00
pw25	24	13	3	3.00	0.00
pw26	24	13	3	3.01	0.00
pw27	24	13	3	3.10	0.00
pw28	24	13	3	3.13	0.00

to be achieved. However, some customers might want specific configurations and thus, the features of the configurable CPS would be manually selected. The tool we employ for elaborating

Table 4 Comparison of ASTERYSCO with manual generation of test systems

Steps	Manual	ASTERYSCO	Difference
Step 1	Elaborate Feature Model	Elaborate Feature Model and Test Feature Model	Employing our method, the feature model related to the test system as well as the reusable test assets must be elaborated
Step 2	Select relevant configurations	Select relevant configurations	The selected configurations with our method include the features related to the test system, whereas when not employing our approach just the features related to the system appear
Step 3		Generate generic test system	Employing the manual approach it is not necessary to generate a generic test system
Step 4	Generate test system for each configuration	Generate test system for each configuration	ASTERYSCO generates automatically test system instances whereas with a manual methodology clone and own strategy would be needed,. This results in a time consuming, error-prone and non-systematic process.

the feature model (i.e., FeatureIDE) supports both options and generates a *.config file for each configuration.

When not using the approach presented in this paper, the features related to a configuration appear in a *.config file. The *benefit* in this case is that only information related to the CPSUT appears, and when configuring the CPSUT model there is no need for dividing CPSUT information and information related to the test system. However, this implies an important *limitation*: system requirements do not appear in the *.config file, and as a result, test engineers have to select them manually. This is a problem in the following steps when generating test systems for specific configurations as test engineers have to constantly check the requirements documents and the selected features. Apart from not being a systematic process, the error proneness when generating the test system is increased (as some requirements might be omitted or incorrectly selected). Moreover, the process of manually selecting the requirements of each configuration when there are many configurations to test (which is the case of configurable CPSs) can be infeasible.

When employing ASTERYSCO, the features related to the test system are automatically selected based on the cross-tree constraints and they appear in the *.config file. One of the *benefits* in this case is that the requirements as well as other features (such as the stimuli signals or the context functions) are specified in a *.config file. This allows for the automatic configuration of the test system by our tool. A possible *limitation* might be that if just information about the CPSUT model is needed, then information related to the test system would need to be removed. The time for selecting relevant configurations (i.e., T_CONF_SEL) is the same for both methods.

7.3.3 Step 3: Generic Test System Generation

The third step corresponds to the generation of the generic test system. In the case of the manual approach for generating the test system, this step can be omitted. The major *benefit* might be

that there is no need to spend time developing a generic test system. Nevertheless, a possible *limitation* for the manual approach is that when configuring a test system for a specific system variant, a generic test system could facilitate this process.

As explained in Section 6.2, in the case of ASTERYSCO, a generic test system is mandatory so that in the configuration step the test system can be configured efficiently. This generic test system is automatically generated. The *benefit* of using our tool for generating the generic test system is that it is fully automated and it can be generated in a few seconds (as shown in Table 4). One possible *limitation* when using of the method we propose is that an error free test feature model has to be developed. If the the feature model contains an error (e.g., the name of a stimuli signal is incorrect) a manual refinement process might be needed. In addition, there might be other major errors; for example, a requirement might not be well traced with a specific feature. This might imply that when a configuration holds this specific feature the requirement is not tested. $T_{genericTS}$ is the time required by ASTERYSCO to generate a generic test system.

7.3.4 Step 4: Configuration of the Test System

In the last step, the test system for each configuration is generated. This process is named configuration of the test system. In the case of manual elaboration of the test systems for each configuration, different strategies can be employed. The most typical would the one related to "clone and own." In this case, a test system as well as the test infrastructure (i.e., test cases and requirement monitors) would be generated for a specific configuration and later reused to generate other test systems for other system variants. The different elements of the test system would be needed to be manually generated (i.e., the stimuli signals, the test oracle, the context environment and the test control); this task can be time consuming as the selected test system can be complex for the test engineer. Moreover, all elements of the test system need to be manually integrated. In addition, there might be many configurations to test (i.e., a manual generation of the test system has to be performed for each system variant). A possible *benefit* of using a manual approach could be that if there are few configurations to test with systems with few requirements, a manual approach might be faster than developing the feature model in Step 1 and the reusable test infrastructure. However, this is not the case of configurable CPSs, which can be configured into many configurations. One *benefit* of manually configuring the test system is that it is not mandatory to have a *.config file with all the requirements. Nevertheless, the practice of manually configuring a test system for each configuration to test has important *limitations*. One of these *limitations* refers to the time required to manually develop a test system. Another *limitations* refers to the time required to generate the test cases and the requirement monitor for each configuration. Another *limitation* is that test engineers are exposed to fatigue when developing many test systems, which could lead to the generation of errors in the test system; this might lead to introduce errors into the test system. Given that there are N relevant configurations to test, $\sum_{i=1}^{N} MAN\_T_{instTS_i}$ is the time required by a manual approach to generate the N test system instances. $\sum_{i=1}^{N} MAN\_Infrastrut_{instTS_i}$ is the time required by a manual approach to generate the test infrastructure for the N configurations.

In our case, ASTERYSCO obtains the generic test system, and the test system for the specific configuration is configured in about 3 seconds (as illustrated in Table 4), reusing the previously generated test assets. This process is fully automated, which warrants the systematic generation of test system instances and reduces the error-proneness as well as the test system generation time. The *benefit* of this approach is related to the configuration time, which is quite low (Table 4). Not only that, employing the tool proposed in this paper ensures a systematic configuration of the test system and reduces the propensity of errors. A possible *limitation* could be that if the test feature model contained an error, this error could be propagated when generating

a test system instance (e.g., a requirement of a specific configuration could be removed from the test system, or a requirement that a specific configuration does not have could be tested). Given that there are N relevant configurations to test, $\sum_{i=1}^{N} AST\_T_{instTS_i}$ is the time required by ASTERYSCO to generate them.

7.3.5 Overall test system generation time

Based on the above-mentioned steps for the generation of test system instances employing ASTERYSCO or a manual approach, we separated the time required by each approach to generate N test system instances. The time required to manually generate N test system instances for testing relevant configurations is given in Equation 1. Equation 2 gives the time required to automatically generate the same configurations employing our method.

$$T\_MAN = T\_FM_{sys} + T\_CONF\_SEL + \sum_{i=1}^{N}(MAN\_T_{instTS_i} + MAN\_Infrastrut_{instTS_i}) \quad (1)$$

$$T\_AST = T\_FM_{sys} + T\_ASSETS + T\_FM_{test} + T\_CONF\_SEL + T_{genericTS} + \sum_{i=1}^{N} AST\_T_{instTS_i} \quad (2)$$

7.4 Discussion

This section discusses the obtained results (Section 7.2) and the comparison between employing ASTERYSCO for generating test system instances with manual generation (Section 7.3).

ASTERYSCO is a tool that automatically generates test systems for specific system variants. To perform this task, the tool first generates a generic test system executing a model to model transformation. The proposed tool transforms a feature model into a generic test system for the tool Simulink. For the case study, the generation time for the generic test system was around 3.85 seconds with a standard deviation of 0.11 seconds. We consider the time required to generate the generic test system good, considering that the chosen case study has 20 requirements and 10 stimuli signals.

The generic test system can be reused across the configurations that the system can be set to. In order to do this, the test system configurator parses the configuration and allocates all the components needed to test the system variant in the test system. The performed experiments show that our tool automates the process of generating the test systems for specific system variants in about 3.5 seconds. On average, 119 seconds were needed to generate the 38 configurations. The test execution time for the 38 configurations took about five hours. Thus, we consider that 119 seconds is a quite small part of the total amount of time required to test the 38 configurations. It is important to highlight that the time required by the method we propose when configuring a test system increases as the number of requirements to be tested increases (Table 3). This might be caused due to the fact that the functions “Replace” (Algorithm 2) take more time to execute than the functions “Remove.” However, the obtained results for configuring a specific test system indicate that our approach considerably reduces the time for generating test system instances. This means that the proposed method can substantially reduce verification and validation stage costs when testing configurable CPSs.

Section 7.3 compares ASTERYSCO with a manual process for the generation of test system instances. Four main steps must be given: (1) feature model elaboration and test infrastructure generation, (2) configuration selection, (3) generation of the generic test system and (4) configuration of the test system. For each step we compare the differences of the two approaches and highlight the main limitations and benefits. In general, the main limitation of of the method we propose as compared to a manual approach is that an amount of time must be invested building the test feature model and generating the reusable test assets. In addition, the test feature model has to be correctly built, otherwise, some errors can be propagated in the test system when generating it using ASTERYSCO. The main limitation of the tool we propose is the main benefit of the manual process for building test system instances. When employing a manual approach for building the test system, it is not mandatory to build a test feature model. However, this can convert the manual generation of the test system into a real challenge, especially when the number of configurations to test is large. The test engineer must check the requirements of each configuration to be tested as well as other features related to the test system before building the test system. This issue involves a human factor, which might result in an increase of the error proneness. Moreover, a manual test system generation process requires a manual test case generation process for each configuration. This could be infeasible because of the time required for this task. The use of the methodology we propose ensures 100% requirements coverage if all the test cases are for each configuration are executed. If the test cases are manually generated, the coverage can achieve 100% requirements coverage. Nevertheless, when the number of configurations to test is too large, the manual test case generation process would be very time consuming and the chances for achieving a high coverage would be reduced.

Concluding Remark: The main limitation of the method we propose is that an investment of time is required when generating the test feature model. Moreover, this test system has to be correctly performed and some testing is required to ensure the correctness of the feature model, otherwise, some errors can be introduced when generating the test system. However, the investment of time when developing the test feature model results in a systematic process when generating the test system. The tool allows full automation of the generation of test system instances for the configurations to test. When N (i.e., the number of configurations to test) is very low, a manual process could be more effective than employing ASTERYSCO. However, this is not usually the case of configurable CPSs. The number of configurations to test is large and thus, a benefit of our approach would be the time to generate test system instances. Equation 3 can be employed to guide when to use the methodology proposed in this paper. Note that we demonstrated that $T_{genericTS}$ and $AST\_T_{instTS_i}$ were in the order of some seconds, and thus might be disregarded. The time required to build the test feature model (i.e., T_FM_{test}) and the reusable test assets (i.e., T_ASSETS) might be some hours, but the time for manually generating each test system instance (i.e., $MAN\_T_{instTS_i}$) and the test infrastructure (i.e., $MAN\_Infrastrut_{instTS_i}$) can also take some hours. Moreover, it is important to note that as the number of configurations to test (i.e., N) increases, considerably more time is required when generating test systems manually. As the number of configurations to test in configurable CPSs is often very large, we recommend using ASTERYSCO for the purpose of generating test system instances.

$$T\_FM_{test}+T\_ASSETS+T_{genericTS}+\sum_{i=1}^{N} AST\_T_{instTS_i} \leq \sum_{i=1}^{N}(MAN\_T_{instTS_i}+MAN\_Infrastrut_{instTS_i}) \tag{3}$$

7.5 Threats to Validity

This section identifies the main threats that can invalidate our evaluation. An *external validity* threat that usually affects most studies is the number of case studies employed. We employed one case study with ten configurations, which might not be enough to generalize our results. However, to reduce this threat, we employed a CPSUT with twenty functional requirements and ten configurations of different complexities. Another *external validity* threat of our study is related to the employed test system. This was an extension for configurable CPSs of that used in (Zander-Nowicka, 2008) for embedded systems. Other kinds of systems might require some adaptation in the test system, and our tool might require more or less time with other test systems. A *conclusion validity* threat involves the randomized time for generating test systems. This means that the time required by ASTERYSCO for generating test systems might suffer some variations. To reduce this threat, the generation of the generic test system as well as the configuration of each test system was repeated 100 times. Later, we calculated the mean time required by ASTERYSCO to generate test system instances as well as the standard deviation. One *internal validity* threat could be the type of computer employed. A non sophisticated computer might require more time for generating test system instances. However, to reduce this threat, we employed a computer with the typical characteristics employed by engineers in industry.

8 Related Work

There are many works in the current literature that involve validation of CPSs. Khakpour & Mousavi (2015) and Mohaqeqi *et al.* (2014) compared different notions of conformance testing for CPSs. Matinnejad *et al.* (2016) proposed a novel search-based algorithm for the automatic generation of test cases for Simulink models of CPSs. However, these works focused on the generation of test cases for CPSs, whereas our work focuses on the generation of test systems for the validation of configurable CPSs. Apart from the generation of test cases, current approaches are also considering testing CPSs under uncertain behaviors to deal with the unpredictability of the physical world (Ali & Yue, 2015)(Zhang *et al.*, 2016). In our case, our test system supports unpredictability of the physical world by (1) simulating the physical world with the context environment and (2) supporting reactive test cases, so that they can monitor the different states of the CPS and react to them.

Configuring configurable CPSs can often be very challenging. Current works have considered different modeling methodologies combined with search engines and constraint solving methods for the efficient configurations of these systems (Nie *et al.*, 2013)(Behjati *et al.*, 2013)(Bak *et al.*, 2015)(Lu *et al.*, 2016b)(Lu *et al.*, 2016a). The objective of our study is not the configuration of CPSs however, but the generation of test systems for configurations of configurable CPSs. As the test system was modeled in Simulink, and the variability of Simulink models can be managed with feature models, we have chosen them for this task. We believe that our approach can be combined with the above-mentioned works that further study the configuration of CPSs by adapting the parsers.

In our previous work-in-progress paper (Arrieta *et al.*, 2014), we analyzed part of the variability of a test system and its elements. We also proposed a traceability strategy among the components of the CPS and the elements of the test system, and we defined a model-based process for the validation of configurable CPSs. However, in (Arrieta *et al.*, 2014), the variability of all the elements of the test system was not analyzed, and although the traceability strategy in the test system management was proposed, the specific elements of the test system management were not specified. This article is the continuation of the research presented in (Arrieta *et al.*,

2014), we have developed a more solid test system, analyzing in detail its architecture as well as its variability. In addition, this article is more focused on the efficient generation of test systems for specific system variants, which is performed by ASTERYSCO. Unlike in (Arrieta *et al.*, 2014), the method presented in this paper allows full automation of the test system generation. In addition, we expand on the evaluation for the generation of test system instances.

Model-in-the-Loop for Embedded System Test (MiLEST) is a toolbox for MATLAB/Simulink developed by Zander-Nowicka (2008). Some elements of our test system architecture, including part of the test stimuli and part of the test oracle have been extended from MiLEST. However, MiLEST is oriented towards the validation of individual embedded systems, and variability was not the objective of the tool. Our test system is designed for configurable CPSs and it is automatically generated by ASTERYSCO to test sytem variants of a configurable CPS. To achieve this goal, the test system supports variability in several points, as explained in Section 6. In addition, the elements of our test system are automatically generated taking into account the test feature model developed in the management stage using the tool FeatureIDE.

Other test systems are also designed for CPSs. For instance, (Kane *et al.*, 2014) focus on requirement monitors that act as a test oracle. We also employ requirement monitors for deciding whether the results of the tests are valid or not. However, their approach is focused on automotive CPSs employing HiL simulations, whereas our approach is designed for MiL and SiL simulations. In addition, Kane *et al.* (2014) just employ test oracles, whereas our test system also focuses on other sources (i.e., test stimuli, test oracle, context environment and test control).

Currently, many industrial projects are moving towards continuous integration and deployment and consequently require an automated solution to test all relevant variants. There are various approaches that use variability in the test system to test different system variants, especially in the Software Product Line domain. Pérez *et al.* (2009b,a) used UML and UTP modeling languages and the variability of the test system was modeled using a UTP extension. Lity *et al.* (2012) and Dukaczewski *et al.* (2013) used state machines as modeling languages, and for variability modeling, a delta oriented modeling strategy was chosen. Pérez *et al.* (2009b,a) did not mention the test strategy followed. Lity *et al.* (2012) used regression testing and Dukaczewski *et al.* (2013) incremental testing . There are several differences between these works and ours. Their SUT target are SPLs, whereas our test system is oriented towards configurable CPSs or CPSs product lines. Their modeling language is UML and UTP or state machines. Although we use state flow (similar to state machines) for some requirement monitors, the modeling language we use is Simulink. With regard to the test strategy, we employed requirements based testing, as the validation is done at system level.

9 Conclusion and Future Work

The number of system variants that have to be tested to ensure that a configurable CPSs will work properly across most of its configurations is large. Manually building a test system for each configuration to be tested is a non systematic and time consuming task. ASTERYSCO is a tool that automates this process. For our prototypical version, we have extended MiLEST (Zander-Nowicka, 2008). In addition, a variability management tool is needed; for this task, we have used FeatureIDE (Thuem *et al.*, 2014). The methodology we propose allows: (1) a systematic generation of test system instances for configurations of configurable CPSs and (2) a reduction in the time for generating these test system instances. An experiment with a case study has shown that employing the tool we proposed, a test system instance can be generated within three seconds. In addition, thanks to the developed tool, we were able to test all the selected configurations in about five hours without any need of human interaction.

As the variability in CPSs is increasing, we believe that this tool may help reduce the validation costs in several domains. For instance, demanding new user requirements or legislation changes lead to multiple development paths in CPSs from some domains (Manz *et al.*, 2014). The variability can appear in the form of configurability or modifiability (Thiel & Hein, 2002). Configurability refers to the variability in product space, whereas modifiability to the variability in time space. Our approach is more focused on configurable CPSs. Nevertheless, evolution is also supported, although the process would need to begin from the beginning, i.e., modifications would be needed in the variability management tool and the generic test system would be generated again.

With regard to the extension of the test system, we believe that although it is designed for configurable CPSs, it can also be applied for other systems such as non-configurable CPSs or embedded systems. Apart from the test system, when testing configurable CPSs it is also important to choose (1) the most relevant configurations, (2) the test cases to execute in the chosen configurations and (3) the order in which the test cases are executed. As this issue is important, ASTERYSCO has been integrated with the test control architecture we proposed in (Arrieta *et al.*, 2015).

In the future, we plan to extend ASTERYSCO to other test systems, as well as other simulation tools. For this task, it will be necessary to properly analyse its variability to adapt the two main elements of our tool, i.e., test system generator, and test system configurator. Regarding the simulation tool, we have used MATLAB/Simulink, due to its easiness to integrate mathematical models that represent the physical layer with software and control algorithms. Although MATLAB/Simulink is a powerful tool, it can have some restrictions (e.g., it is a proprietary software). In the future, we plan to use other simulation tools such as Modelica, LabView, etc. We also want to focus on a plan to transfer the tool to our industrial partners, adapting it to their needs.

Other future plans include the distributed simulation of various test systems. We believe that simulating several system variants in parallel, using different computers can reduce the overall validation time. Currently we are developing a higher level test system that executes in parallel the tests of different system variant instances. Each system variant uses the test system generated by ASTERYSCO, (i.e., each of them has its own test control, test oracle, test stimuli and context environment). We have performed some experiments using the tool "Building Controls Virtual Test Bed" (Wetter & Haves, 2008).

Acknowledgement

This work was developed by the embedded systems group of Mondragon Goi Eskola Politektnikoa, supported by the Department of Education, Universities and Research of the Basque Government.

References

Ali, Shaukat, & Yue, Tao. 2015. U-test: Evolving, modelling and testing realistic uncertain behaviours of cyber-physical systems. *Pages 1–2 of: 8th IEEE international conference on software testing, verification and validation, ICST 2015, graz, austria, april 13-17, 2015.*

Arcuri, Andrea, & Briand, Lionel. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. *Pages 1–10 of: Software engineering (icse), 2011 33rd international conference on.* IEEE.

Arrieta, Aitor, Sagardui, Goiuria, & Etxeberria, Leire. 2014. A configurable test architecture for the automatic validation of variability-intensive cyber-physical systems. *Pages 79–83 of: Valid 2014: The sixth international conference on advances in system testing and validation lifecycle.*

Arrieta, Aitor, Sagardui, Goiuria, & Etxeberria, Leire. 2015. Test control algorithms for the validation of cyber-physical systems product lines. *Pages 273–282 of: Proceedings of the 19th international conference on software product line.* SPLC '15. New York, NY, USA: ACM.

Bak, Kacper, Diskin, Zinovy, Antkiewicz, Michal, Czarnecki, Krzysztof, & Wasowski, Andrzej. 2015. Clafer: unifying class and feature modeling. *Software & systems modeling*, 1–35.

Behjati, Razieh, Yue, Tao, Briand, Lionel, & Selic, Bran. 2013. Simpl: A product-line modeling methodology for families of integrated control systems. *Information and software technology*, **55**(3), 607 – 629. Special Issue on Software Reuse and Product Lines.

Benavides, David, Segura, Sergio, & Ruiz-Corts, Antonio. 2010. Automated analysis of feature models 20 years later: A literature review. *Information systems*, **35**(6), 615 – 636.

Berger, Thorsten, Rublack, Ralf, Nair, Divya, Atlee, Joanne M., Becker, Martin, Czarnecki, Kryzsztof, & Wasowski, Andrzej. 2013. A survey of variability modeling in industrial practice. *Pages 7:1–7:8 of: Variability modelling of software-intensive systems (vamos).*

Briand, Lionel, Nejati, Shiva, Sabetzadeh, Mehrdad, & Bianculli, Domenico. 2016. Testing the untestable: Model testing of complex software-intensive systems. *Pages 789–792 of: Proceedings of the 38th international conference on software engineering companion.* ICSE '16. ACM.

Bringmann, Eckard, & Kramer, Andreas. 2008. Model-based testing of automotive systems. *Pages 485 – 493 of: Proceedings of the 1st international conference on software testing, verification and validation.*

Chvatal, V. 1979. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, **4**(3), 233–235.

Derler, Patricia, Lee, Edward A., & Sangiovanni-Vincentelli, Alberto. 2011. Modeling cyber-physical systems. *Proceedings of the ieee (special issue on cps)*, **100**(1), 13 – 28.

Dukaczewski, Michael, Schaefer, Ina, Lachmann, Remo, & Lochau, Malte. 2013. Requirements-based delta-oriented spl testing. *Pages 49 – 52 of: 4th international workshop on product line approaches in software engineering, please 2013.*

Dziobek, Christian, Loew, Joachim, Przystas, Wojciech, & Weiland, Jens. 2008. *Functional variants handling in simulink models.* Tech. rept. Mathworks.

Eidson, John C., Lee, Edward A., & Matic, Slobodan. 2011. Distributed real-time software for cyber-physical systems. *Proceedings of the ieee*, **100**, 45–59.

FreeRTOS. 2014. *Freertos.* http://www.freertos.org/.

Garvin, BradyJ., Cohen, MyraB., & Dwyer, MatthewB. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical software engineering*, **16**(1), 61–102.

jean Bristeau, Pierre, Callou, Franois, Vissire, David, & Petit, Nicolas. 2011. *The navigation and control technology inside the ar.drone micro uav.*

Jensen, Jeff C., Chang, Danica, & Lee, Edward A. 2011 (July). A model-based design methodology for cyber-physical systems. *Pages 1666 – 1671 of: Wireless communications and mobile computing conference (iwcmc), 2011 7th international.*

Johansen, Martin Fagereng, Haugen, Øystein, & Fleurey, Franck. 2012. An algorithm for generating t-wise covering arrays from large feature models. *Pages 46–55 of: Proceedings of the 16th international software product line conference - volume 1.* SPLC '12. New York, NY, USA: ACM.

Kane, Aaron, Fuhrman, Thomas E., & Koopman, Philip. 2014. Monitor based oracles for cyber-physical system testing: Practical experience report. *Pages 148–155 of: 44th annual*

IEEE/IFIP international conference on dependable systems and networks, DSN 2014, atlanta, ga, usa, june 23-26, 2014.

Khakpour, Narges, & Mousavi, Mohammad Reza. 2015. Notions of Conformance Testing for Cyber-Physical Systems: Overview and Roadmap (Invited Paper). *Pages 18–40 of:* Aceto, Luca, & de Frutos Escrig, David (eds), *26th international conference on concurrency theory (concur 2015).* Leibniz International Proceedings in Informatics (LIPIcs), vol. 42. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Kuhn, Rick, Kacker, Raghu, Lei, Yu, & Hunter, Justin. 2009. Combinatorial software testing. *Computer*, **42**, 94–96.

Lee, Edward A. 2007 (May). *Computing foundations and practice for cyber-physical systems: A preliminary report.* Tech. rept. UCB/EECS-2007-72. University of California, Berkeley.

Lee, Edward A. 2008. Cyber physical systems: Design challenges. *Pages 363–369 of: Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on.* IEEE.

Lee, Edward A., & Seshia, Sanjit A. 2015. *Introduction to embedded systems - a cyber-physical systems approach.* 2 edn. Lee and Seshia.

Lity, S., Lochau, M., Schaefer, I., & Goltz, U. 2012. Delta-oriented model-based spl regression testing. *Pages 53 – 6 of: 3rd international workshop on product line approaches in software engineering, please 2012.*

Lopez-Herrejon, Roberto E., Linsbauer, Lukas, & Egyed, Alexander. 2015. A systematic mapping study of search-based software engineering for software product lines. *Information and software technology*, **61**, 33 – 51.

Lu, Hong, Yue, Tao, Ali, Shaukat, & Zhang, Li. 2016a. Integrating search and constraint solving for nonconformity resolving recommendations of system product line configuration. *Pages 57–68 of: Ieee international conference on software testing, verification and validation (icst).*

Lu, Hong, Yue, Tao, Ali, Shaukat, & Zhang, Li. 2016b. Model-based incremental conformance checking to enable interactive product configuration. *Information and software technology*, **72**(04/2016), 68–89.

Lu, Z., Bezemer, M. M., & Broenink, J. F. 2015. Model-driven design of simulation support for the terra robot software tool suite. *Pages 143 – 158 of: Communicating process architectures 2015.*

Manz, Christian, Schulze, Michael, & Reichert, Manfred. 2014 (April). An approach to detect the origin and distribution of software defects in an evolving cyber-physical system. *In: Workshop on emerging idears and trends in engineering of cyber-physical systems (eitec '14).*

MathWorks. 2016 (March). *http://es.mathworks.com/discovery/cyber-physical-systems.html.*

Matinnejad, Reza, Nejati, Shiva, Briand, Lionel C., & Bruckmann, Thomas. 2016. Automated test suite generation for time-continuous simulink models. *Pages 595–606 of: Proceedings of the 38th international conference on software engineering.* ICSE '16. New York, NY, USA: ACM.

Mjeda, Anila. 2013. *Standard-compliant testing for safety-related automotive software.* Ph.D. thesis, University of Limeric.

Mohaqeqi, Morteza, Mousavi, Mohammad Reza, & Taha, Walid. 2014. Conformance testing of cyber-physical systems: A comparative study. *ECEASST*, **70**, 1–16.

Mosterman, Pieter J., & Zander, Justyna. 2016. Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems. *Software & systems modeling*, **15**(ISSN 1619-1366), 5–16.

Mosterman, Pieter J., Sanabria, David Escobar, Bilgin, Enes, Zhang, Kun, & Zander, Justyna. 2014. Automating humanitarian missions with a heterogeneous fleet of vehicles. *Annual reviews in control*, **38**(2), 259–270.

Mueller, Wolfgang, Becker, Markus, Elfeky, Ahmed, & DiPasquale, Anthony. 2012. Virtual prototyping of cyber-physical systems. *Pages 219 – 226 of: Asia and south pacific design automation conference.*

Nie, Kunming, Yue, Tao, Ali, Shaukat, Zhang, Li, & Fan, Zhiqiang. 2013. Constraints: The core of supporting automated product configuration of cyber-physical systems. *Pages 370–387 of: Acm/ieee 16th international conference on model driven engineering languages and systems.*

Pérez, Beatriz, Polo, Macario, & García, Ignacio. 2009a. Model-driven testing in software product lines. *Pages 511 – 514 of: Proceedings of the 2009 ieee international conference on software maintenance (icsm 2009).*

Pérez, Beatriz, Polo, Macario, & Piattini, Mario. 2009b. Towards an automated testing framework to manage variability using the uml testing profile. *Pages 10–17 of: Ast.*

Perrouin, Gilles, Sen, Sagar, Klein, Jacques, Baudry, Benoit, & Le Traon, Yves. 2010. Automated and scalable t-wise test case generation strategies for software product lines. *Pages 459–468 of: 2010 third international conference on software testing, verification and validation.* IEEE.

Piketec. 2015 (July). *Tpt - model-based testing of embedded control systems.*

Polzer, A., Merschen, D., Botterweck, G., Pleuss, A., Thomas, J., Hedenetz, B., & Kowalewski, S. 2012. Managing complexity and variability of a model-based embedded software product line. *Innovations and systems and software engineering*, **8**(1), 35 – 49.

Pure-Systems. 2014 (July). *pure::variants.* http://www.pure-systems.com.

Robinson, William. 2010. A roadmap for comprehensive requirements modeling. *Computer*, **43**(5), 64–72.

Sánchez, Ana B., Segura, S., & Ruiz-Cortés, Antonio. 2014. A comparison of test case prioritization criteria for software product lines. *Pages 41–50 of: Ieee international conference on software testing, verification, and validation.*

Shi, Jianhua, Wan, Jiafu, Yan, Hehua, & Suo, Hui. 2011. A survey of cyber-physical systems. *Pages 1–6 of: Proc. of the int. conf. on wireless communications and signal processing.*

Shokry, H., & Hinchey, M. 2009. Model-based verification of embedded software. *Computer*, **42**(4), 53 – 59.

Thiel, Steffen, & Hein, Andreas. 2002. Systematic integration of variability into product line architecture design. *Pages 130–153 of: Splc.*

Thuem, Thomas, Kastner, Christian, Benduhn, Fabian, Meinicke, Jens, Saake, Gunter, & Leich, Thomas. 2014. Featureide: An extensible framework for feature-oriented software development. *Science of computer programming*, **79**, 70 – 85.

Tidwell, Terry, Gao, Xiuyu, Huang, Huang-Ming, Lu, Chenyang, Dyke, Shirley, & Gill, Christopher. 2009. Towards configurable real-time hybrid structural testing: A cyber-physical systems approach. *Pages 37–44 of: Isorc.*

Wan, Jiafu, Suo, Hui, Yan, Hehua, & Liu, Jianqi. 2011. A general test platform for cyber-physical systems: Unmanned vehicle with wireless sensor network navigation. *Procedia engineering*, **24**, 123 – 127. International Conference on Advances in Engineering 2011.

Wang, Shuai, Gotlieb, Arnaud, Ali, Shaukat, & Liaaen, Marius. 2013a. Automated test case selection using feature model: An industrial case study. *Pages 237–253 of: Models.*

Wang, Shuai, Ali, Shaukat, & Gotlieb, Arnaud. 2013b. Minimizing test suites in software product lines using weight-based genetic algorithms. *Pages 1493 – 1500 of: Proceedings of the 2013 genetic and evolutionary computation conference.*

Wang, Shuai, Buchmann, David, Ali, Shaukat, Gotlieb, Arnaud, Pradhan, Dipesh, & Liaaen, Marius. 2014. Multi-objective test prioritization in software product line testing: An industrial case study. *Pages 32–41 of: Proceedings of the 18th international software product line conference - volume 1.* SPLC '14. New York, NY, USA: ACM.

Wang, Shuai, Ali, Shaukat, & Gotlieb, Arnaud. 2015. Cost-effective test suite minimization in product lines using search techniques. *Journal of systems and software*, **103**(0), 370 – 391.

Wang, Shuai, Ali, Shaukat, Gotlieb, Arnaud, & Liaaen, Marius. 2016. A systematic test case selection methodology for product lines: results and insights from an industrial case study. *Empirical software engineering*, 1–37.

Wetter, Michael, & Haves, Philip. 2008. A modular building controls virtual test bed for the integration of heterogeneous systems. *Pages 69–76 of: Proceedings of the 3rd simbuild conference.*

Zander-Nowicka, Justyna. 2007. Reactive testing and test control of hybrid embedded software. *Pages 45–62 of: Proceedings of the 5th workshop on system testing and validation.*

Zander-Nowicka, Justyna. 2008. *Model-based testing of real-time embedded systems in the automotive domain.* Ph.D. thesis, Technical University Berlin.

Zhang, Man, Selic, Bran, Ali, Shaukat, Yue, Tao, Okariz, Oscar, & Norgren, Roland. 2016. Understanding uncertainty in cyber-physical systems: A conceptual model. *Pages 247–264 of: European conference on modelling foundations and applications.* Springer.