Gaining Certainty About Uncertainty

Testing Cyber-Physical Systems in the Presence of Uncertainties at the Application Level

Martin A. Schneider(✉), Marc-Florian Wendland, and Leon Bornemann

Fraunhofer FOKUS, Berlin, Germany
martin.schneider@fokus.fraunhofer.de

Abstract. A cyber-physical system (CPS) comprises several connected, embedded systems and is additionally equipped with sensors and actuators. Thus, CPSs can communicate with their cyber environment and measure and interact with their physical environment. Due to the complexity of their operational environment, assumptions the manufacturer have made may not hold in operation. During an unforeseen environmental situation, a CPS may expose behavior that negatively impactsits reliability. This may arise due to insufficiently considered environmental conditions during the design of a CPS, or – even worse – it is impossible to anticipate such conditions. In the U-Test project, we are developing a configurable search-based testing framework that exploits information from functional testing and from declarative descriptions of uncertainties. Itaims at revealing unintended behavior in the presence of uncertainties. This framework enables testing for different scenarios of uncertainty and thus, allows to achieve a certain coverage of those, and to find unknown uncertainty scenarios.

Keywords: Cyber-Physical systems · Reliability · Search-based testing · Uncertainty · UML state machines

1 Introduction

Cyber-physical systems (CPS) are increasingly affecting our daily lives, e.g. in form of an autopilot of airplanes and autonomous cars, medical devices such as insulin pumps, or less visible in logistic centers that are receiving, storing and distributing goods. They often perform safety-critical tasks (as for autonomous vehicles and medical devices) or mission-critical tasks (as for logistic centers). Due to this criticality and their impact on our daily lives, it is even more important that CPSs work reliably, otherwise health or business is at risk.

Due to their nature, CPS interact with their cyber environment as well as with their physical environment. Along with the increasing connectivity and pervasiveness of

The research leading to these results has also received funding from the European Union's Horizon 2020 Programme under grant agreement no. 645463.

J. Großmann et al. (Eds.): RISK 2016, LNCS 10224, pp. 129–142, 2017.
DOI: 10.1007/978-3-319-57858-3_10

CPSs, the complexity of such interaction increases as well and is getting more complex, in particular for the physical world. Manufacturers cannot predict all circumstances, in particular with respect to the physical world, CPSs are exposed to. They have to make assumptions to make the design and development of CPSs manageable and affordable. Incomplete knowledge leads to uncertainties about their assumptions. If such an assumption fails while a CPSs is in operation, it may heavily impact its reliability and may harm human beings in their environment. Hence, finding uncertainties and testing CPSs in their presence is inevitable to increase their reliability. Since the complexity of the environment and the uncertainties of manufacturers' assumptions are difficult to grasp, traditional testing approaches are not sufficient and have to be adapted to overcome these issues.

In this paper, we propose a search-based approach to testing CPSs that copes with the challenges of the complexity of the environment and the implicit uncertainties of manufacturers' assumptions. The proposed approach provides means to declaratively describe uncertainties that are already known, e.g.due to analysis or from field tests. These descriptions are then used to bootstrap the search-based algorithm, to confine the search space and to find new uncertainties. The presented approach is subject to ongoing work done under the European research project U-Test[1].

The remaining paper is organized as follows: Sect. 2 discusses related work relevant for this paper, Sect. 3 introduces the uncertainty taxonomy used as a conceptual model for declarative description of uncertainties with respect to CPSs and catches a glimpse on declarative descriptions of uncertainties by means of an uncertainty taxonomy. Section 4 describes the proposed methodology for uncertainty testing of CPS. Section 5 closes with a conclusion and future work.

2 Related Work

2.1 Uncertainty

The term uncertainty has several meanings in different sciences and contexts as pointed out by Ramirez et al. [1]. While we are not specifically interested in the meaning in the science of psychology and economics and can ignore them, even in the field of systems engineering this term has different meanings with respect to latent or unknown properties and behaviors of a software system [2] or from the perspective of assumptions upon a certain goal [3].

However, a recognized article from Walker et al. [4] defines uncertainty as "any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system". This definition takes into account the fact that there might be different kinds of knowledge beside completely deterministic knowledge. Ramirez et al. [1] defines uncertainty from the perspective of a dynamically adaptive system as a system state of incomplete or inconsistent knowledge such that it cannot decided which environmental or system configuration holds.

[1] http://www.u-test.eu.

Refsgaard et al. [5] specify incomplete, inaccurate, unreliable, inconclusive, or potentially false information sources for uncertainty. Ramirez et al. [1] collected several sources of uncertainties from the literature, e.g. missing or ambiguous requirements, false assumptions, unpredictable entities or phenomena in the execution environment, incomplete or inconsistent information caused by imprecise, inaccurate, and unreliable sensors. Thus, uncertainties can be introduced in the requirements, design, and runtime phase.

Walker [4] also introduces the classification in epistemic and variability uncertainties and called this the nature of an uncertainty: epistemic uncertainty results from missing knowledge whereas variability uncertainty results from the variability, for instance in human and natural systems, and is also called aleatory, stochastic, or ontological uncertainty. Erkoyuncu et al. [6] and Refsgaard et al. [5] also use the terms epistemic uncertainty and stochastic or aleatoric uncertainty in order to express the unpredictability of an event.

According to the classification of nature of uncertainties from Walker [4], Erkoyuncu et al. [6] describe the characteristics of epistemic uncertainties by a lack of knowledge. Further research may increase the amount of knowledge and thus, reduce epistemic uncertainty. In contrast, aleatoric uncertainties are characterized to be stochastic and random where the uncertainty cannot be reduced by further research.

In order to determine the knowledge level about an uncertainty, Walker [4] introduced a scale reaching from statistical uncertainty to total ignorance: *Statistical uncertainty* can be described in statistical terms. *Scenario uncertainty* is characterized by scenarios that indicate what might happen in the future, and what the effects to the system are. *Recognized ignorance* means that functional relationships are nearly unknown and there is no significant scientific basis for developing acceptable scenarios. However, for

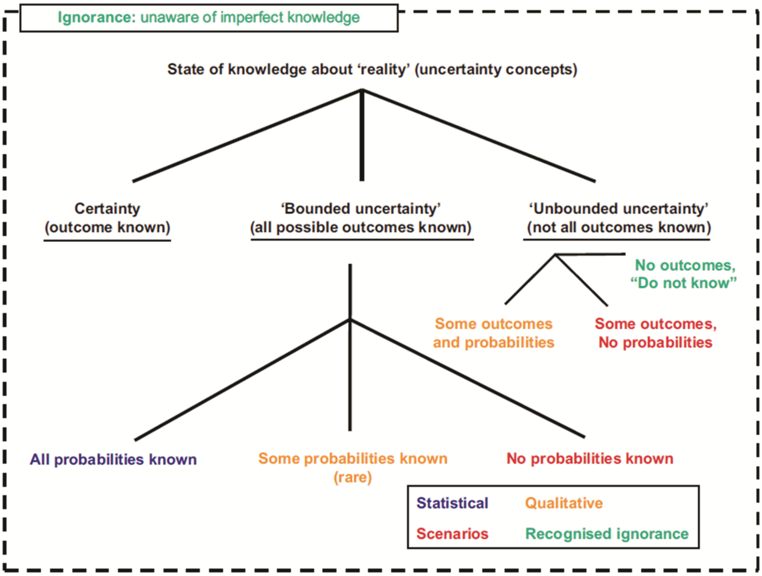

Ignorance: unaware of imperfect knowledge
State of knowledge about 'reality' (uncertainty concepts)
Certainty (outcome known)
'Bounded uncertainty' (all possible outcomes known)
'Unbounded uncertainty' (not all outcomes known)
No outcomes, "Do not know"
Some outcomes and probabilities
Some outcomes, No probabilities
All probabilities known
Some probabilities known (rare)
No probabilities known
Statistical
Qualitative
Scenarios
Recognised ignorance

Fig. 1. Brown's taxonomy of imperfect knowledge adapted by Refsgaard et al. [5]

reducible uncertainties, this missing knowledge can be investigated in order to shift such recognized ignorance to scenario uncertainty or even statistical uncertainty. *Total ignorance* comprises all the uncertainties that one is aware of but have no or only little knowledge, those we do not know, in principle as well as those uncertainties we are not aware of.

Refsgaard [5] merged Walker's knowledge level with Brown's [7] spectrum of confidence by a taxonomy of imperfect knowledge as shown in Fig. 1. This taxonomy distinguishes uncertainties by the knowledge of possible outcomes and the probabilities of the different outcomes. Refsgaard [5] mapped this taxonomy to Walker's scale described above.

However, as discussed by Erkoyuncu [6], there are also opinions that uncertainties do not have a probability assigned and this is the main distinction between uncertainties and risks[8, 9]. Uncertainty is considered as a source of risk [10]. Erkoyuncu [6] distinguishes uncertainty from risk by the lack of any outcome predictability – in contrast to Walker and Refsgaard – and that uncertainty covers positive outcomes while risk only covers negative outcomes.

2.2 Mutation Testing, Fault-Based Test Generation and Search-Based Testing

Mutation testing [11] and mutation analysis [12] are techniques to introduce faults either in the implementation, i.e. source code, or in the specification, e.g. models, to assess the quality of test cases and test suites. Mutation operators are considered as fault models that are applied to code or models and can be used to generate test cases [13, 14]. Higher order mutation testing uses combinations of mutation operators to find real bugs [15]. Mutation operators are specific to a modelling or programming language and thus, work on a syntactic rather than on a semantic level. Semantic-level mutation is considered as a relevant research-topic [15].

Search-based testing employs search-based software engineering algorithms for testing purposes. The typical search-space is usually too large for exhaustive testing. Thus, search-based testing employs metaheuristics [16, 17] to explore the search space more efficiently. Since those metaheuristics are generic techniques that requires the formulation of the problem as an optimization problem, a quality function is used to assess individual candidate solutions, e.g. test cases for search-based testing. A frequently used algorithm belonging to the class of search-based algorithm is the genetic algorithm [18] that employs mutation as used by mutation analysis, and additionally crossover and selection based on fitness values calculated by a quality function. The guidance by fitness values is one of several differences to mutation testing and higher order mutation testing. However, mutation and crossover is usually done on a syntactical level guided by the fitness calculation. Depending on the quality and appropriateness of the fitness function, even search-based testing may degenerate into random testing. Therefore, developing the fitness function has to be done carefully and may pose a significant challenge.

3 The Uncertainty Taxonomy: Declarative Descriptions of Uncertainty

In this paper, we consider uncertainties coming from the environment and accordingly, call them environmental uncertainties [19–23]. According to Cheng [20], environmental uncertainties come from the physical environment and the cyber environment. Uncertainties from the physical environment come from unforeseen or environmental conditions with a lack of knowledge about it and may also result from sensor failures or noisy environments [23]. Uncertainties from the cyber environment may result from malicious threats or unexpected (human) input [23].

In addition to environmental uncertainties, uncertainties of CPS may also occur within the technical infrastructure when connected embedded systems, sensors and actuators interact in an unforeseen way or errors occur in the communication infrastructure of a CPS. However, in this paper we consider uncertainties in the environment of a CPS since uncertainties in the infrastructure are different compared to environmental level uncertainties, are often originating from technical uncertainties within the CPS's technical infrastructure and have to deal with other aspects than the environment, e.g. elasticity and virtualization. Therefore, we assume the infrastructure was sufficiently tested and is working fine.

Environmental uncertainties may impact the application running on the technical infrastructure of a CPS either directly, if the application is confronted with invalid or unexpected data or behavior, e.g. from another system or a user, or indirectly because a sensor works correctly but its reading is tampered by physical circumstances such as smoke. Therefore, we refer to environmental uncertainties by the term *application level uncertainties*.

An application level uncertainty is constituted by a circumstance that does not comply with the specified or expected environmental behavior, and may not be foreseen by a CPS' manufacturer. If a CPS behaves in an undesired manner due to an uncertainty, i.e. in a way that negatively impacts its reliability, we call such a behavior an *uncertain behavior* to indicate that it results from an unforeseen or unexpected environmental condition. We distinguish between *known uncertain behaviors* that are known by analysis or field tests, i.e. before the system is deployed, and those that are not known a priori, and call them *unknown uncertain behaviors*. Accordingly, the corresponding uncertainties are referred to by the terms *known uncertainty* for those that may be known a priori and *unknown uncertainty* for uncertainties one not aware of.

To describe known uncertainties, we developed a taxonomy that allows to specify their characteristics. The purpose of the taxonomy is twofold. On one hand, by providing a scheme for properties of uncertainties may support their analysis and thus, their understanding. Furthermore, we use it to test for these, in order to find uncertainty scenarios and even new uncertainties as explained in Sect. 4.

The characteristics of uncertainties comprise properties such as the origin, i.e. whether the uncertainty occurs in the physical or the cyber environment, or its causes, on a high-level distinguishing between human behavior, natural process and technological process. Figure 2 provides an excerpt of the taxonomy.

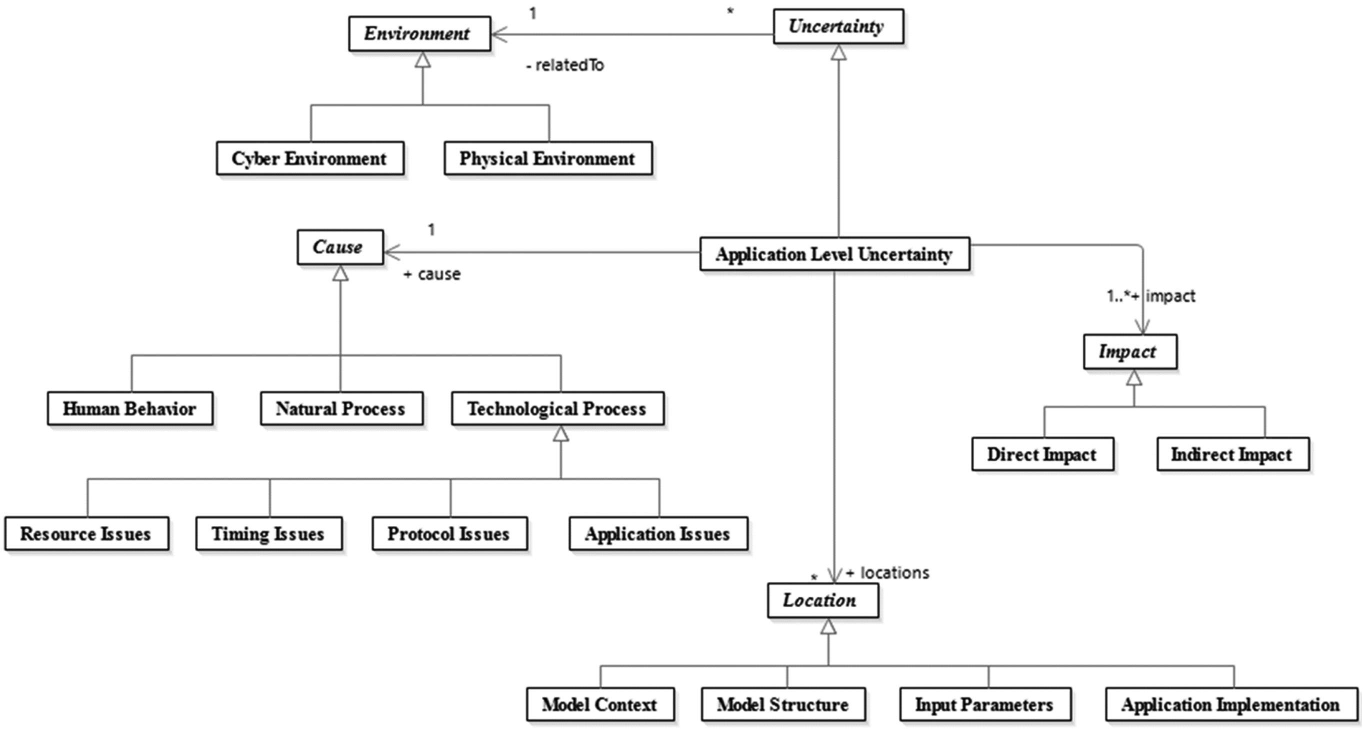


Fig. 2. Excerpt of the taxonomy for application level uncertainties

We aim at characterizing uncertainties systematically by its *origin*, its *cause*, its *location* and its *impact* on a system. The origin is related to the *environment* where the uncertainty may occur. This is denoted by the environment and may either be the *cyber environment* or the *physical environment*. The *cause* is used to characterize the originator of an uncertainty. We basically distinguish human behavior, natural process and technological processes as cause for an uncertainty. A person that regularly interacts with a cyber-physical system may show behavior that may contribute to an uncertainty. Natural processes are usually related to the physical environment where randomness may come into play, a very simple example may be radiation that may impact the readings of a sensor. Technological processes are distinguished in those related to resources, timing, protocols and the application itself.

Timing issues may result from the uncertainty whether a system is working with the expected performance while abstracting of real time, e.g. by a cycle counter.

Resource issues are reflecting different issues with respect to the cyber world as well as to the real world, e.g. resource competition meaning that two instances are working on or using the same resources and thus interfering with each other. *Resource location* means that the expected resource is not where it is expected to be. *Insufficient resources* comprises uncertainties regarding the demanded and the provided resources where the demanded resources are higher than the provided resources, e.g. a missing resource item in the physical environment or insufficient CPU resources with respect to the cyber environment.

Protocol issues summarize different uncertainties with respect to communication protocols. *Interoperability issues* occur if the specification of a communication protocol is ambiguous and two communication partners differ in their protocol implementation *Faulty protocol implementation* is a result of an incorrect protocol implementation leading to communication errors between two communication partners, e.g. different components of the application or between the application and the infrastructure of the cyber-physical system.

Application issues comprise uncertainties inherent to the application itself. Communication issues with platform is referring to situations where the application fails to communicate with platform devices, maybe resulting from a faulty application configuration. Functional faults are traditional implementation bugs within the application.

The concept *impact* represents the impact of an uncertainty from the environment to the impacted element of a cyber-physical system, such as hardware and/or application.

4 Uncertainty Testing

This section introduces the proposed approach for search-based uncertainty testing by (i) providing an overview how to create models suitable for uncertainty testing in the first subsection and (ii) describing how the proposed approach evolve such models, aiming at revealing uncertain behavior by eventual test case generation and execution, in the second subsection. Since we employ a genetic algorithm, we require a quality function that provides a measure to evaluate whether we are about to find uncertain behavior of a system under test (SUT). To do so, we provide a model-based framework to describe relevant behavioral characteristics of the SUT for fitness evaluation in the third subsection. The last subsection provides information that can be used to describe an exit criterion when to stop uncertainty testing in terms of coverage criteria (Fig. 3).

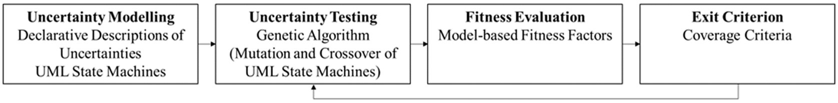


Fig. 3. Overview of uncertainty testing process steps

4.1 Modelling for Uncertainty Testing

Modelling for uncertainty testing requires two artifacts: modelled uncertainties and functional models in terms of UML state machines. The latter one can be easily obtained from a functional testing process performed in a model-based way. Such models can be reused for uncertainty testing and thus, reducing the effort to start the proposed approach. If functional descriptions in form of UML state machines do not exist, they can be created by anyone who has enough information on the requirement but does not need specific knowledge about uncertainties. The more challenging task is to describe characteristics of uncertainties. Such information can be obtained by a risk analysis approach or by obtaining information from tests in the field. Figure 4 provides an example of a UML state machine describing valid interaction of the environment with the SUT. It describes a simple geo-locating system that determines the positions of tags whose positions are determined through a set of locators. The locators receive the signal of a tag and the application calculates its position via triangulation. First, tags are configured for the system (transition ‘configureTag’) and locators mounted (transition ‘subsequentMonitoring’). After that, the system is calibrated (transition ‘calibrate’), and by changing the position of a tag (transition ‘setPosition’), it can be checked whether the tag’s position is correctly calculated by the system (transition ‘getAllPositions’).

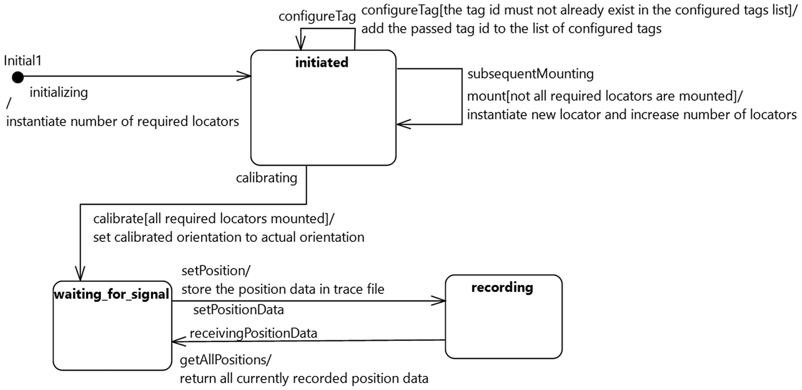
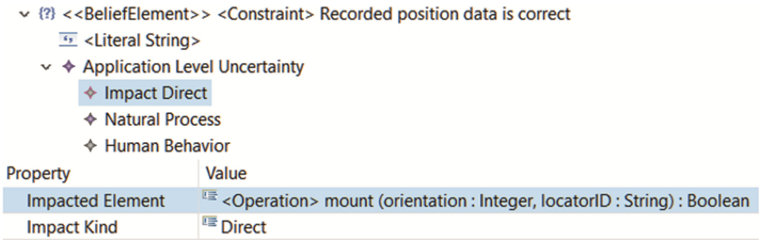

configureTag
configureTag[the tag id must not already exist in the configured tags list]/
add the passed tag id to the list of configured tags
Initial1
initiated
initializing
/
instantiate number of required locators
subsequentMounting
mount[not all required locators are mounted]/
instantiate new locator and increase number of locators
calibrating
calibrate[all required locators mounted]/
set calibrated orientation to actual orientation
setPosition/
store the position data in trace file
waiting_for_signal
recording
setPositionData
receivingPositionData
getAllPositions/
return all currently recorded position data

Fig. 4. UML state machine providing a functional description

The system may not calculate a tag's position correctly if a locator is mounted after the system has been calibrated. The corresponding uncertainty would influence the correct recording of position data. The uncertainty consists in the unmodified position of the locators, i.e. that locators are not remounted after calibration. Hence, the correct functionality may be discontinued by an application level uncertainty related to the *mount* operation on an already mounted locator (referred by the transition 'subsequent-Mounting') after the system has been calibrated. Figure 5 provides a description of such an uncertainty whose impact refers to this *mount* operation.



- <<BeliefElement>> <Constraint> Recorded position data is correct
    - <Literal String>
    - Application Level Uncertainty
        - Impact Direct
        - Natural Process
        - Human Behavior

Property	Value
Impacted Element	<Operation> mount (orientation : Integer, locatorID : String) : Boolean
Impact Kind	Direct

Fig. 5. Example of a modelled application level uncertainty

For this small example, it's enough information to perform uncertainty testing as described in the next subsection.

4.2 Evolving UML State Machines and Generating Test Cases

Uncertainties at the application level comprise all the stimuli from the environment of the SUT. The purpose of uncertainty testing is (i) discovering *known uncertain behaviors* resulting from uncertainties that may be known at design time, and (ii) discovering *unknown uncertain behaviors* that may occur in the presence of yet unknown uncertainties.

Since we do not know all the manifestations of an uncertainty and would like to reveal unknown uncertain behavior resulting from unknown uncertainties, we employ search-based techniques to efficiently walk through the input space. Aiming at measuring whether we are approaching an uncertainty that may expose known or unknown uncertain behavior or if we have already discovered one, we exploit different outputs of the SUT as inputs to a fitness function.

The modelled uncertainties and the functional models as described in the previous subsection form the basis for evolving state machines. Thus, we exploit the coupling effect [24]. The goal is to gain state machines of which at least one path reveals uncertain behavior.

Mutation. Mutation is performed on one hand using information from modelled uncertainties, and on the other hand independent from that by using information of the system provided by the test model. In contrast to mutation analysis and search-based testing, we take into account semantical information provided by modelled uncertainties for mutation and eventual test generation. Thus, we do not apply mutation on a syntactical level but on the semantic level as well. By this approach, we can reduce the search space further.

We apply mutations to transitions based on information from modelled uncertainties, evaluating its *impact* property that refers to a single operation or an interface containing operations. Several mutations of the same element are allowed, although a few combinations of mutations are excluded to generate executable test cases eventually, e.g. those combinations that do not lead to a new state machine, e.g. in case one mutation is the inverse of another mutation. Based on the literature [9, 18], we use the mutation operators as follows and adapted them to UML state machines.

- *Add Transition*: Adds a new transition by duplicating an existing one and setting a new source and target state.
- *Remove Transition*: Completely removes a transition.
- *Reverse Transition*: Swap source and target of a transition.
- *Change Source/Target of Transition*: Move the source/target of a transition to any other state.
- *Remove Trigger of Transition*: Transforms a transition to a completion transition.
- *Change Trigger of Transition*: Changes the operation of a transition's trigger to another one of the same interface.

Based on the example state machine depicted in Fig. 4 and an application level uncertainty depicted in Fig. 5, we can identify those transitions that refer to the operation *mount* as the uncertainty. Such a mutation looks as in Fig. 6.

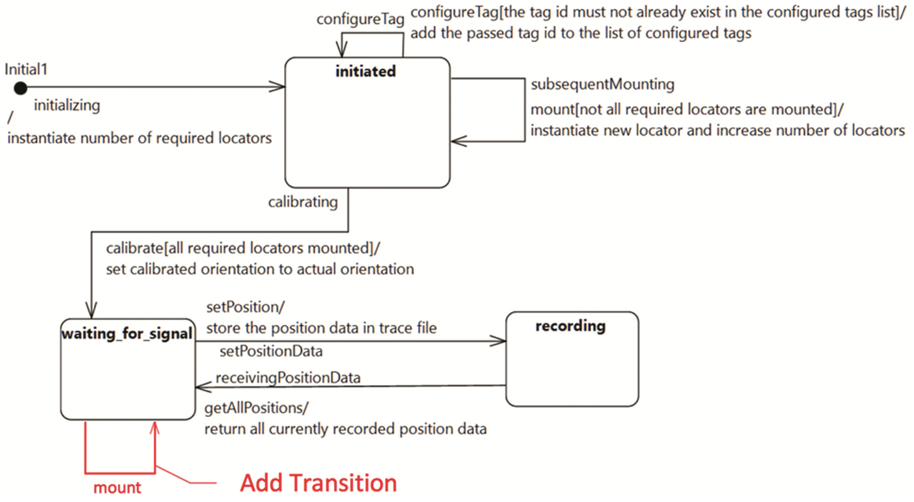

configureTag
configureTag[the tag id must not already exist in the configured tags list]/
add the passed tag id to the list of configured tags
Initial1
initiated
initializing
/
instantiate number of required locators
subsequentMounting
mount[not all required locators are mounted]/
instantiate new locator and increase number of locators
calibrating
calibrate[all required locators mounted]/
set calibrated orientation to actual orientation
setPosition/
store the position data in trace file
waiting_for_signal
recording
setPositionData
receivingPositionData
getAllPositions/
return all currently recorded position data
mount
Add Transition

Fig. 6. UML state machine mutated by adding a transition with trigger *mount*

Crossover. Mutations are the atomic piece of information to perform uncertainty testing. Therefore, for the recombination/crossover of state machines, we solely consider mutations instead of whole state machines. We propose to use the following crossover operators: *Combine all mutations of both parents*. This yield one new child UML state machine. *Uniform crossover*: swap *n* mutations of both state machines. This yield two new child state machines. This approach can be refined by swapping an unfit mutation of state machine A with a fit mutation of state machine B if the share at least one path. *Combine only the fittest path(s)*: This yields one new state machine with less mutations.

Test Case Generation. We generate test cases based on evolved UML state machines using Microsoft's Spec Explorer [30] that calculates all paths through it. To generate executable test cases, we use UML-based behavioral description of so-called execution invariants that describe those sequences that all test cases must respect. A simple example of such an execution invariant is that a system must be switched on before it can be configured. These execution invariants are different from system requirements since we would like to intentionally violate system requirements. Execution invariants represent those invariants whose violation is actually impossible and would lead to test cases that could not be executed against the system under test or that would impede evaluation of test cases.

Considering again the example state machine depicted in Fig. 4. It has two transitions named 'setPosition' to change the position of a tag and 'getAllPositions' to retrieve the position calculated by the SUT. Each time we change the position, we would require to retrieve the calculated position to decide whether it still matches sufficiently. Therefore, we would describe this as an execution invariant in form of a sequence diagram as shown in Fig. 7.

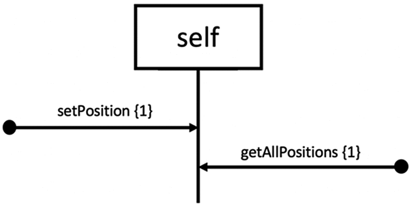


Fig. 7. Simple example of an execution invariant requiring that *getAllPositions* is called immediately after *setPosition*

4.3 Modelling Fitness Factors

To specify use-case specific factors, we provide stereotypes to identify elements that allow obtaining values from test runs (*FitnessFactorProviders*) that may be compared with an expected value by the corresponding counterpart (*ExplicitProvider*). These can be used for instance, to measure the distance between a measured position and the actual position. For measured values without any comparative value, *ImplicitProviders* can refer to them. An optional threshold can be specified together with a *metricGoal* that specifies whether the actual, measured value should be minimized, maximized or approach the threshold. In case of *ExplicitProviders*, the difference between the actual and the expected value can be minimized or maximized. System-specific factors can be identified by the same means. Furthermore, we use generic measures such as response time, CPU load, and memory consumption if we can obtain these values from the SUT. Figure 8 shows the different stereotypes for fitness factor descriptions.

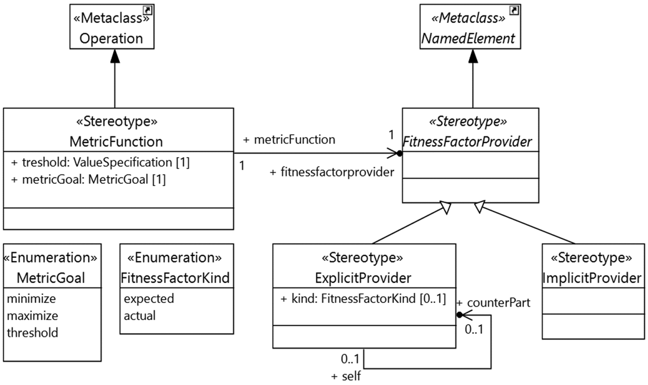


Fig. 8. Framework for model-based fitness factors description

With respect to the example given in Fig. 4, we would use the stereotype *ExplicitProvider* referring to the operation to retrieve the position that was actually set with the *kind* property set to the value *expected* to describe the expected position data, i.e. *setPosition*, and a second one referring to the operation that retrieves the position data calculated by the SUT with the *kind* property set to the value *actual*, i.e. the operation *getAllPositionData*. Since both values should be equal, the corresponding *MetricFunction* would have the *metricGoalminimize*, and an implementation would calculate the difference between the values provided by the *actual* and the *expected* fitness factor provider.

4.4 Metrics for Measuring the Progress of Uncertainty Testing

To measure the progress of uncertainty testing and to be able to provide an exit criterion to determine when the testing process should be stopped, we described an Uncertainty Space Coverage metric that would be first step towards this goal.

It measures all generations of evolved state machines related to a single uncertainty. The Uncertainty Space is spanned by all the possible mutation on triggers, guards and effect of transition in a UML state machine denoted by the variable N. If g is the number of steps of evolving a state machine, i.e. the number of generations (equal to the number of mutations), we can describe all the possible mutations by

$$NG(N, g) = \frac{g(g+1)}{2}N - 2\sum_{j=1}^{g}\sum_{k=1}^{j}(k-1)$$

that can be simplified to the following version:

$$NG(N,g) = \frac{g(g+1)}{2}N - 2\sum_{j=1}^{g}\frac{j(j+1)}{2} - j$$

The Uncertainty Space comprises all possible combinations:

$$UncertaintySpace(N,g) = \sum_{j=1}^{g} NG(N,g)$$

As expected, the uncertainty space may grow strongly with the number of mutations applied to a single state machine and strongly depends on the constant *N*.

5 Conclusion and Future Work

We introduced an approach for testing the reliability of CPS in the presence of uncertainty with the help of declarative descriptions and UML state machines evolved by a genetic algorithm. The approach aims at finding manifestations of known uncertainties and unknown uncertainties and the corresponding uncertain behaviors. We described execution invariants to ensure generation of executable test cases. Finally, we proposed a coverage criterion based on the uncertainty space appropriate for specifying an exit criterion for uncertainty testing.

Since this paper presents ongoing work, there is still a lot of work to do. Obviously, information used from uncertainty description is currently. For an effective approach, other kind of information should be obtained, particularly with respect to the cause of an uncertainty. Currently, the number of mutation operators is limited. Krenn et al. [11] provides an exhaustive specification of mutation operators for UML state machines. Since the uncertainty space is very huge, more ways to confine the process should be investigated. Eventually, the approach has to show its feasibility, effectiveness and efficiency by a thorough evaluation and compare it with traditional approaches.

References

1. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C.: A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-managing Systems, Piscataway, NJ, USA, pp. 99–108 (2012)
2. Goldsby, H.J., Cheng, B.H.C.: Automatically discovering properties that specify the latent behavior of UML models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems, pp. 316–330. Springer, Berlin Heidelberg (2010)
3. Welsh, K., Sawyer, P.: Understanding the scope of uncertainty in dynamically adaptive systems. In: Wieringa, R., Persson, A. (eds.) Requirements Engineering: Foundation for Software Quality, pp. 2–16. Springer, Berlin Heidelberg (2010)
4. Walker, W.E., et al.: Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support. Integr. Assess. **4**(1), 5–17 (2003)

5. Refsgaard, J.C., van der Sluijs, J.P., Højberg, A.L., Vanrolleghem, P.A.: Uncertainty in the environmental modelling process – a framework and guidance. Environ. Model Softw. **22**(11), 1543–1556 (2007)
6. Erkoyuncu, J.A., Roy, R., Shehab, E., Cheruvu, K.: Understanding service uncertainties in industrial product–service system cost estimation. Int. J. Adv. Manuf. Technol. **52**(9–12), 1223–1238 (2011)
7. Brown, J.D.: Knowledge, uncertainty and physical geography: towards the development of methodologies for questioning belief. Trans. Inst. Br. Geogr. **29**(3), 367–381 (2004)
8. Faro, D., Rottenstreich, Y.: Affect, empathy, and regressive mispredictions of others' preferences under risk. Manag. Sci. **52**(4), 529–541 (2006)
9. Knight, F.H.: Risk, Uncertainty and Profit. Courier Corporation (2012)
10. Emblemsvåg, J.: Life-Cycle Costing: Using Activity-Based Costing and Monte Carlo Methods to Manage Future Costs and Risks. Wiley, New Jersey (2003)
11. Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E., Brandl, H.: MoMut::UML model-based mutation testing for UML. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–8 (2015)
12. Fabbri, S.P.F., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Mutation analysis testing for finite state machines. In: Proceedings of 5th International Symposium on Software Reliability Engineering, pp. 220–229 (1994)
13. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Program mutation: a new approach to program testing. Infotech State Art Rep. Softw. Test. **2**, 107–126 (1979)
14. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241), pp. 46–54 (1998)
15. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
16. Luke, S.: Essentials of metaheuristics (2013). lulu.com
17. McMinn, P.: Search-based software test data generation: a survey: research articles. Softw. Test Verif. Reliab. **14**(2), 105–156 (2004)
18. Harman, M., Zhang, Y., Mansouri, S.A.: Search based software engineering: a comprehensive analysis and review of trends techniques and applications. King's College (2009)
19. Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems, pp. 468–483. Springer, Berlin Heidelberg (2009)
20. Tackling Uncertainty for Transportation Cyber-Physical Systems | CPS-VO. http://cps-vo.org/node/11229. Accessed 25 Sep 2016
21. NIST Foundations for Innovation for Cyber-Physical Systems. http://events.energetics.com/NIST-CPSWorkshop/. Accessed 25 Sep 2016
22. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., Knoester, D.B.: Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, pp. 568–571 (2011)
23. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.-M.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010)
24. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**(4), 34–41 (1978)