Generating test-cases from qualitative knowledge – Preliminary report

Franz Wotawa[*]
Technische Universität Graz, Institute for Software Technology
Inffeldgasse 16b/2, A-8010 Graz, Austria
wotawa@ist.tugraz.at

Abstract

In this paper we introduce a methodology for extracting test-cases from qualitative knowledge which represents the expected behavior of the environment of a system under test. Usually in software engineering test-cases are only derived from the requirements documentation and do hardly consider environmental constraints. This is especially problematic if a system like a mobile device has interactions with the environment which cannot be foreseen in advance in all details. An approach that is based on the behavior of the world which is external to the system would help to generate test cases which are realistic and capture the whole range of interactions. The use of qualitative reasoning for representing for example the physical world is an advantage because the underlying models capture all possible behaviors and thus guarantee completeness of the generated test-case set to some extent.

Introduction

The complexity of systems and software increases every year which is mainly caused by a strong demand for smarter products that have to provide more and more functionality. For example in the automotive industry the number of CPUs with control software running on it is still increasing. You should not wonder that the number of such control units onboard of a vehicle is likely to more than 40. Because of the fact that control units even implement different functionalities there is also heavy communication between them. Hence, complexity of the whole system increases and makes it difficult to construct such systems and to validate and verify them. Moreover, there is a tendency that less and less effort is spent in verification and validation (V & V) due to market requirements, e.g., pre-defined dates for introducing a new product, and economical requirements, e.g., labor costs and expected revenue. Such considerations of course do not apply in the development of safety-critical systems like vehicles.

A proposed solution to overcome the mentioned problems is to automate testing to some extent. This can be done by providing tools for test execution and test-case generation. The former helps to reduce effort in running test-cases especially in cases of several product releases and where regression testing is necessary. Such test execution tools also provide statistical information regarding failing test-cases and other measures. Test-case generation tools are used to generate test-cases from specifications or from the source code which fulfill certain requirements like coverage or maximizing the mutation score. One important aspect of development of good test-cases especially for system tests as part of the validation procedure is that test-cases should reflect possible interactions between the system and its environment. For example, if there is no pre-defined order for entering data into a system, different sequences have to be tested. Moreover, unexpected but possible interactions have to be tested. For example, what happens when killing a process which has an open transaction with a database? If the database software is correct, the transaction is not allowed to be confirmed in order to ensure integrity. In this case a rollback procedure would be necessary.

In this paper, we focus on the generation of test-cases from qualitative models. The reason for that is the need for test-cases which test not only specified requirements but also unexpected but still possible interactions of the system with the environment. Qualitative reasoning is appropriate for that purpose especially when generating test-cases for embedded systems that have to work more or less autonomously. One reason is the fact that QR models capture all possible behaviors which make them a perfect choice for explanation and in our case test-case generation. For test-case generation using QR models we have two application areas in mind:

1. Systems that have to have knowledge of its domain in order to fulfill a certain task. Such systems might be control or decision support systems. For example, if we want to have an automated advisor that helps us in deciding actions for protecting the ecosystem (or some parts of it like a river), we have to provide test-cases which describe a range of possible scenarios. For the generation of such test-cases someone obviously has to have knowledge about the considered ecosystems. If we want to automate test-case generation, the test-case generator has to use knowledge about possible behaviors of the ecosystem.

[*]This work has been supported by the FIT-IT research project *Self Properties in Autonomous Systems project (SEPIAS)* which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology and the FFG.

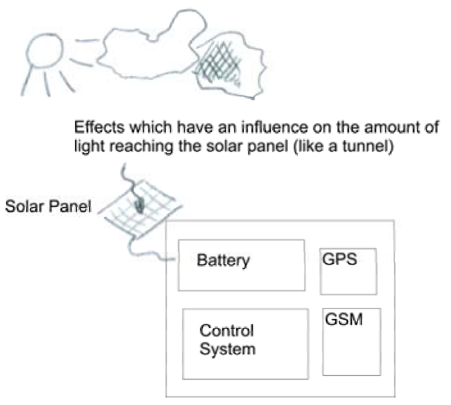
Effects which have an influence on the amount of light reaching the solar panel (like a tunnel)
Solar Panel
Battery
GPS
Control System
GSM

Figure 1: An mobile embedded system with a solar panel for providing electricity

2. The second scenario considers systems that implement a certain functionality and interact with the surrounding environment. In this case knowledge about the behavior of the environment can be used to generate test-cases that are unexpected. In this case the environment knowledge is not used to test the basic functionality of the system but to test the interaction with its environment. This is especially important for system tests.

In the paper we will use an example from the second application area but the ideas should be applicable as well for the other area.

Basic idea

To illustrate the basic idea of our approach we use a small example which is depicted in Figure 1. In this example a solar panel is used to provide electrical power of mobile embedded systems. The system comprises a battery which is loaded whenever the solar panel is providing energy, and other components like a GPS for measuring the global position and a GSM module for communicating with a server. Because the systems have to fulfill a task, e.g., sending its position to the server every minute, it requires electrical power. During the night the power is provided only by the battery and thus the battery is discharged. Of course the system should be designed in a way that the capacity of the battery is large enough to provide electrical power for the whole night. However, the system's designer might not considered all possible situations when computing the required capacity. For example, there can be several cloudy days where the solar panel does not provide enough power. The solar panel might produce no electricity because the device is in a building or a tunnel. Because of aging the battery capacity and the solar panel capabilities are decreasing. Moreover, the device is used in the north of the globe during winter where there is almost no sun light during a day.

A qualitative model describing the mentioned situation has to describe the relationships between the important entities. In particular, we introduce the following variables together with their domains:

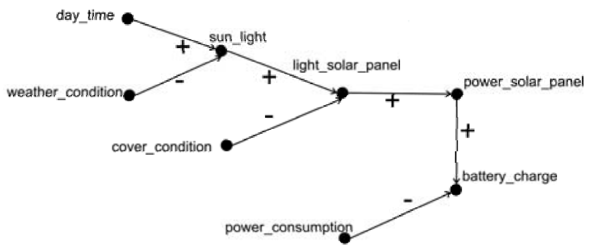

Figure 2: The cause-effect relationships of the solar panel example

`day_time`	{day, night}
`weather_condition`	{sunny, cloudy }
`sun_light`	{ no, medium, full }
`cover_condition`	{ yes, no }
`light_solar_panel`	{ no, medium, full }
`power_solar_panel`	{ no, medium, full }
`battery_charge`	{ empty, medium, full }

The `day_time` variable indicates whether it is currently day or night. `weather_condition` is for stating the current weather situation where in our case only the amount of clouds is required. From `day time` and `weather_condition` we can derive the amount of sun light (`sun_light`) at the ground level of the earth. If there the solar panel is not covered by anything,e.g., `cover_condition` is false, `sun_light` has to be equivalent to the light available directly at the panel (`light_solar_panel`). Otherwise, there is no light available for the solar panel to produce electrical power. Depending on the light at the solar panel more or less power (`power_solar_panel`) can be produced. And from the energy we can derive the charging status of the battery. Figure 2 depicts the cause-effect relationship between the variables. Note that this model is of course simplified and does not handle all possibilities. The model can be extended to represent parts of the internal behavior of the mobile embedded system, e.g., the influence of the subsystems to the status of the battery charge capacity.

This model can be formally represented using a qualitative reasoning methodology like Qualitative Process Theory (Forbus 1984) or Qualitative Simulation (Kuipers 1986). A qualitative reasoning engine like Garp 3 (Bredeweg *et al.* 2006) can be used to produce simulation runs. For this paper we assume that the qualitative model together with a reasoning engine is available.

For example, we know that the weather condition has a negative impact on the amount of sun light if we assume the following order `sunny<cloudy` and `no<medium<full` defined for the domains of the corresponding variables. If the weather condition increases (which means the amount of clouds is increasing), then the sun light is decreasing. Using the QSIM representation we would write $M^{-}(\texttt{weather\_condtion}, \texttt{sun\_light})$. For other relationships between variables we are able to define similar relationships. Moreover, we specify a relationship between the state of the charge of the battery and the power consumption of the device in order to compute the

discharging of the battery whenever a consumer is added. If a device is added to a battery there is a negative impact on the battery's charge capacity:

$$M^{-}(\texttt{power\_consumption}, \texttt{battery\_charge}).$$

Hence, we might be interested in finding a behavior that leads to a value of no for battery_charge when assuming that the battery is fully charged at the beginning.

In order to compute such a behavior we have to have information about the values of some variables, the model, and a test condition. The latter specifies the behavior to be searched by simulating the model. For our example, one test-case might be of the form:

$$\left\langle \begin{array}{c} (\texttt{day\_time}, \texttt{night}) \\ \ldots \\ (\texttt{power\_consumption}, \texttt{yes}) \\ (\texttt{battery\_charge}, \texttt{full}) \end{array} \right\rangle_0 \left\langle \begin{array}{c} (\texttt{day\_time}, \texttt{night}) \\ \ldots \\ (\texttt{power\_consumption}, \texttt{yes}) \\ (\texttt{battery\_charge}, \texttt{medium}) \end{array} \right\rangle_1 \left\langle \begin{array}{c} (\texttt{day\_time}, \texttt{night}) \\ \ldots \\ (\texttt{power\_consumption}, \texttt{yes}) \\ (\texttt{battery\_charge}, \texttt{empty}) \end{array} \right\rangle_2$$

Note that the values of other variables over time are not given for this example. The test-case states that the battery can be discharged during the night. Such a test-case is an abstract test-case because it does not provide any information about the quantities, e.g., the length of night. Hence, in order to create an executable test-case we have to instantiate or refine the abstract test-case. For example, we might define that a night lasts for 12 hours and thus we have to test whether the device is designed to fulfill its expected functionality during the night or even longer.

Definitions and algorithms

Generating test-cases automatically requires the availability of a formal model (or the source code). In our case we assume that the formal model is a qualitative model which captures the important aspects of the environment and of the system to be tested. In the example given in the previous section we introduced variables for stating properties of the solar panel, the battery as well as the power consumption which correspond to parts of the system. In the following definition of the model to be used for test-case generation we distinguish the environment model from the system model. The purpose of separating the model is that re-use of models is supported. It is very likely that the environment part of the model can be used together with models of different systems under test.

Definition 1 (Test-case model) *A test-case model is a tuple $(QM \cup QS, C_I, C_T)$ where QM is the qualitative model of the environment, QS is the partial qualitative model of the system, C_I is a set of constraints of input variables, and C_T is a set of test-case constraints.*

Note that in the definition of a test-case model a test purpose can be specified using C_T. The test purpose gives information about which test-case to generate. In our previous example we wanted to have a test-case which leads to an empty battery. Formally, we would write the input and test-case constraint of our example as follows:

$$C_I = \{(\texttt{day\_time}, \texttt{night})\} \cup C$$
$$C_T = \{(\texttt{battery\_charge}, \texttt{empty})\}$$

In this example C denotes the set of constraints stating that it is not possible for variables to have more than one value assigned at the same time.

Given a test-case model we now are interested in specifying a test-case. We do this by defining a test-case as the outcome of a qualitative simulation where all input constraints have to be fulfilled and where the test purpose is reached.

Definition 2 (Abstract test-case) *Given a test-case model $(QM \cup QS, C_I, C_T)$ and a qualitative simulator QEXEC. An abstract test-case t is the result of a run of the simulator on the model where the inputs do not contradict C_I and t covers all requirements of C_T, i.e., $t = QEXEC(QM \cup QS, i)$ where $i \cup C_I$ is consistent and $endState(t) \models C_T$.*

In the definition of abstract test-cases the constraints which apply on the test-case, i.e., the test-case requirements C_T are somehow stronger that the constraints for the inputs. A reason is that we expect a test-case to entail all requirements whereas consistency checks are enough for the input. The test purpose has to be reached in all cases.

The following algorithm computes an abstract test-case from the given test-case model. The algorithm does not only compute one test-case for each run of the QR simulation engine but combines different runs. A reason for extending the computation is to generate larger test-cases which are likely to capture a more complex behavior. Moreover, we further be able to represent information about possible inputs over time using C_I. For example, we might be interested in testing the system over several days. Hence, we have to specify that after the night we start with a day which is followed by a night and so on. Such knowledge is assumed to be captured by C_I.

Algorithm *AbstractTestCase*
Inputs: A test-case specification $(QM \cup QS, C_I, C_T)$
Output: An abstract test-case

1. Let t be the empty test-case sequence and let b be the empty behavior.
2. Choose inputs i which are consistent with the input constraints C_I and the last state of the behavior b computed in the previous run.
3. Select a behavior b which is computed by calling a QR engine on model $QM \cup QS$ and inputs i.
4. If no new behavior b exists, then return t.
5. Let t be t extended with b, i.e., $t := t + b$.
6. If t fulfills the criteria for test-cases C_T, then return t. Otherwise, go to 2

The algorithm ***AbstractTestCase*** obviously computes an abstract test-case accordingly to our definition. The algorithm halts if a test-case that entails the test purpose C_T can be found or if the whole search space has been explored. The latter cannot be guaranteed in general. However, in practice someone would specify a boundary which when reached terminates the computation. The complexity of the algorithm depends on the complexity of the simulator and the size of the model in terms of variables and their domains.

AbstractTestCase can be used to compute a set of test-cases by calling it more often. In this case we might change the input constraints and test purpose. If we do not change the constraints, we also expect that ***AbstractTestCase*** returns different solutions because we assume the selection of inputs and behaviors to be random. In a practical implementation someone might make this selection deterministic. In this case the algorithm can also be extended to compute different solutions by exploring the search space in a breadth-first manner.

Refining test cases

Abstract test-cases cannot be directly used because of the abstraction of time and domain knowledge. In order to have a test-case or a test suite which can be directly used we have to convert the abstract test-case to a concrete one. For this purpose we have to define a function which maps qualitative values to their corresponding quantitative values. This function has to be adapted for specific qualitative models of systems in order to capture the relevant aspects of a system. In particular, the function has to distinguish cases where the state sequence which represents an abstract test-case has to be mapped to a sequence of quantitative variable values from cases where information about dense time is required. In our example, the states carry information about the time during the day.

The function which maps abstract test-cases to their concrete counterparts is called a refinement function because it has to refine the abstract knowledge in order to lead to a specific and executable test-case. In the following, we discuss requirements of refinement functions:

- The refinement function has to preserve the order of events. Consider for example two immediately succeeding states at the qualitative level where we have events x and x' respectively. If we map x to y and x' to y' at the quantitative level, then the occurrence of y has to be before y'. For events occurring at the same state of the qualitative level no ordering can be ensured at the quantitative level.
- The mapping from quantitative values to qualitative ones has to preserve the order relation.
- The refinement function should ignore those events which are neither used to stimulate the system under test nor to check whether the system behaves correctly or not. Hence, events which are only relevant to compute a certain environment behavior but cannot be used to test the system should be ignored.

Given such a refinement function f_R we can compute a concrete test-case from an abstract one. However, even if f_R fulfills all requirements, the test-case at the quantitative level needs not to be a valid test-case in terms of being executable. This problem is similar to the one in verification. In (Ball & Rajamani 2002) a program is compiled into a representation using predicate abstraction. If a counter-example can be obtained from the abstract version of the program, the real program might be correct. Such problems always occur when using abstraction and correspond to the used abstraction mechanism which might ignore knowledge which is necessary to avoid the problem.

The methodology for generating test-cases from qualitative models has to have the following steps:

1. Compute abstract test-cases for the given qualitative model, the input constraints and the test purpose.
2. Apply the refinement function f_R to all abstract test-cases to obtain a set of concrete test-cases.
3. Evaluate the set of concrete test-cases by (i) executing the system under test on those tests and (ii) manually checking the test-cases for plausibility. Test-cases which are not plausible or which cannot be executed because of other reasons can be removed from the test suite.

An advantage of this methodology is that we obtain a lot of tests which based on a firm ground and all of them represent a certain interaction between the system and its environment. Hence, it is very unlikely to miss a test-case because of missing requirements.

The concrete test-case for our example would be of the form:

Simulate a night for a duration of 10 hours. The system is not allowed to run out of power during the simulation.

The specific value for the duration depends on the expected area of operation of the system. Hence, such values have to be specified when defining the refinement function f_R. The assertion that the system is not allowed to run out of power would come from a system engineer and is equivalent to the negation of the qualitative test purpose, i.e., `battery_charge` reaches the value `empty`.

The question of how to execute the obtained test-cases is a different one and is not in the focus of this paper. Moreover, it is not always possible to automate test-case execution. In our example we have to simulate the conditions of a night which can be hardly automated.

Related research

Classical white-box or black-box testing and test-case generation techniques (Beizer 1990) assumes both the existence as well as the accessibility of a source code, or a specification from which test-cases can be extracted. In most cases the objective is to prove the correct implementation of functions. Since specifications are hardly ever complete there might be cases where interactions of systems with their environment lead to harmful situations. This becomes even more critical if the complexity of systems is increasing. One solution to this problem is to provide models of the environment and its interaction with the system to be developed and extract possible interaction sequences for testing. For example, (Auguston, Michael, & Shing 2005) follow this solution.

In classical white-box testing and test-case generation the objective is to generate test-cases which fulfill a certain program coverage criterion like statement or path coverage. Such critera are very usefull for verification purposes because they allow to judge the quality of the used test suite to some extend. However, in black-box testing where a specification is available someone is more interested in test-cases covering the specification. This is the case for our approach where the test-cases are extracted from QR models using possible simulation runs. The quality of the computed test-cases wrt. to program coverage is left for future research.

(Auguston, Michael, & Shing 2005) introduced the use of attributed event grammars for generating test-cases from environment models for reactive systems. In the paper the authors use the grammar for representing an event-based model. Possible execution traces of the model form the test-cases. Insofar the underlying idea for test-case generation as described in this paper is very similar and can also be found in other papers, e.g., in (Fraser & Wotawa 2006a; 2006b). However, the mentioned papers can be distinguished with respect to the underlying modeling language. Whereas Auguston et al. are using attributed event grammars, Fraser et al. are using temporal logic and model-checking techniques, and in this paper we are proposing the use of qualitative models for test-case generation.

(Esser & Struss 2007) focused on test-case generation from finite state machines and on the underlying theory. In their work the authors want to create distinct test-cases, i.e., test-cases that allow for distinguishing different faults. For that purpose, the finite state machine model is compiled into a constraint representation. In contrast to Esser and Struss the methodology proposed in this paper does not require a transformation of the model. Moreover, we are more interested in obtaining test-cases from environment models (like (Auguston, Michael, & Shing 2005)) and not from behavior models of the system.

Conclusion

In this paper we followed the basic idea of (Auguston, Michael, & Shing 2005), i.e., using information about the environment of a system in order to generate test-cases for validating the system. The focus on the behavior of the system's environment is important because this potentially leads to test-cases which would not be generated when considering only the system's functional requirements. In contrast to (Auguston, Michael, & Shing 2005) we introduced the use of qualitative models for describing the behavior of the environment. We argued that qualitative models provide the right means for describing the behavior of the environment in terms of physical laws and causal relationships and the important parts of the system under test. The models themselves can be simulated and thus reveal possible interactions of systems with their environment. Hence, simulation results induce potential test-cases. Because the obtained test-cases present only an abstraction of the real behavior they have to be refined.

Advantages of the proposed methodology are:

- Test-case generation relies on well defined modeling paradigms and simulation engines are available.
- Because QR methods provide all possible behaviors (something which is a drawback in certain cases) even rare interactions of the system with its environment can be found.
- The restricted value domains in QR allows for taking into account all possible inputs. Those input values have to be refined, i.e., the abstract values have to be mapped to their corresponding quantitative values when computing the real test-cases for a system. This refinement step has to be specified by the user of the test-case generator. This is not a drawback but ensures flexibility. Moreover, the same happens when using abstraction which is sometimes necessary for formal verification and test-case generation using classical methods.
- QR is very well adapted for representing environment models and thus makes it very attractive for generating test-cases for reactive systems which interact with the environment.

Future research has to provide case studies for generating test-cases from QR models.

References

Auguston, M.; Michael, J. B.; and Shing, M.-T. 2005. Environment behavior models for scenario generation and testing automation. In *International Workshop on Advances in Model Based Testing (A-MOST 2005)*. St. Louis, Missouri, USA: ACM.

Ball, T., and Rajamani, S. K. 2002. The slam project: Debugging system software via static analysis. In *Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*, 1–3.

Beizer, B. 1990. *Software Testing Techniques*. Van Nostrand Reinhold.

Bredeweg, B.; Bouwer, A.; Jellema, J.; Bertels, D.; Linnebank, F.; and Liem, J. 2006. Garp3 – a new workbench for qualitative reasoning and modelling. In *Proceedings of the 20th International Workshop on Qualitative Reasoning (QR-06)*.

Esser, M., and Struss, P. 2007. Fault-model-based test generation for embedded software. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, 342–347.

Forbus, K. D. 1984. Qualitative process theory. *Artificial Intelligence* 24:85–168.

Fraser, G., and Wotawa, F. 2006a. Property relevant software testing with model-checkers. In *Proceedings of the Second International Workshop on Advances in Model-based Testing (A-MOST '06)*.

Fraser, G., and Wotawa, F. 2006b. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*.

Kuipers, B. 1986. Qualitative simulation. *Artificial Intelligence* 29:289–388.