2011 Fourth International Conference on Software Testing, Verification and Validation Workshops

Requirements-driven testing with behavior trees

Marc-Florian Wendland, Ina Schieferdecker and Alain Vouffo-Feudjio
Fraunhofer Institut FOKUS
Berlin, Germany
{marc-florian.wendland, ina.schieferdecker, alain.georges.vouffo.feudjio}@fokus.fraunhofer.de

***Abstract*—Requirements engineering is vital for a software development project's success or failure. As today's software systems are getting more and more complex, their related requirements specifications contain often hundreds, even thousands of natural language requirements. The so called *behavior engineering* developed by Geoff Dromey is suitable to handle the complexity of large-scale software requirements specification by relying on a scalable requirements formalization methodology. The outcome of that methodology is a requirements model in the behavior tree notation, describing the intended, externally visible behavior of the system. By deliberately extending the behavior engineering methodology with testing activities, those requirements models can be further exploited for testing purposes like system and acceptance level testing. This also addresses the common challenge in model-based testing scenarios, namely the availability of a meaningful test model. By reusing a testable requirements model, both the system and test model can be derived from the same specification, since all requirements are intended to be captured in the requirements model properly and consistently. In this paper, we present an approach of how behavior trees can be extended with testing activities to leverage the definition of test requirements. We also briefly discuss how augmenting test-related information to make the requirements model more complete in terms of the IEEE830 standard.**

Keywords-requirements-driven testing, model-based testing, validation and verification, behavior engineering, behavior trees, testability of requirements

I. INTRODUCTION

Commonly, testing of software systems targets two main objectives that are gaining confidence in that the system does what it is supposed to do and finding defects or anomalies in the system.

A crucial precondition for finding anomalies is that the system's requirements are captured in an appropriate manner to perform further analysis on them. Since testing is always a comparison of what a system shall do and what it actually does, the quality of a system's requirements is the decisive factor for the quality of the system itself. Knowing the quality of a system before it is released is essential to avoid maintenance costs resulting from failures in the production environment.

Unfortunately, requirements are not that easy to be defined unambiguously. Mostly, they are captured in natural language, leading to imprecision and inconsistencies due to natural language's inherent ambiguities. Though lots of efforts have been spent in the last decades in research and industry, requirements are still hard to specify. Standards like [1] support the creation of reliable and sustainable software requirements specifications (SRS) by defining classification criteria of what characteristics a requirement specification shall meet. An SRS is the foundation for the definition of system use cases [2]. A system use case defines the required functionality of a system (i.e. what it is supposed to do). Additionally, they relate the system to entities of its environment, expressing how the system is intended to be used by those external actors.

From a tester's view, use cases are the starting point for system and acceptance level (henceforth referred as system testing) testing, since they are a valuable input for capturing the intended user behavior [4]. Typically, test analysts extract test relevant information from the natural language SRS and the use case descriptions, which is often a difficult task. The degree of how easily information may be obtained and extracted from the SRS and use case descriptions influences the efficiency and effectiveness of the requirements-driven validation and verification process. Validation and verification aims at assessing to what extent

- the system behaves as it is intended to (all requirements are reflected in the system), and
- the system's intended behavior is correctly realized (each requirement is implemented properly).

Our work addresses parts of a requirements-driven system testing process, including validation and verification, by

- use of the behavior engineering methodology [5] for formalization and validation of requirements specification,
- extending the behavior engineering methodology with appropriate test activities,
- showing how testing information may be weaved into behavior trees, and by
- providing an outlook on how test case specifications may benefit from the systematic approach of behavior engineering.

II. CHALLENGES OF MODEL-BASED TESTING

According to [6], today's testing still faces some serious challenges. Often, the way in which test cases are derived from an SRS is captured is a tester's implicit knowledge. Worse, the quality of the resulting test cases and transitively

978-0-7695-4345-1/11 $26.00 © 2011 IEEE
DOI 10.1109/ICSTW.2011.25

IEEE computer society

the informative value of the quality estimation are bound to a tester's ingenuity. If the whole process is barely documented (another shortcoming) and the responsible tester leaves the test team, or even worse, the whole company, the whole process will no longer be reproducible. Reproducibility, in fact, is a crucial feature for reliable test processes.

In the opinion of most experts in the realm of testing ([4], [7]), model-based testing is a promising approach to mitigate these problems. Model-based testing refers to a new paradigm of testing, whereby test cases are automatically generated from (semi-) formal models. Those models are called *test models*. Roughly, a test model describes how a SUT shall be used or tested, whereas the *system model* contains all architectural design details [8].

One vital question is how to obtain those test models. MBT scenarios are located somewhere between two extremes ([6], [9]), either creating the test models completely new from scratch or trying to exploit already existing system models for testing. [10] names the first one as combined models approach (Figure 1) and the second one as independent models approach (Figure 2).

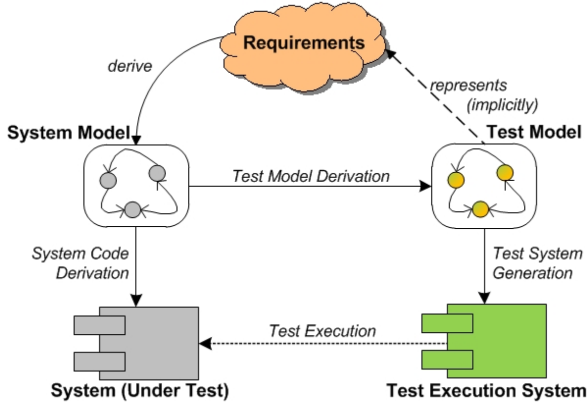

Figure 1. Combined models approach

Reminding that testing is always a comparison of intended and actual behavior of a SUT, both variants exhibit shortcomings. The combined models approach lacks an independent interpretation of the requirements by transitively taking all errors over into the test model that have been made in the system models when representing requirements. Such a process will not allow the requirements stated in the SRS to be validated. Missing or erroneously implemented requirements cannot be identified.

The independent models approach prevents this by not taking already existing models (or just those elements which are not relevant for the generation of test cases) into account. Instead, a completely new, independent model is created for testing purposes, thus, error propagation from system model into test model is avoided. The main challenge here is that the requirements must be modeled twice (see two derivation arrows in Figure 2, going from requirements to both a system and a test model). Since modeling is not trivial, this approach is resource-consuming and error-prone.

An additional challenge for all model-based testing approaches is the educational aspect. Testers are usually not that familiar with sophisticated modeling notations and languages, thus, if the modeling language is too unfamiliar to the tester it will meet with a refusal.

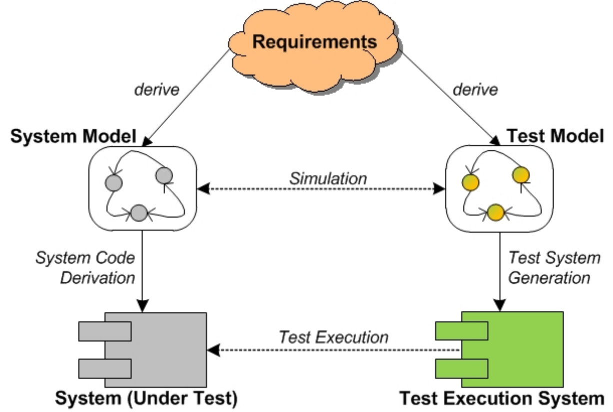

Figure 2. Independent models approach

To sum up, a key challenge of MBT scenarios is to ease the translation of requirements specifications into test models[1]. If there would be an intuitive and easy-to-use methodology for the formalization of requirements including checks for validity and correctness thereof, time and money could be saved by exploiting such a requirements model for test model creation. This idea is sketched in Figure 3.

III. FORMALIZING REQUIREMENTS WITH BEHAVIOR TREES

The behavior engineering methodology was firstly presented by [11]. Behavior engineering addresses classical problems in the requirements engineering domain by systematically capturing and validating the requirements in the very first stage of a system's development process. As large system requirements specifications may quickly become too complex to be handled and kept consistent by human beings, behavior engineering encounters this challenge with a scalable, repetitive methodology. Behavior engineering consists of two subparts, the Behavior Modeling Language (BML) and the Behavior Modeling Process (BMP). BML defines all elements and visual notations to describe the three different types of trees of the behavior engineering methodology: behavior tree (BT), composition tree (CT) and structured tree (ST). Additionally, due to its highly readable notation, BML assists domain and system experts to review the system's intended behavior. Another advantage is that it maintains the language of the stakeholder as it is documented in the natural language requirements, being very useful for walkthroughs and reviews with the stakeholders. BMP

[1] Undoubted, system model creation is challenging, too.

defines the behavior engineering's scalable methodology. It handles complexity of SRS by tailoring the global problem space into a set of local problem spaces, iteratively completing the global one [12].

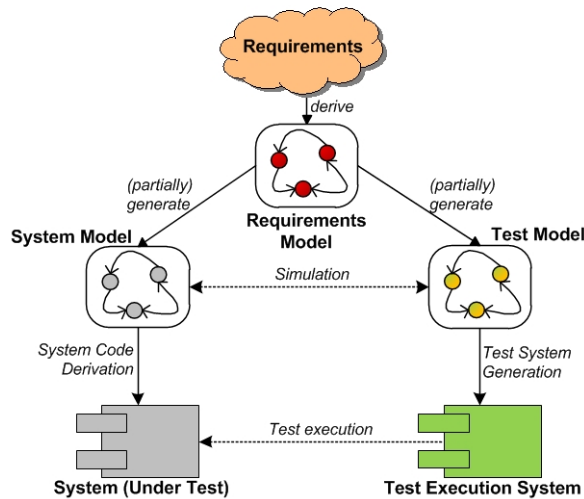

Figure 3. Exploiting formalized requirements for system design and testing

The examples on the generic infusion pump, being presented subsequently, are taken from the ROTESS [13] project, which deals with risk-oriented testing of embedded, safety critical systems. The software (and safety) requirements are taken from the *Generic Infusion Pump* (GIP) project [14], carried out by University of Pennsylvania and Food and Drug Administration (FDA). It dealt with the definition of generic safety requirements intended to be used for the specification of safety properties for different classes of infusion pumps. The following requirements definitions have been partially re-engineered from MathLab models of the GIP. Due to page restrictions and readability, only few trees are presented in the paper. The full material can be obtained from [13].

F-1.1: If the power button of the infusion pump is pressed, the pump will be switched on and initialized.

F-1.2: The infusion pump is programmed with a basal and bolus rate as well as application time by the user. The programmable basal rate shall be between 1 ml and 9999 ml.

F-1.3: When the user starts the infusion pump, it starts infusing drugs to the patient until the application timer runs out..

F-1.4: After the application timer expires, the pump program will be reset automatically. Afterwards, the pump is again programmable.

A. BML in a nutshell

The BML comprises three different kinds of trees describing the requirements models. The composition tree defines components and compositional relationships among those components. Commonly, there is one composition tree for the entire system. The behavior trees specify the behavior being obtained from the SRS and intended to be executed by components. Each behavior is associated to a component. Finally, the BML provides the structured trees which are used to describe the structure of the system and to define constraints that structure must respect. Due to page restriction, we will exclusively focus the BML for behavior trees in this article.

The BML defines a graphical concrete syntax which is reused for all of its integrated views. The notation consists of two basic graphical constructs: rectangles and edges, connecting those rectangles. The rectangles differ into behavior and component nodes, depending on the kind of tree they are used in. Within behavior trees, the rectangles represent behavioral nodes, i.e. behavior, supposed to be executed on an associated component. The general elements of a behavior node are shown in Figure 4, and explained subsequently:

A. Component name: Specifies the component on which the behavior is executed
B. Operator: Additional operations for threading behaviors (not further explained in this article)
C. Label: An additional label for disambiguation (not further explained in this article)
D. Statement: States the type of behavior more precisely
E. Behavior Type: Delimeters indicating the type of behavior
F. Traceability Link: A reference to a requirement outside of the tree
G. Tag: The box on the left, used for traceability information
H. Traceability status: Indicates how the node relates to the traceability link

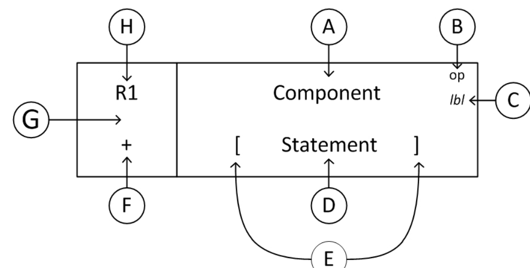

Figure 4. Syntax of behavior nodes

Figure 5 depicts the type of behaviors used for capturing natural language requirements specifications. Their semantics are briefly introduced subsequently:

a) State realisation: Component C realizes state S when entering the node
b) Selection: Control flow is passed either to component X or component Y, depending on the evaluation of the condition between the question marks.
c) Event: Represent a high level event. Control flow is passed only if the preceding behavior node is active and the event e occurs.

d) Guard: A continually re-evaluated condition; passes control if the condition becomes true.
e) Input: Indicates the reception of either a message from the environmental component (external input) or a component within the system boundaries (internal input)
f) Output: Indicates the sending of a message to either an environmental component (external output), or a component within the system boundaries (internal component).
g) Assertion: Indicates a certain condition must held when control flow is passed to the node.

Nodes are connected with each other by connectors. The BML for behavior trees provides two kinds of connectors, that are atomic and sequence connectors. Sequence connectors end with an arrow at the lower end, whereas atomic connectors are just a simple line.
Behavioral nodes, connected by a sequence connector, may be interleaved with the execution of other behavior nodes in a separate thread. In opposite, the semantics of the atomic connector prohibits interleaving behavior between the execution of the source and target behavior. They are executed in one indivisible step.
Most of the examples given in this article use the sequence connector. An example of an atomic connector is shown in Figure 9 between the selection and the state realization for state *blocked*.

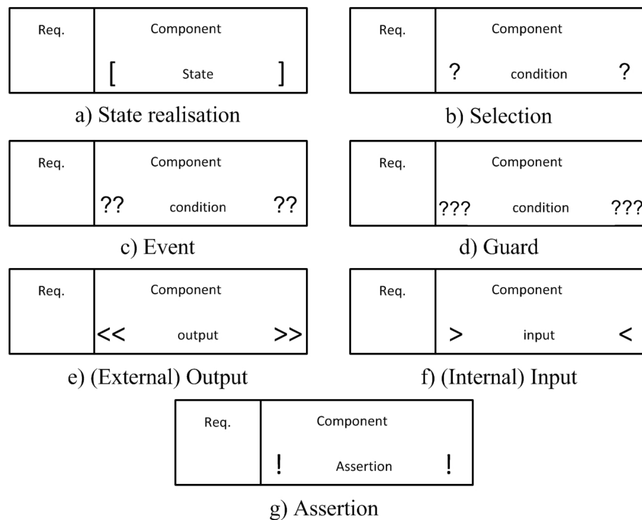

Figure 5. Behavior elements of the BML

B. Requirements formalization

The very first step in BMP is the formalization of each informally expressed (mostly in natural language) requirement into a r*equirements behavior tree* (RBT). The RBT for the requirement F-1.3 is depicted in Figure 6.
A RBT captures the behavior of a single requirement, as well as the components, on which this behavior is intended to be executed. The content of the original informal requirement is supposed to be translated almost one-to-one in the RBT. It is the basic step for any other activity in behavior engineering and is succeeded by the integration phase.

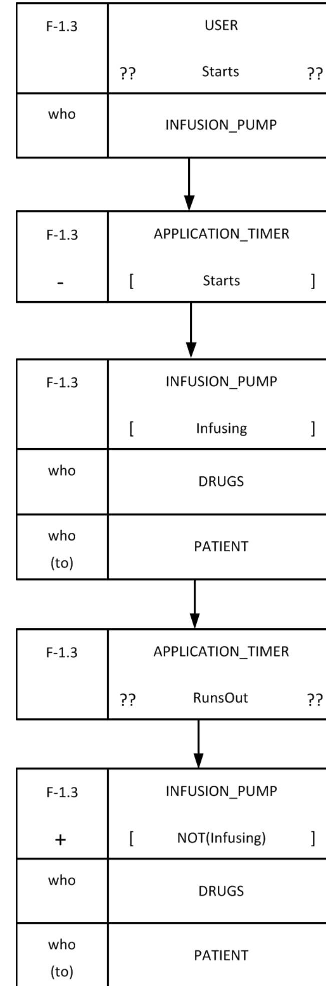

Figure 6. RBT for requirement F-1.3

C. Fitness-for-Purpose (integration)

The integration of single RBTs targets the creation of an i*ntegrated behavior tree* (IBT), giving a holistic view of the system's intended behavior. For that, two steps are to be carried out subsequently.
The first one aims at integrating the RBTs gradually until they form the IBT. The identification of integration points of a single RBT is based on two foundational axioms, the precondition and interaction axiom [11]. The precondition axiom states that each "constructive, implementable, individual functional requirement of a system, […], has associated with it a precondition that needs to be satisfied […]". The interaction axiom extends the first one by saying that such a precondition "[…] must be established by the behavior tree of at least one other functional requirement that belongs to the set of functional requirements of the system." In short, each RBT interacts somehow with at least one other requirement, either by using it as precondition or by defining the precondition for it. In fact, there is exactly

one RBT that does not have a precondition to satisfy, namely the root node of the IBT[2]. An insight into the IBT for of the GIP is given in Figure 8, showing the integrated requirements F-1.2 and F-1.3

If an RBT cannot be integrated with any other (maybe already integrated) RBT, it is an indicator of missing or misunderstood behavior within the set of RBTs. In that case, a second step is required to clarify and correct either the requirement, defining the precondition, or the requirement, trying to integrate into the precondition. Changes have to be affected in a way, allowing the dangling RBT being integrated into the IBT.

D. *Specification*

The specification phase serves for the definition of executable requirements specifications. A behavior tree resulting from the specification phase is called modeling behavior tree (MBT). Although this phase is also interesting for testing purposes, it will not be discussed in greater detail in this paper due to page restrictions. Please refer to [15] and [12] for further details.

IV. Preparing Behavior Trees for Testing

The IBT presents an overview of the system's intended behavior, originated from its informal requirements. Often, those requirements merely specify the normal situation[3], neglecting exceptional situations. Those are necessary to cover in order to create reliable and stable software systems. One could argue, if something is not specified to be explicitly forbidden, it is by default allowed. However, this is not a satisfying way to deal with insufficiently specified software requirements. From a tester's point of view, the determination of how the system shall react in exceptional situations is one of the key activities a test analyst has to perform at a very early stage of a test process. Therefore, this information must be obtainable from the SRS. Among others, IEEE 830 standard requires a requirement, captured in a SRS, to be *complete* and *verifiable*. Those two features can be summarized as follows:

Completeness. The SRS is considered to be complete, iff, the definition of the response to all realizable classes of input data in all realizable classes of situation for valid and invalid values are given.

Verifiability. A software or a software requirement is verifiable (or testable), iff a finite cost-effective process exists with which a person or machine can check that the software meets the requirement.

In conjunction, those two features are most relevant for testing. Projected to behavior engineering, we propose a new phase *test augmentation*, extending the already existing methodology. This makes the final requirements model containing all required information for both, system and test specification, as depicted in Figure 3.

A. *Test augmentation*

Test augmentation comprises two related subtasks as depicted in Figure 7, which are *test requirement elicitation* and *test requirement formalization*. Before going into detail, we discuss how to integrate test augmentation into the behavior engineering methodology.

The popular V- and W-process models propose the test activities taking place in parallel to system development and as early as possible, right after the requirements specification has been finalized. Sticking to this suggestion, this would mean to weave test augmentation into the formalization phase of RBTs. Though this represents the earliest point in time in behavior engineering, it seems to be inadequate as the content of a RBT is not consolidated yet. As already discussed, the formalization phase aims at translating the textual description into an RBT and at finding implied and missing behaviors related to a requirement. Even if all ambiguities in an RBT are resolved, it is still possible, that the RBT will be adapted in the fitness-for-purpose phase. Additionally, single requirements are hardly isolated, but are interrelated to other requirements in use cases. Having taken this into account, starting from a consistent and stable IBT for the test augmentation phase makes most sense: most of the ambiguities among RBTs (ideally all) are resolved and use cases can be identified as a basis for testing.

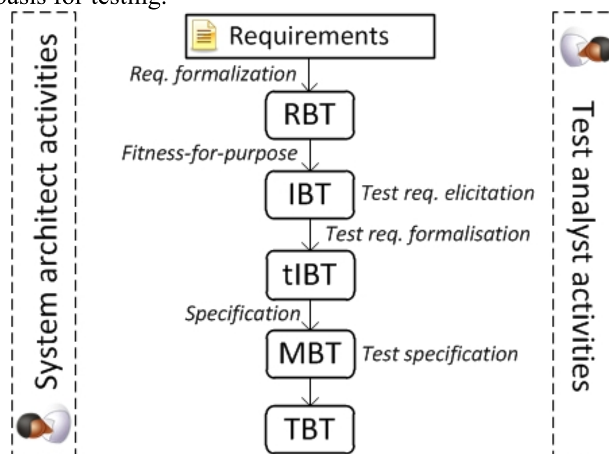

Figure 7. Behavior engineering methodology with testing activities

Use cases comprise sets of requirements they realize, thus, all information needed to identify. We propose a particular way of how to gradually integrate RBTs and IBTs with each

[2] If there are reversions within the requirement to the root node, than those reversion nodes represent the precondition for the root node.

[3] Other terms are happy path or positive way.

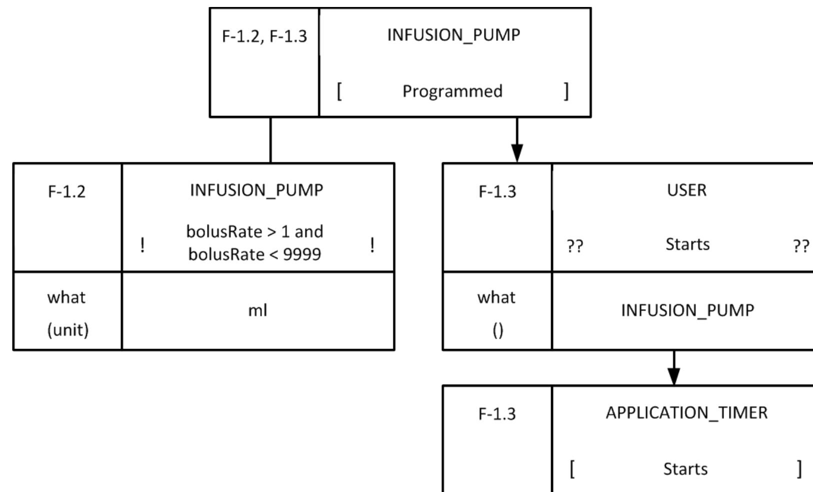

Figure 8. Partial IBT covering requirements F-1.3 and F-1.2

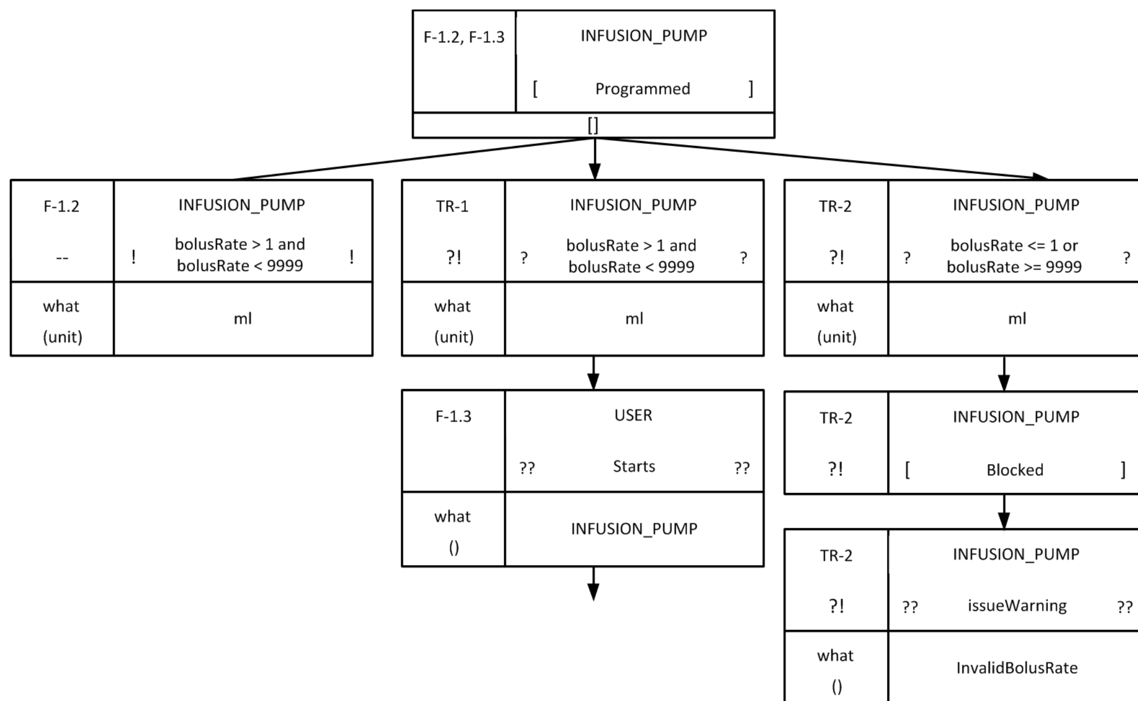

Figure 9. Partial tIBT covering TR-2

other, to facilitate the identification use cases during this stage. The term *use case* does not necessarily refer to UML Use Case diagrams, but rather to the logical concept beyond. Due to its scalable and repeatable methodology, RBTs may firstly be combined to partial IBTs. Each partial IBT represents a particular use case. Even if the usage of partial IBTs for a stepwise integration is not new, to use it for testing purposes by explicitly gathering requirements realizing certain use cases under consideration was not proposed yet.

After the use cases are consolidated, a test analyst can review the tree regarding testing purposes and addressing the IEEE 830 feature *completeness.* The question to be answered is "Which aspects of this use case need to be tested?" The answer is related to quality criteria of software systems as standardized by ISO 9126 standard [16]. Use cases can be examined from different quality criteria, meaning that different quality criteria result in different test related information, required to be attached to the IBT.

The most intuitive quality criterion is functionality, specifying to what extent a system fulfills its intended functionality. Since an IBT contains the required functional behavior of a system, for each scenario of a use case under consideration, at least one test requirement should be defined. If we apply this to the scenario covering the requirements F-1.2 and F-1.3, a corresponding test

requirement can be understood as a mirrored scenario. Let us consider an example:

TR-1: Ensure that if a valid bolus range was programmed, the infusion pump starts infusing the drugs to the patient.

In terms of robustness testing (being a sub-criterion of the reliability quality characteristics), this use case lacks some important information, that is, what should happen if the programmed basal rate does not match the valid range (if the assertion in Figure 8 is violated). This leads to a second, important test requirement.

TR-2: If the programmed bolus rate is not valid, the infusion pump shall be blocked and a warning shall be issued, indicating the invalid bolus rate.

Of course, this information cannot be determined by the test analyst alone, rather the system architect must be included in the *test requirement elicitation*. By doing so, the IBT will be enriched with test information by formalizing test requirements as explicit parts of the IBT. In our simple example, the way of deriving the tIBT out of the IBT includes the resolution of the assertion node, introduced by the original, natural language requirement F.1-2. The assertion implies only ranges between 1ml and 9999ml are allowed for the bolus rate. This is an important information for testing, since it must be clarified how the system shall react, if the assertion expression is violated. Therefore, we suggest resolving the assertion into two selection nodes; one for the intended bolus rate and one for the exceptional situation (see Figure 9). The selection nodes refine the assertion node, which will be replaced and marked as deleted from the system (see the -- traceability status, indicating this node is no longer part of the system behavior).

Hence, *test requirement formalization* leads to an extended system behavior, which we call the *testable integrated behavior tree* (tIBT). Inserted behavior nodes are marked as introduced for test augmentation. We propose an additional traceability status to be included into the behavior engineering methodology for testing support[4].

By forming the tIBT, the SRS is completed with respect to the IEEE 830 standard. It still contains events that need to be refined for latter test specification, but it reveals all important behavioral descriptions to measure the reliability and quality of the targeted system.

B. *Test specification*

To clarify, test augmentation does not claim to produce an already implementable test specification. The tIBT, however, contains all information needed to be refined into a specification of test cases. To achieve this, we define an additional phase: the *test specification* phase. During test specification, the information added by test augmentation is exploited to derive a test model, which we call *testing behavior tree* (TBT).

As Figure 7 depicts, building the IBT is not the final stage in behavior engineering. We already mentioned the specification phase, with its most important step, the resolution of remaining high level events and the consolidation of relational behavior.

During test specification, the MBT, resulting from changing the tIBT into an executable one, must be refined in order to be usable for further test implementation and execution. As mentioned, the MBT is already part of the behavior engineering methodology. We claim the testing methodology for behavior engineering to be minimal invasive by just partially extending the methodology where needed. Therefore, it would not bring any benefit to introduce a new term *testable MBT* here. The specification phase, leading to the MBT, can be always performed whether the input is an IBT or a tIBT whereas both inputs results in an executable specification. In the latter case the MBT is expressive enough for further test specification, but it still does not contain any dedicated testing activities solely. If the foundational SRS would contain all needed information for testing, IBT and tIBT are identic.

First step is to identify the SUT boundaries. Fortunately, for system testing, this problem is easy to solve. Every component, belonging to the software system, is part of the SUT. Every external component represents test components, stimulating the SUT and observing its reaction.

The second task is to include additional behavior, indicating test case specific actions, which are not present in MBT yet. A tIBT does not contain test specific actions, but enough information to define test actions subsequently. We differentiate stimulation actions, sending test data (or stimuli) to the SUT and a validation action, checking the outcome or state of the SUT for adequacy regarding its specification.

The stimulation actions are already determined along the identification of the SUTs boundary. Any message going from environmental components to components, marked as SUT, is a test action. In our example, sending anything to the infusion pump will be considered as stimulation action.

Since validation actions have to be stated explicitly in order to determine the test case being passed or failed, we suggest reusing the assertion behavior nodes for this issue. Figure 10 depicts the partial TBT for TR-2. Adding assertion nodes to locations of the MBT, new paths within the TBT are created, each considered representing a test case. By weaving that information into behavior tree, we also the address the feature *verifiability* of IEEE 830, saying that as long as we can somehow express validation checks as assertions and can observe them at the SUT, the requirements, for which this test case is being carried out, is verifiable.

---

[4] Traceability statuses describe how this particular behavior was identified and created within the behavior tree. It has nothing common with traceability of artifacts during a model transformation. The traceability status *?!* is reused from input output labeled transition systems, where *?* stands for an input into an object and *!* for the output from that particular object.

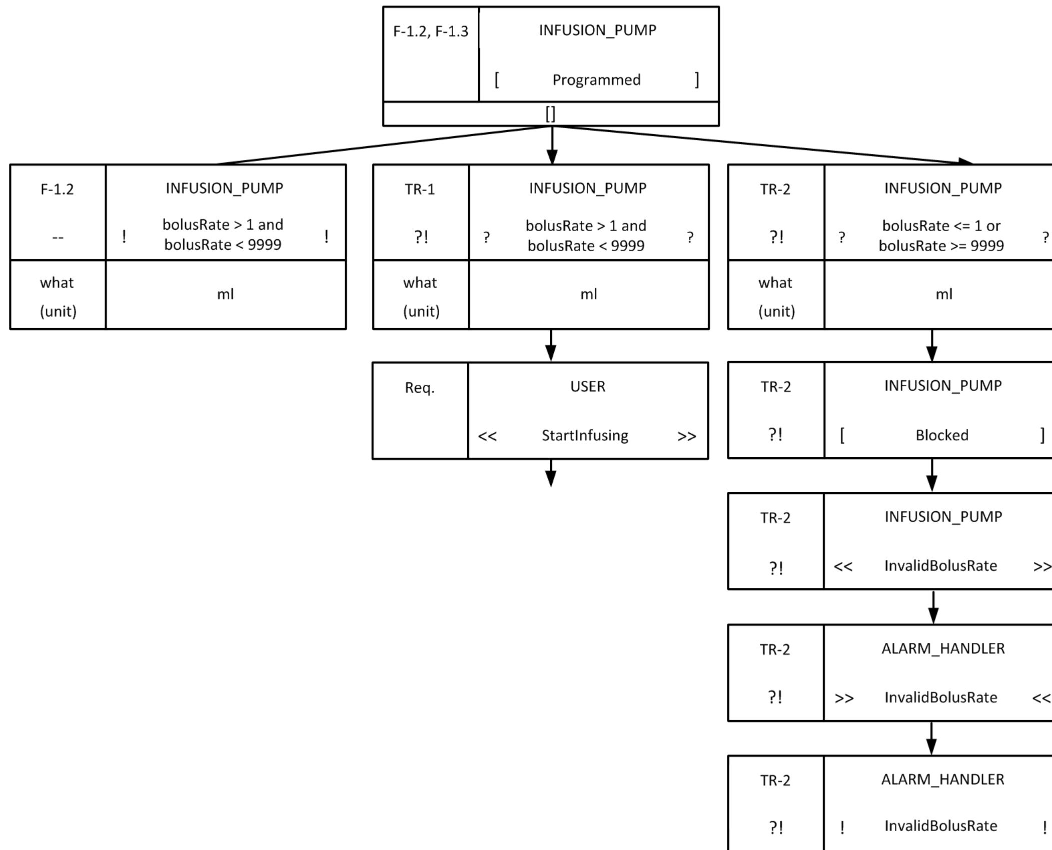

Figure 10. Partial TBT covering test actions for requirement TR-2

C. *Test case realization*

Once a TBT is specified, it can be exploited in several ways. A tester can easily read, understand and implement the TBT with an appropriate test implementation language (e.g. TTCN-3 [17]). Even if the testing would be done manually, the test process still benefits as tests are obtained along a structured and well-defined methodology.

Another idea is to perform further transformations on the TBT to go from the behavior tree notation into a more commonly used modeling notation for testing like the UML testing profile (UTP) [3]. Being a native UML profile, UTP inherits all concepts of UML. By transforming the TBT into UTP with a state machine describing the SUT's intended *and complete* abstract behavior, it is possible to perform automated test case generation on it, by relying on both structural and data coverage criteria. Since UTP can be mapped to TTCN-3 natively, UTP test cases can be seen as executable, too. Research is currently undertaken to specify a transformation between behavior trees and UML state machines, although it is not finished yet.

D. *Benefits of test augmentation in behavior trees*

An obvious benefit of test augmentation is the early detection of missing (unspecified) behavior. The integration phase merely cares for missing or ambiguous behavioral parts of requirements preventing their RBTs from being integrated with each other. In case the SRS lacks a complete requirement, which does not negatively affect the integration, such a behavioral leak might not be identified. Such a not-affection appears if a requirement is completely isolated from other requirements. More precisely, if that requirement would result in a leaf node within the behavior tree on which no other requirement is depending. Because of the early involvement of a test analyst, being an expert in finding such leaks, the whole behavior engineering methodology may take advantages from it. In fact, checking for completeness was already intended by [15] for the specification phase, the earlier such detection takes place the better for the entire succeeding process phases. In case that creating an executable specification is not desired and behavior engineering is stopped after the consolidation of IBT, the missing behavior will be detected during test augmentation phase anyhow. If the specification phase is carried out nevertheless, most of the missing behavior would hopefully be found during test augmentation, so that the specification phase can focus on its other subtasks.

E. *Implied activities in this example*

As mentioned earlier, there is a rigorous methodology defined to gradually come from high-level RBTs to the executable MBT. This includes a number of refinement steps to be performed. What we have not shown in this

example is the way how to get from the (commonly) non-executable tIBT to the executable MBT, which is necessary to derive the TBT subsequently. The specification phase was implicitly carried out between Figure 9, showing a part of the tIBT, and Figure 10, showing the same part of the TBT. Since the focus of this article is not to explain or discuss the specification phase of the behavior engineering methodology, we neglected it due to page restrictions. However, in order to comprehend the refinement between Figure 9 and Figure 10, we give a short introduction to the refinement steps composing the specification phase following.

The most important step is to resolve the generic event nodes (??) into a specific event. Event nodes are very often used during formalization phase, since it is often not possible to determine what concrete event will take place. The integration phase, which is still not executable, potentially introduces concurrency to the system behavior. It is than possible to identify to what kind of concrete event (guard, input, output) the generic event node must be refined to. A guard node indicates the existence of a parallel behavior, fulfilling at some point in time the condition of the guard. Input and output nodes indicate the existence of the particular counterparts somewhere else in the system. This information is commonly absent within a single RBT. In Figure 9 the user is supposed to start the infusion pump somehow. After specification phase (Figure 10), the way how the user starts the pump is stated more concrete by saying that the user will send a message to the infusion pump (the reception of the message is not relevant for this overview and hidden from the figures).

Another important action is to resolve relational behavior. Due to page restriction, we have neglected relations in this paper. However, in the tIBT there is an event node defined for the infusion pump, stating a warning will be issued. The rectangle underneath this node is a relation, clarifying what kind of warning will be issued (*InvalidBolusRate*). To create an executable specification this must be resolved into a concrete message (in this example). This is depicted in Figure 10 where the infusion pump sends a concrete message of type *InvalidBolusRate* to the environment, expressed as an external output node. The reception of that message is given with the following external input node, defined for the component *AlarmHandler*.

There are of course other relevant steps needed for complete the specification phase, which we did not mentioned here. Please refer to the literature regarding specification phase we mentioned earlier in this article (see section III.D).

V. RELATED WORK

To the knowledge of the authors, adding test information and activities into behavior trees have not been addressed yet. The methodology we propose fits very well into the ISTQB fundamental test process, as explained in[22]: Analysis refers to test augmentation (IBT to tIBT), design refers to test specification (MBT to TBT), realization can be done either in TBT or by going from TBT to UTP and execution is a question of what test execution language will be chosen. The other phases are rather analytical test management steps which are not covered in the methodology yet.

However, behavior engineering used for system design has been discussed in several research and industrial projects, all listed at [5]. Industry use cases has proven the behavior engineering methodology to be extremely beneficial for extracting system-behavior models directly from natural language requirements ([23], [24], [25]). A similar case study like GIP [14] was presented by [18]. The main difference is that the authors did not focus testing activities as we do and they re-engineered all security and safety requirements from the pump's user manual.

A comparison of behavior engineering to other popular system design techniques is given in [26]. It reuses the case study, provided by the Design-Methods Comparison Project, extend it with the behavior engineering methodology. It also includes a survey of how behavior engineering and UML, respectively SysML, differ from each other.

[1] provide a standardized set of characteristics a SRS shall met. It also defines structural elements for an SRS document should contain, going from of the intended use of the system through its environmental interfaces to single requirements descriptions. Additionally, templates of concrete SRS structures are provided in the annexes. As often with standards, it has to be tailored to meet specific demands.

The Object Management Group (OMG) adopted a new specification for requirements specification, the requirements interchange format (ReqIF) [19]. As the name suggests, ReqIF is supposed to merely represent a formal data model for requirement specifications with no methodology defined, on how to come to those requirements.

The sequence-based specification methodology ([20], [21]) represents the most similar approach to behavior engineering with testing. It deals with the formalization of safety-related requirements into Mealy machines. Those machines contain the normal situation states plus added erroneous states, obtained from the results of a Fault Tree Analysis (FTA). Structural coverage algorithms are performed on those machines. Although the idea of partially creating test models for hazardous situations is similar to our methodology, this approach does not deliberately focus the integration questions of requirements. Ambiguities, propagated in the normal situation machines are not systematically identified and removed.

VI. CONCLUSION AND FURTHER WORK

In this paper, we briefly presented how testing information and activities can be integrated into the behavior engineering methodology. We proposed to add new phases, the test augmentation and test specification to the initial methodology. The first one results in a new kind of behavior

tree, we called *testable Integrated Behavior Tree* (tIBT). A tIBT contains all information, considered to be necessary for system tests. We also suggested a new sort of traceability status for inserted behavior related to testing purposes

The second phase dealt with the creation of test specifications. We added a second new tree, the testing behavior tree (TBT). It results from identifying the system's boundaries and the definition of test actions. A TBT can be exploited both manually and automatically to implement and execute the test cases it specifies.

The most promising advantage of the approach is the elicitation of test requirements by reviewing the IBT from a tester's viewpoint. By doing so, the step to test specification gets simpler, because the information relevant for specifying test cases is present and does not need to be gradually determined somehow. The presented test augmentation phase leads to a systematic formalization of test requirements, which must be respected for designing and testing a system.

Further work will address in particular the transformation from behavior trees to state machines to benefit from already existing tooling for the generation of executable test cases from state machines. Another vital point is to analyze in greater detail how concrete test cases can be specified with TBT. It might also be the case that one MBT results in several TBT, each defining a distinct constellation of SUT boundaries. This comes along with thoughts whether behavior engineering can be used for integration or subsystem testing.

Finally, a broader case study will be performed in ROTESS and tool support must be prototyped to proof the methodology being applicable in real-world scenarios.

ACKNOWLEDGMENT

This work was partially supported by the projects ROTESS [13] and BTTest.

REFERENCES

[1] IEEE Standards Association (IEEE): 830-1998 – IEEE Recommended Practice for Software Requirements Specifications. *http://standards.ieee.org/findstds/standard/830-1998.html*

[2] Object Management Group (OMG). Unified Modelling Language (UML) Specification. Version 2.3, February 2009. Available at: *http://www.omg.org/spec/UML/2.3/*

[3] Object Management Group (OMG). UML Testing Profile, Final Adopted Specification. Version 1.0, July 2005. Available at: *http://www.omg.org/spec/UTP/1.0/*

[4] Baker, P., Dai, Z.R., Grabowski, J., Haugen, Ø., Schieferdecker, I. Williams, C.: Model-driven testing – using the UML testing profile. Springer (2007)

[5] Behavior Engineering (BE). *http://www.behaviorengineering.org*

[6] Utting, M.; Pretschner, A., Legeard, B.: A Taxonomy of Model-Based Testing. ISSN 1170-487X, 2006. *http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf*.

[7] Utting, U., Legeard, B.: Practical Model-Based Testing – A Tools Approach. Morgan Kaufmann Publ. (2007)

[8] Stefanescu, A.; Wendland, M.-F.; Wieczorek, S.: Using the UML testing profile for enterprise service choreographies. In: Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'10), pp. 12-19. IEEE Computer Society, 2010.

[9] Alexander Pretschner, Jan Philipps: Methodological Issues in Model-Based Testing. In: Model-Based Testing of Reactive Systems. Springer, 281-291, (2004).

[10] Schieferdecker, Ina: Modellbasiertes Testen. ObjektSpektrum 3/07, S39-45, 2007.

[11] Dromey, R. G. (2003), `From Requirements to Design: Formalising the Key Steps', in IEEE International Conference on Software Engineering and Formal Methods (SEFM'03), pp. 2-11, Brisbane, Australia, (Invited Keynote Address).

[12] Myers, Toby: The Foundations for a Scaleable Methodology for Systems Design , PhD Thesis, School of Computer and Information Technology, Griffith University, Australia, 2010.

[13] ROTESS project: Risk-oriented testing of embedded, safety-critical systems. *http://www.fokus.fraunhofer.de/de/motion/projekte/laufende_projekte/ROTESS/index.html*

[14] The Generic Infusion Pump (GIP) project: *http://rtg.cis.upenn.edu/gip.php3*

[15] Zafar, S.: Integration of Access Control Requirements into System Specications, Ph.D. thesis, Grith University, (2009).

[16] Internation al Organization of Standardization (ISO): ISO/IEC 9126:2001, *http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749*.

[17] European Telecommunications Standards Institute (ETSI): The Testing and Test Control Notation version 3 (TTCN-3). URL: *http://www.ttcn-3.org*.

[18] Zafar, S. and Dromey, R. G., (2005), Integrating Safety and Security Requirements into Design of an Embedded System. Asia-Pacific Software Engineering Conference 2005, 15th-17th December, Taipei, Taiwan. IEEE Computer Society Press

[19] Object Management Group (OMG). Requirements Interchange Format (ReqIF) 1.0 – Beta2, 2010. *http://www.omg.org/spec/ReqIF/*

[20] Prowell, S.; Poore, J.: Foundations of Sequence-Based Software Specification. In: IEEE Transactions on Software Engineering. IEEE Computer Society, Volume 29, No.5, May 2003.

[21] Bauer, Thomas; Böhr, Frank; Landmann, Dennis; Beletski, Taras; Eschbach, Robert; Poore, Jesse H.: "From Requirements to Statistical Testing of Embedded Systems", In: Fourth International Workshop on Software Engineering for Automotive Systems, SEAS 2007 - Proceedings. Los Alamitos : IEEE Computer Society, 2007.

[22] Spillner, A; Linz, T: Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester-Foundation Level nach ISTQB-Standard, Dpunk Verlag, Heidelberg, 2010.

[23] Powell, D.: Behavior Engineering - A Scalable Modeling and Analysis Method. In: Proc. of 8th IEEE International Conference on Software Engineering and Formal Method (SEFM 2010), pp.31-40, 2010, IEEE Computer Society, 2010.

[24] Bosten, J.: Behavior trees – how they improve engineering behaviour. In: 6th Annual Software & Systems Engineering Process Group Conference (SEPG), Melbourne, Australia, August 2008.

[25] Powell, D.: Requirements evaluation using Behavior Trees – findings from industry. In: Industry track of Australian Engineering Conference (ASWEC) 2007.

[26] Lindsay, P.: Behavior Trees: from Systems Engineering to Software Engineering. In: Proc. of 8th IEEE International Conference on Software Engineering and Formal Method (SEFM 2010), pp.31-40, 2010, IEEE Computer Society, 2010.