A Formal Method to Identify Deficiencies of Functional Requirements for Product Lines of Embedded Systems

Florian Markert
Computer Systems Group
Technische Universität Darmstadt
markert@rs.tu-darmstadt.de

Sebastian Oster
Real-Time Systems Lab
Technische Universität Darmstadt
oster@es.tu-darmstadt.de

Abstract: Functional requirements that were stated in cooperation with the stakeholders have to be analyzed and reviewed. Deficiencies like incompleteness, contradictions and redundancy within the requirements may lead to an extended development effort. Identifying and resolving these deficiencies in an existing or evolving set of functional requirements for embedded systems is of major importance. Especially, if the requirements describe a set of possible products. Formal methods provide a powerful way to review the requirements automatically. This paper proposes a method adopted from the formal verification of hardware components to uncover the deficiencies within a given set of requirements. The basis of this approach is built by safety properties represented as Linear Temporal Logic (LTL) formulas which are extracted from the requirements. The presented process is evaluated by means of the specification of a car seat.

1 Introduction

Formulating requirements and analyzing them is a vital prerequisite for keeping the development costs low. The set of requirements describing the attributes of a system is called specification. The development process usually starts with a customer delivering a rough specification that is then refined in talks with the contractor. A specification may change, be extended or completed during the development process. This may become an even bigger problem if the requirements describe a set of products, a so called product line (PL). The developer or design team, who must implement an actual product from the PL, relies on a consistent and complete specification. Therefore, single requirements must not contradict each other or become contradictory due to further customer wishes that change the requirements. The customer must define the characteristics and behavior of the desired product carefully and clearly. Specifications need to be complete and must not leave undesired degrees of freedom to the designer. Due to project teams working on single products and their requirements, it may also happen that redundant requirements are added to the set of requirements. Redundant requirements are those that are covered by one or more requirements that are already part of the specification. They do not provide any additional information to the system behavior.

The contribution of this paper is to provide support for requirements analysis of PLs of Embedded Systems by methods adopted from the formal verification of hardware systems and their components. These methods can prove the consistency of a system and deliver a

measure of completeness of the requirements. We put our focus on functional requirements that can be expressed as safety properties expressed in temporal logic. Non-functional requirements are not taken into account because they usually cannot be expressed by means of safety properties. We refer to functional requirements as requirements concerning the observable behavior of an embedded system. They define the valid input/output behavior the user observes when operating the system. The specification must specify the following ingredients with respect to the behavior of the system:

- inputs (sensors, buttons, messages...)
- outputs (actors, motors, messages...)
- temporal-logical relationships between inputs and outputs

If the embedded system is modeled as a black-box with inputs and outputs, the relation between the inputs and outputs is described by the functional requirements given by the specification. They state how an output has to react to a specific input or to a combination of inputs. The reaction of the output may also depend on internal states of the system.

The remainder of the paper is structured as follows: Related work is referenced in Section 2. Our running example is described in Section 3. In Section 4 an overview of the concept is presented. The property-checking algorithm is described in Section 5 and applied to the running-example in Section 6. In Section 7 the results are summarized and an outlook to future work is provided.

2 Related Work

The quality of specifications with respect to the requirements included may vary strongly. Goetz and Rupp show how specifications should be formulated to avoid contradictions and over-specifications [GR03]. Chantree et al. analyze ambiguities in natural language to improve the overall quality of the specification [CNdRW06]. For our approach the quality of the specification is of major importance.

Uncovering inconsistencies in specifications is discussed by Koopman et al. in [KAP08]. Lauenroth and Pohl address the problems that occur when dealing with variability in requirements analysis [LP08]. Both approaches relate to a model that is expressed as an automaton. The automaton must be extracted from the specification and may become very large for complex specifications.

In [HJL96] Heitmeyer et al. present a method to find several classes of inconsistencies within specifications. The SCR-notation (Software Cost Reduction) is used to define the requirements by means of condition, event and mode transition tables. Checking for contradictions or nondeterminism works similar to our approach, but the definition of completeness differs. Heitmeyer et al. state that a system is complete if every input event changes the system state. Our definition of completeness is more restrictive as we state that a system is complete if its outputs are defined for every possible input combination. Therefore,

our check reveals unwanted degrees of freedom within a specification. Furthermore, we normalize the property set which leads to an accurate view on contradictions and incompleteness and allows us to measure the completeness of a specification.

Bormann [Bor09] presents a technique for the analysis of property-sets. This approach lacks of providing a measure of completeness. Furthermore, the properties need to be written as so called transaction properties and cannot be extracted intuitively. Therefore, our approach is based on [OSE07] by Oberkönig et al. which also relates to the completeness of properties. We refer to a functional requirement as a system requirement to the system that states a functional interrelationship of the observable outputs based on the inputs and the current system state. Such an interrelationship is phrased as an LTL property rather than as an automaton.

3 Running Example

As a running example, we use a car seat controller known from the automotive industry. The car seat needs to fulfill the following requirements which we describe in form of an item list:

1. The seat incorporates three motors that can move the seat up/down, left/right and can change the angle of the backrest.
2. Each motor is controlled by a switch that can be brought in the three positions back-stop-front.
3. The motors can only be active up to a car speed of 15 km/h. The seat must not move over that speed due to safety reasons.
4. A seat heater is included into the seat that can reach the states off-on and is also controlled by a switch.
5. The seat heater can only be active if the seat belt is fastened.
6. Only one motor may be activated at a time or the seat heater may be changed.
7. The seat also incorporates an automatic entry function that helps the driver to access the car.
8. When the door is opened the seat moves back and the driver can enter.
9. When the door is closed the seat moves forward until it reaches the initial position.

3.1 Feature Models

Feature models (FM) are frequently used to describe variabilities and commonalities in a PL. A feature model is an acyclic graph consisting of features representing the properties

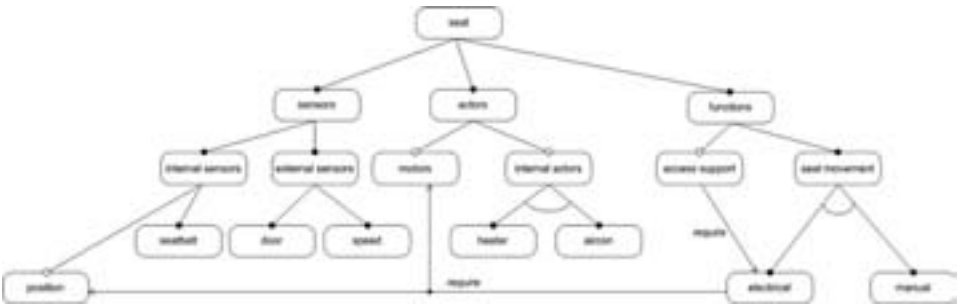

Figure 1: Feature Model of the car seat SPL

of the PL. A Boolean representation of an FM can be obtained according to [CW07]. We need a variability model that can be represented as a Boolean formula in order to add the information provided by the model to the LTL formulas. According to [HST+08] the purposes of a feature model are the following:

- describing common and variable features
- depicting dependencies and constraints between features
- determining which feature combinations are permitted and which are forbidden
- describing all possible products of the SPL

To satisfy these purposes, notations have been defined allowing to describe optional and mandatory features as well as groups of features from which exactly one feature (alternative-group) or at least one feature has to be selected (or-group). The hierarchical as well as cross-tree dependencies such as require and exclude decide which feature combinations are permitted. In [KCH+90] feature models are initially introduced as part of FODA. Fig. 1 depicts the feature model of our running example. The model and the names of the features is one possible representation for the variability incorporated within the PL. The feature model is no canonical representation. There are also other equivalent feature model representations that describe the same PL.

4 Concept

We extract formal properties from the functional requirements. Formal properties provide an unambiguous way to express functional relations between the inputs and outputs of a system. Usually, temporal logics, like Linear Temporal Logic (LTL) [Pnu77], express formal properties. We transform those formal properties into a normal form, and we then check if the properties suffer of one of the following deficiencies:

Contradiction: A contradiction within a set of requirements leads to an inconsistency that cannot be resolved by the system designer. Two or more requirements can contradict each

other by directly stating contradictory behavior, which leads to an over-specified output behavior. Moreover, contradictions may exist that cannot be identified at the first glance due to their indirect dependency on several requirements. A contradiction exists if two possible ways are specified in which the system can behave given the same set of inputs and the same system state.

Incompleteness: The set of requirements may be incomplete. We define that a set of requirements is complete if it describes the output behavior of every actor for every possible combination of inputs. If it is intended that the output behavior is unspecified in certain situations then this don't-care situation can also become part of the resulting property-set.

Redundancy: Two or more requirements are redundant if they state equal behavior of an output. Thus, they are specified in a non-contradictory way. This may trivially happen if several copies of a requirement exist or if a requirement can be deduced from other requirements. If the algorithm identifies these redundant properties, the set of requirements can be reduced without losing information about the system behavior.

Our approach starts with the initial specification, which may be given as plain text only. A state chart, activity diagram, or use-case may also serve as a specification of the system behavior. Functional requirements are extracted from the specification and arranged as a list. The list represents the specification in a semi-formal structure facilitating the extraction of properties. We also build an FM representing the commonalities and variability of the PL. The list of requirements must then be traced to the FM. For every requirement the respective feature or feature combination that causes the requirement to become part of a derived product from the PL must be known. The concept is closely related to the concept presented in [MO10] that aimed at the generation of test oracles from formal properties.

Afterwards, we derive LTL formulas from the previously formatted set of functional requirements according to Section 5. We then add the Boolean representation of the FM to the LTL formulas. The tracing of the features to a respective requirement is done by adding the single features to the LTL formula. The method presented in Section 5 then checks the formal properties automatically. For this check we do not need any further model because the properties incorporate all necessary information about the system behavior. Deficiencies are then detected and given to the user.

As long as deficiencies are found within the property-set, the deficiencies need to be analyzed and resolved in the list of functional requirements. Resolving the deficiencies must be done manually with the help of the results of the checking algorithm. Afterwards, the property-set must be modified due to the changes made within the list of functional requirements. This procedure is repeated until there are no more unwanted deficiencies in the formal properties. As a consequence, the underlying set of functional requirements is free from unwanted deficiencies, too. Since we added the information of the FM to the properties, we can also assure that there are no deficiencies left for every derivable product.

A major advantage of this procedure is that small changes of the functional requirements can be mapped to the existing set of formal properties easily, once the formal properties have been elicited. Using formal properties for requirement representation, newly added, removed, or changed requirements can be checked for adding unwanted deficiencies on the fly. For this procedure no implementation or further model is taken into account. The

properties are checked with respect to themselves. This procedure works with all types of temporal logics which incorporate the expressiveness necessary to describe the behavior of embedded systems. We use LTL since it is an established and well-known temporal logic.

5 Formal Analysis of Requirements

We focus on functional requirements that can be expressed by implications. A functional requirement can describe everything that represents the functionality of a system or a device. Statements like: "When the driver enters, the information to fasten the seatbelt must be displayed" can be seen as a global requirement. It omits the necessary implementation and leaves that to the design team. However, it can also be seen as an implication ($driver\_enters \rightarrow information\_displayed$).

A requirement that is extracted from a specification is called property in the area of formal hardware verification. Thus, the set of all requirements is called property-set. In order to describe a system formally, the properties may have to be extracted from a non-formal specification. We will explain the procedure by a small example taken from the running example:

...The seat heater may only be activated if the seat belt is fastened...

This excerpt leads to the following implication in LTL:

$$G(\neg seat\_belt \rightarrow X(\neg seat\_heater)) \tag{1}$$

If the seat belt is not fastened, the seat heater must not be active in the next cycle. As the requirement is modeled as a safety property the LTL formula begins with the Globally operator *G*. The Next operator *X* expresses the temporal relation between the left and right side of the implication.

Since we take a PL into account, a conjunction of the features that trigger the requirement must become part of the LTL formula. All mandatory features may be omitted because they are always part of the PL and, therefore, always true.

$$G(heater \wedge \neg seat\_belt \rightarrow X(\neg seat\_heater)) \tag{2}$$

This formula is only checked if the seat heater (*heater*) is part of the derived product.

Fig. 2a shows a path which is defined by the left side of an LTL formula. The next state of this path is undefined because none of the requirements defines a value for a certain output on the right side of the implication. Fig. 2b shows a path which has two concurrent next states. Two left sides of implications are valid on the same path but lead to two different next states defining contradictory values for the same output with their right sides. Fig. 2c shows a scenario describing redundancy. A specification consists of three requirements R1, R2 and R3. All the behavior which is defined by R3 has already been defined by R1 and R2. Thus, the requirement R3 is redundant. Considerable attention has been given to the problem of completeness-checking of test suites in the past few years A number of so-called "coverage tests" have been developed which examine different aspects of a test suite. Those tests usually deal with code coverage or control flow analysis [Lig02]. In the

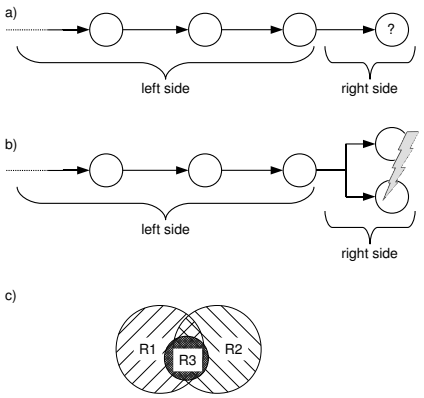

Figure 2: Possible Deficiencies

area of hardware simulation it has been state-of-the-art to use coverage metrics in order to evaluate the completeness of the simulation environment for more than ten years [CK93]. In contrast to that, our approach does not ask the question "Which part of the system is covered" but "Do the requirements incorporate all relevant functionality?".

In the field of formal verification of hardware systems (model checking with formal properties), a procedure was developed that is capable to analyze a specification consisting of properties with respect to its completeness [OSE07]. The completeness analysis calculates the *degree of determination* of all outputs within a specification. Checking the completeness of formal requirements may result in a gap. An output is called fully determined if it is set to a defined value for every combination of inputs and internal states.

We assume that inputs and outputs of embedded systems can only be set to the values '0' and '1' as known from digital hardware. Thus, other data types like integers or complex data types must be first transformed into an equivalent representation using bit vectors. Its size must be sufficient to cover the range of all possible values that the system can assign to the integer value.

Properties are subdivided into two groups: One limits an output in a specified case to '0' and the other one to '1'. The disjunction of all properties' left sides that force the output to become '0' define the so-called off-set (v_0). Accordingly, the disjunction of all properties' left sides forcing the output to become '1' defines the on-set (v_1).

We define the determination function and the consistency function from v_0 and v_1:

$$Determination\ function := v_0 \vee v_1 \tag{3}$$

$$Consistency\ function := v_0 \wedge v_1 \tag{4}$$

If the determination function equals '1' the output is fully determined (complete). In all other cases it is partly or completely undetermined.. Therefore, an output is fully determined if the disjunction of the values forcing it to '0' or '1' represents the complete set of inputs in all combinations. The output is then driven to a specific value on all possible input combinations without leaving any degrees of freedom. The counterpart of the determination

function characterizes all situations in which the output is not determined.

The consistency function must always be equal to '0'. Otherwise, the specification is inconsistent. If the conjunction of (v_0) and (v_1) becomes true, there must be at least one property that forces the respective output to '0' and '1' at the same time. The consistency function defines all situations in which the properties are inconsistent. These situations must be corrected within the requirements.

Definition: If the determination function of an output equals '1', the output is fully determined. In all other cases the following metric defines the percentage of the determined situations:

$$degree\ of\ determination = \frac{\#minterms}{2^n} \tag{5}$$

where n is the number of variables of the determination function and ***#minterms*** corresponds to the number of satisfied assignments in all Boolean variables of the function. A satisfied assignment in all Boolean variables corresponds to a single 1 in a Karnaugh map.

If there are, for instance, the following requirements for the output c given (as Boolean expression)

$$a \rightarrow c$$
$$\neg a \wedge b \rightarrow \neg c$$

the Karnaugh maps in Fig. 3 result. The diagrams represent the off-set, on-set and determination function of c, respectively. The Karnaugh maps depend on the two inputs *a* and *b*. In this example, the off-set is not the inverted on-set because the two properties do not fully determine the output c. Therefore, the degree of determination is 75 % since three minterms of a total of four are specified by the Boolean requirements.

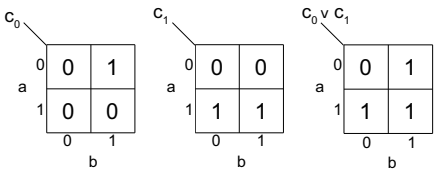

Figure 3: Karnaugh Maps of the Off- and On-Set

If the consistency function does not equal '0', inconsistencies have been found which represent contradictions within the specification. A metric for the degree of consistency is obsolete. For the algorithm that is capable of transforming properties into an off- and on-set please refer to [OSE07].

6 Exemplary Workflow

In the following we will present an exemplary workflow based on the running example from section 3. We will first write a complete and non-contradictory set of LTL properties using the procedure explained in section 4. Afterwards, we will demonstrate how contradictory and incomplete situations are detected.

6.1 Preparation

First of all, we formulate the requirements as LTL properties. Afterwards, we apply the procedure presented in section 5 to normalize the property-set and create microproperties. The algorithm uses the microproperties to analyze the requirements considering consistency and completeness. A degree of determination of 100 % is shown and it is proven formally that the properties are free from contradictions. The following two paragraphs show the main problems of the consistency analysis aided by slightly modified properties. Tab. 1 shows the number of generated microproperties, the total degree of determination, and the runtime. All results were produced on a Core2Duo system with 3 GHz using a single core.

Table 1: Results of the Car Seat Example

experiment	microproperties	determination degree	time
initial properties	40.282	100 %	35 s
with gap	40.274	97 % with 2 gaps	55 s
with contradiction	40.354	100 % with 250 contradicitions	34 s

6.2 Discovering Incompleteness

We modified a property for the seat heater according to equation (6) in order to create an incomplete property set. Equation (6) describes the behavior of the seat heater extracted from the requirements at the first glance. The completeness check detects a specification gap and returns a degree of determination for the seat heater (*Heat_output*) of 50%. The gap is depicted in Fig. 4 as a path. The diagram reveals the non-determined output due to the specification gap and the history leading to its non-determinism. The user is then requested to complete the path by adding the property that defines the output in the next state. In the example the behavior of the seat heater was not specified in the situation when the seat belt is not fastened.

$$G(Seatbelt \wedge Heat\_request \rightarrow X(Heat\_output)) \tag{6}$$

Therefore, equation (6) needs to be changed as follows:

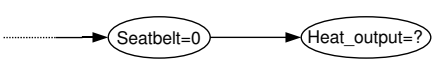

Figure 4: Path Representation of the Specification Gap

$$\begin{aligned} &G((Seatbelt \wedge Heat\_request \rightarrow X(Heat\_output)) \vee \\ &\quad (\neg Seatbelt \rightarrow X(\neg Heat\_output)) \vee \\ &\quad (Seatbelt \wedge \neg Heat\_request \rightarrow X(\neg Heat\_output))) \end{aligned} \tag{7}$$

Equation (7) defines *Heat_output* in all possible combinations of the inputs *Seatbelt* and *Heat_request* and leaves no further gaps. According to the gap free formal LTL specification

the initial set of requirements can be revised. The functional requirement (5) from section 3 should be written in a clearer way as follows:

- If the seatbelt is not fastened the heater must not be active.
- If the seatbelt is fastened the heater must be active if it receives a request from the corresponding switch and must be inactive otherwise.

These formulations lead to only one possible way in which the seat heater may operate and leave no degrees of freedom to the designer.

6.3 Discovering Contradictions

To force the specification to be contradictory we use the properties in Equation (11) to (14). Equations (11) to (13) describe the behavior of one of the engines with three properties. To ensure the readability of the resulting LTL expressions we substituted parts of the properties with corresponding macros (*M1; M2; M3*).

$$\begin{aligned} M1 :&(Engine2\_request = 0) \wedge \\ &(Engine3\_request = 0) \wedge (Speed\_Sensor < 16) \end{aligned} \tag{8}$$

$$M2 :(Engine1\_request = 1) \wedge (Seat\_Back > 0) \tag{9}$$

$$M3 :(Engine1\_request = 2) \wedge (Seat\_Back < 50) \tag{10}$$

$$G((M1 \wedge M2) \rightarrow X(Engine1\_output = 1)) \tag{11}$$

$$G((M1 \wedge M3) \rightarrow X(Engine1\_output = 2)) \tag{12}$$

$$G((M1 \wedge \neg M2 \wedge \neg M3) \rightarrow X(Engine1\_output = 0)) \tag{13}$$

Equation (14) is a small excerpt of the LTL property leading to the contradiction. It defines the output behavior due to bus conflicts of one or more accesses to the bus simultaneously.

$$\begin{aligned} G(((&Heat\_request \neq 0) \wedge (Engine1\_request \neq 0)) \rightarrow \\ &X((Engine1\_output = 0) \wedge (Invalid\_Inputs = 1))) \end{aligned} \tag{14}$$

The four properties have reached a size which does not allow the stakeholder to uncover the contradiction at the first glance. The formal analysis discovers a contradiction which can be represented as a path (Fig. 5). The path depicts the contradiction within the specification of the output of the motor by having two contradicting next states. It occurs when two devices try to access the bus simultaneously. Within this example an assumption of the property belonging to the motor was not stated that excludes the seat heater to be active together with the motor.

To resolve the contradiction *M1* needs to be changed as shown in Equation (15).

$$\begin{aligned} M1 :&(Engine2\_request = 0) \wedge (Engine3\_request = 0) \\ &\wedge (Speed\_Sensor < 16) \wedge (Heat\_request = 0) \end{aligned} \tag{15}$$

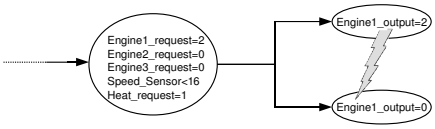

Figure 5: Path Representation of the Contradiction Discovered

Due to this change *Engine1_output* can only be active if no other device is accessing the bus simultaneously. In the initial set of properties the heater was omitted which lead to the contradiction.

Redundant properties can be discovered by leaving them from the property set. If the degree of determination remains unchanged the property is redundant.

7 Conclusion and Future Work

The later the stakeholder identifies deficiencies in a specification or in a set of requirements, the more time and effort is needed to redefine the affected parts of the system. Within the scope of this paper it could be shown that requirements can be tested for incompleteness, contradictions and redundancy with formal methods taken from the hardware verification. The challenge of this approach is to formulate the requirements by means of the property specification language LTL. Our approach is suitable to review specifications which were formulated by a stakeholder. Since the resulting property-sets can easily and intuitively be modified and extended, our approach can be used in the field of requirements engineering. When extending the LTL property set with the boolean representation of the FM, we can also check requirements of PLs. Especially, when reviewing evolving specifications the approach is capable to help maintaining the consistency.

As a next step, we need to review the kind of requirements that can be modeled with property specification languages. Up to now, we used the algorithm exclusively for systems that can be described by a set of safety properties. We also need to determine how errors in the extraction process of properties from the specification impact the completeness and if they can be identified. The maximum problem size that can be handled must also be evaluated.

LTL is used in many different domains to specify system behavior. Therefore, we need to figure out whether our approach can handle specifications from other areas, like business processes. In principle the approach should be suitable for any specification that can be expressed as safety properties.

Finally, we will try to adopt the method presented in [Bor09] to tackle the completeness problem. We need to evaluate the differences with respect to the drawbacks and advantages of both methods.

References

[Bor09] Jörg Bormann. *Vollständige funktionale Verifikation*. PhD thesis, Universität Kaiserslautern, 2009.

[CK93] Kwang Ting Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using The Extended Finite State Machine Model. In *DAC '93: Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 86–91, New York, NY, USA, 1993. ACM Press.

[CNdRW06] Francis Chantree, Bashar Nuseibeh, Anne de Roeck, and Alistair Willis. Identifying Nocuous Ambiguities in Natural Language Requirements. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 56–65, Washington, DC, USA, 2006. IEEE Computer Society.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Software Product Line Conference*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[GR03] Rolf Goetz and Chris Rupp. Psychotherapy for System Requirements. In Dilip Patel, Shushma Patel, and Yingxu Wang, editors, *IEEE ICCI*, pages 75–80. IEEE Computer Society, 2003.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5:231–261, 1996.

[HST+08] Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Yves Bontemps, Raimundas Matulevicius, and Andreas Classen. Evaluating formal properties of feature diagram languages. volume 2, pages 281–302, 2008.

[KAP08] Pieter Koopman, Peter Achten, and Rinus Plasmeijer. Testing and Validating the Quality of Specifications. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 41–52, Washington, DC, USA, 2008. IEEE Computer Society.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[Lig02] Peter Liggesmeyer. *Software-Qualität*. Spektrum Akad. Verlag, Heidelberg [u.a.], 2002.

[LP08] Kim Lauenroth and Klaus Pohl. Dynamic Consistency Checking of Domain Requirements in Product Line Engineering. In *RE '08*, pages 193–202, Washington, DC, USA, 2008. IEEE Computer Society.

[MO10] Florian Markert and Sebastian Oster. Model-Based Generation of Test Oracles for Embedded Software Product Lines. In *Workshop Proceedings of the 14th International Software Product Line Conference*, 2010.

[OSE07] Martin Oberkönig, Martin Schickel, and Hans Eveking. A Quantitative Completeness Analysis for Property-Sets. In *FMCAD '07*, pages 158–161, Washington, DC, USA, 2007. IEEE Computer Society.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. *Symposium on Foundations of Computer Science*, pages 46–57, 1977.