Linking Functional Requirements and Software Verification*

Hendrik Post, Carsten Sinz
University of Karlsruhe
Institute for Theoretical Computer Science
Karlsruhe, Germany
{post, sinz}@ira.uka.de

Florian Merz, Thomas Gorges, Thomas Kropf
Robert Bosch GmbH
Chassis Systems Control
Leonberg, Germany
Thomas.Gorges@de.bosch.com

Abstract

Synchronization between component requirements and implementation centric tests remains a challenge that is usually addressed by requirements reviews with testers and traceability policies [16]. The claim of this work is that linking requirements, their scenario-based formalizations, and software verification provides a promising extension to this approach. Formalized scenarios, for example in the form of low-level assume/assert statements in C, are easier to trace to requirements than traditional test sets. For a verification engineer, they offer an opportunity to better participate in requirements changes. Changes in requirements can be more easily propagated because adapting formalized scenarios is often easier than deriving and updating a large set of test cases.

The proposed idea is evaluated in a case study encompassing over 50 functional requirements of an automotive software developed at Robert Bosch GmbH. Results indicate that requirement formalization together with formal verification leads to the discovery of implementation problems missed in a traditional testing process.

1 Introduction

The importance of linking testing and requirements,e.g., via *test traceability* has been investigated in a recent case study reporting practices and experiences from Finnish organizations [16] and is supported by previous work, e.g., by Graham [10]. A tight link will likely improve the outcome of the software development process. Lindstrom [13] even claims that missed links between people or documents will lead to a flawed product.

*This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

In this work, an emerging trend in industry, employing *software verification*, is integrated into this scenario. Software verification is a technique to provide formal guarantees that software implementations conform to their specifications. Recently, several approaches for verification have reached a status where successful integration into the industrial software development process has been achieved [1, 5, 6, 12].

The applicability of formal methods is also reflected in *formal requirements analysis*. The aim of this approach is the qualitative improvement of requirement documents by directly translating them into a formal language. The goals—formally proved consistency and early defect detection—are shared between both approaches. A difference, though, is that integration of formal methods is perceived on the implementation level for verification, and on the level of documents or artifacts for requirements analysis.

If testing and requirements need to be linked, the same should hold for requirements and software verification. We therefore review and perform software verification from the perspective of checking consistency between component requirements and C implementations.

In contrast to other software verification case studies, specifications are not derived from an abstract correctness goal (e.g., termination), but from a set of dynamically changing functional requirements. Up to now, it was mainly unknown whether verification can handle the timing constraints posed by industrial development processes and how the technique can be linked to component requirements.

The case study performed by Uusitalo et al.[16] analyzed best practices and experiences for linking testing and requirements by interviewing experts. We cannot adapt their interviewing technique in our setting because verification is not yet integrated into the industrial development process. Therefore we provide our own case study, where we formalize a set of 50 requirements, and verify that software releases conform to them using an automatic technique called

software bounded model checking. We use the tool CBMC developed by Daniel Kröning at the University of Oxford as bounded model checker [4]. Even though the application of an academic tool is challenging by itself, we provide detailled insights on the experience of applying verification concurrently with changing requirements and over multiple releases.

Our conclusion is that the need for linking testing and requirements must be extended to incorporate verification. The quality improvement of the process is indicated by reporting ten safety-critical violations of functional requirements.

The software we analyze in this paper is part of a product for automotive driver assistance developed by Robert Bosch GmbH. In the case study, three components implementing redundant safety monitors are covered. They are highly safety critical, as they implement a watch-dog functionality on top of the main systems. A common functional requirement in this domain is that the driver assistant system shall remain passive if any kind of internal error, e.g. video blindness, has occurred.

We now present a brief introduction to software verification, and software bounded model checking in particular.

2 Software Verification

Even though it is well known that proving non-trivial properties of programs is undecidable in general, techniques have been developed that are able to deal surprisingly well with industrial software. One major approach is *correctness by design*, where the whole development process is centered around model-based preservation of correctness properties. The central notion of this field is *model refinement* that links initial, abstract mathematical models with more detailled ones that can eventually be used to programmatically generate correct source code. *Correctness by design* is certainly appealing, but still not widely used in industry. The reasons for this are that, first, generated programs cannot easily be modified after being generated, and, second, that each requirement change triggers a complete reconstruction, including all intermediate steps down to the generated source code. As the process is not automated—because of the powerful formalisms used—practitioners seem tentative to perform this development approach.

The more popular alternative is *implementation-specification* based consistency checking. Implementation is achieved by a conventional development process and verification is performed, similar to testing, on the implementation using a low-level formal specification. We claim that the latter approach is more easily integrated into industrial practice, as no need for changing existing processes arises. Our work solely deals with implementation-specification based verification.

We will now review the specification technique used, and its application to C implementations. Afterwards, we describe the automatic technique called *bounded model checking* for verification.

2.1 Specifications

In order to make the process of verification more transparant to requirement engineers, we have chosen a low-level formalism for specification that resembles the syntax and expressiveness of a normal imperative programming language. *Assert/assume* based specifications make use of the fact that most interesting functional properties can be expressed in terms of the source code language. Similar to test cases, one can insert `assert(expr)` statements into the code. The verification condition is encoded in `expr`. We illustrate this by a small example:

```
// INPUT:
//    int a, the dividend, a is non-negative
//    int b, the divisor
// OUTPUT:
//    int c, if "(b!=0)" the result of "a/b",
//           else return int constant UNDEF

int divide(int a, int b) {
   int result;
   assume(a >= 0);
   if (b==0)
       result = UNDEF;
   else
       result = a / b;
   assert( (b != 0) ?
              result == (a / b) :
              result == UNDEF
   );
   return result;
}
```

The above code shows a C implementation of a division function that catches the special case that the divisor may be zero. The non-formal specification is present in the comments. The formalization of the specification is encoded in the `assume` and `assert` statements. Conditions that must hold prior to the execution of a function are classically named *preconditions* and related to `assume` statements. Conditions that must hold after the function has been executed (if the preconditions hold), are called *postconditions*, and are encoded in `assert` statements. Formalizing functional properties in this way is easy to understand for testers. It seems less obvious, though, how to link these C expressions to component requirements.

2.2 Requirements Formalization by Scenarios

Requirements are first filtered whether they actually contain functional specifications that can be checked in the im-

plementation. Similar to scenario-based testing [15] they can be expressed in a description under which circumstances restrictions on the output should hold. An example for this the following requirement taken from our case-study:

Requirement 1 *If the video sensor is not working, the driver assistance system shall not act.*

In the following we will derive a formal scenario out of the textual description. At first, design documentation needs to be consulted. For this product, design artifacts consist of textual documentation, an additional tool-supported documentation defining the semantics of every interface variable, as well as developer knowledge. The first question is how the fact that the video sensor works can be observed in the C implementation of the product. By the design documentation we know that this is encoded in the following way:

Design Artifact 1 *The variable* `VIDEO_SENSOR` *is set to one if and only if the video sensor is working.*

This information already allows to formalize the `assume` part of the scenario: `assume(VIDEO_SENSOR != 1)`. In order to complete the scenario, the `assert` condition has to be added:

Design Artifact 2 *The driver assistance system acts if and only if the result of the* `veto` *function is* `true`*.*

According to the above design information we can complement the scenario: `_Bool vetoed = veto();` `assert(vetoed==true);`. The embedding of the two parts is performed by a test (or verification) function derived from a general pattern:

```
// task called every 100ms
main()
{
    havoc();
    assume(VIDEO_SENSOR != 1);
    ...
    component_task();
    ...
    _Bool vetoed = veto();
    assert(vetoed==true);
}
```

The `havoc` function unrestricts the global state of the component, e.g. if a global variable is initialized to zero, `havoc` will "remove" this assignment in order to obtain an overapproximation of possible inputs. This is necessary in order to prove that the code is correct for *every* possible input. Otherwise executions *after* the first execution of `main` would not be included in the analysis.

We now can define the term *formalized scenario*:

Definition 1 *A* formalized scenario *is a pair of C expressions* $\langle \mathrm{pre}, \mathrm{post} \rangle$ *encoding the precondition* pre *and the postcondition* post *for a C function.*

For the given example, the notation of the scenario looks as follows:

$$\langle \texttt{(VIDEO\_SENSOR != 1)}, \texttt{(vetoed==true)} \rangle$$

Even though the definition of scenarios is of a low-level kind and restricted to pre- and postconditions on a functional level, we found that almost all requirements we analyzed could be easily formalized this way.

In addition to the formal parts of the scenario, we maintain an implementation of the automatically derived `havoc` function for each component. The link between a formalized scenario and component requirements is maintained in a document.

The next section gives a brief overview how an automatic verification technique called *bounded model checking* is able to prove that an implementation complies to a formalized scenario.

2.3 Software Bounded Model Checking

Bounded model checking (BMC) is a technique that was introduced by Biere *et al.* [2] to check properties of hardware designs, but has later been extended to also allow verification of C programs [4]. Bounded model checking generates program execution traces with bit-precision on the data level. It cannot handle unlimited recursion, however, and restricts loop executions to a fixed bound (by unwinding loops up to this bound). If the bound is high enough to capture the system semantics, BMC is sound and complete. If the bound is too low a warning about the possible unsoundness will be provided. Values of fixed-size variables are not approximated, but handled on the bit-level instead. Variable-size (possibly infinite) data structures must be approximated, though. This means, that BMC implements a precise program semantics up to the point where the loop, recursion or data structure size bound is reached. By using a precise semantics on fixed-size variables, effects like overflows can be handled accurately. An implementation of BMC for C programs is CBMC [4]. We have used CBMC for the experiments reported in this paper. Internally, CBMC generates program traces by modelling the effects of each program statement as a propositional logic formula. Each formula encodes a single-step transition relation. These formulas are then put together for all program steps on a trace. A further formula is added that excludes entering error states. The complete formula is then checked by a propositional logic satisfiability checker. The execution bound imposed by BMC removes the need for a fix-point computations, which differentiates bounded

model checkers from general symbolic model checking algorithms, in which fix-points are computed. General model checking approaches are also in use for software verification, and may either work on abstract models which resemble abstract domains, or they follow a dynamic refinement process where modelling precision is automatically gradually increased as needed (CEGAR). Bounded model checking has already been applied successfully for medium to large scale software projects, e.g. for checking the correctness of Linux kernel modules [14]. Model checking by abstraction refinement has also been applied successfully [1, 3].

3 The Case Study

Our case study employing software bounded model checking has been performed in parallel to a normal FMEA (Failure Mode and Effects Analysis) development process. In addition to requirements engineers, test managers and testers, a verification engineer (in our case a student working on his master thesis) was added to the development team. The task of this engineer is to formalize requirements into scenarios. Over time, the requirements need to be updated and results have to be communicated to developers as well as to requirements engineers.

3.1 Driver Assistance Software

In our case study we consider three software components. The first component is the software part of the *Adaptive Cruise Control (ACC)* system. ACC is applied in cars to monitor the current traffic situation. If an obstacle, e.g. a truck, is detected in front of the car, the driving speed is adapted such that the driver does not need to hit the breaks manually. If the obstacle does not block the road anymore, ACC accelerates the car to a pre-selected driving speed. ACC is a *driver comfort system*. Nevertheless errors in requirements or implementations could lead to unexpected safety-critical situations, e.g. the car does not slow down even though the driver expects it to do so.

Two other similar components are also incorporated into our study. For confidentiality reasons, we had to anonymize their names and requirements.

3.2 Process

For each requirements and software release the following sequence of steps is performed:

1. Filter requirements to determine whether they can at all be violated by the implementation. Non-functional requirements like "a Simulink model must be provided" are excluded from the verification.
2. Map new requirements to former formalized scenarios, if they exist. Otherwise set up a new scenario.
3. If requirements were changed since the last verification run, update all scenarios that are linked to them.
4. Formally verify all scenarios (using the software bounded model checker) where either the formalization or the corresponding implementation changed.
5. Report results to requirements engineers and to developers.
6. Check that the problems do not persist after bugfixes.

In our experiments, two releases of the requirements documents and numerous small updates on the C implementation were covered. The verification engineer interacted with a requirements management system which is used to manage customer requirements in textual form. Design artifacts were also at hand. In some cases, developers had to be asked in order to gain additional information about the encoding of some system states.

Note that the creation of scenarios is a task that involves a dependency analysis of the requirements. In our experiment several artifacts are linked. The most common case is a case distinction between different error states that the system could have:

Requirement 2 *If sensor_A fails,* ϕ.

Requirement 3 *If sensor_B fails,* ϕ.

Requirement 4 *If any sensor fails,* ϕ.

Requirement 5 *If no sensor fails,* $\neg\phi$.

Where ϕ is any constraint on the behaviour of the function. These cases are inherently linked together. One could formalize one scenario that encompasses all conditions or split them into four cases. In any case the terms *any sensors* and *no sensor* are defined in terms of the set of concrete sensors, e.g. {sensor_A, sensor_B}. The exact mapping of requirements to scenarios is a matter of granularity and varies for different components (cf. Table 1). A finer granularity allows for more detailled results. A scenario that is not matched by the behavior of a C program may violate *multiple* or *single* requirements as indicated in Table 2.

3.3 Technical Results

Results we have obtained are of two kinds: (1) about the requirements formalization process, and (2) about the verification process using software bounded model checking. For the latter, we report on runtimes for the verification tool as well as on the number of problems we found.

Table 1. Number of formalized requirements, by component.

Component	Total Requirements	Release 1	Release 2	Functional	Formalized	Scenarios
ACC	33	27	13	30	21	24
B	40	26	40	33	21	15
C	8	0	8	8	8	8

3.3.1 Formalizing Requirements

Table 1 documents the amount of requirements that we have translated into scenarios during two releases of the requirements documents[1]. It should be noted that the set of requirements varied to a great extend between the two releases.

In Table 1, the first column indicates the name of the component to check. The further columns, in turn, denote the number of total requirements specified for the component over all releases, the number of requirements given in the first and second release, the number of requirements that express functional properties of the component. The second but last column shows the number of functional components that were accessible to formalization, and the last column gives the number of formal scenarios we created out of the formalizable requirements. For component ACC, some requirements were split up into multiple scenarios.

We were able to formalize 70%, 63%, and 100% of the functional requirements for the three components that we analyzed. Some requirements were not accessible for scenario formalization, however. The reasons for this are as follows:

- In case of component B, the second release of the requirements documents occurred only a few days before the project ended. Due to a lack of time, new requirements were only partially considered.

- Two requirements were actually misclassified, as they restricted the behaviour of other components of the system. While creating the scenarios, this misclassification was detected and reported to the requirements managers. As a consequence they did not appear in the second release.

- In some cases, the scenario would have been too close to the implementation, e.g., the requirement is stated in terms that are solely defined in the implementation: "The distance to another car is sufficient."

We continue to report what kind of problems we could identify by using the software bounded model checker CBMC [4].

[1]Due to the fact that the verification engineer performing the experiments was not a formal member of the company, access to the requirements management system was restricted to two releases of the requirements documents.

3.3.2 Checking Requirements

Table 2 gives an overview about the ten implementation related errors we discovered. As mentioned before, two requirements were detected to be assigned to the wrong component while formalizing. It follows an example of an implementation specific error (corresponding to rows 1 and 2 in Table 2).

In ACC time is discretely represented by a counter variable. During a refactoring, a function for the calculation of time differences was replaced (the C code for a fraction of this function is given below). The new version contained an arithmetic overflow of a variable. The result of this overflow is that a function that encodes a safety monitor is not called during a time window of 2.2 seconds every 30 seconds. As the function is not called, all requirements that depend on the safety monitor to be active are not satisfied. In this case the error can clearly be considered an implementation error.

The newly created code for detecting the overflow did contain a new error:

```
unsigned short x = . . . ;
unsigned short y = . . . ;
if ( ( unsigned short ) ( x + y )
                    < _USHORT_MAX)
 { ... } // no overflow
else
 { ... } // overflow
```

The `else` branch of the code is only executed if `(unsigned short) ( x + y )` equals exactly `_USHORT_MAX`. The intention of the `else` branch was to catch any overflow which was later achieved by introducing new casts to larger datatypes.

The other errors (3-9) were of similar nature involving low-level technical implementation problems. Error 10 was caused by a requirement change (concerning interval bounds) that was not reflected in the implementation.

We have found three cases in which testing revealed errors that the scenario based formalization missed. The reasons for this were:

- In one case, the formalization of the scenario was incorrect.

- A requirement (concerning existence of a C function) could not be formalized.

- An error in the verification backend masked a floating point related error.

It is well known that the quality of verification approaches highly depends on the quality of the specifications. Our finding is that, even though the verification engineer was not familiar with the requirements, the design, and the implementation, few errors were missed. Moreover, we found that testing and verification complemented each other very well. Interaction between testers and the verification engineer allowed to increase the quality of the implementation in a shorter time than with testing alone.

In addition to the implementation problems we detected some inaccuracies in the requirements. These are given in Table 3.

3.3.3 Runtimes

Tables 4 and 5 give information about typical runtimes of CBMC for successful proofs of formalized scenarios. The number of *assignments* refers to the number of equations $x = expr$ the C program is converted into by the bounded model checking procedure (by unwinding loops, inlining functions, etc). These equations are later converted by CBMC into bit-vector logic equivalences. A *claim* is one proof obligation that CBMC has to prove. This number encompasses the `assert` statements as well as proofs that the unwinding was sufficient and that no generic runtime errors (like invalid memory accesses, array-out-of-bound-errors, or arithmetic overflows) occur in the program. Some claims can be proven by heuristics before actually running the core propositional logic (SAT) engine inside CBMC—the number of *remaining claims* is also listed in Table 5. CBMC then translates the set of bit-vector equations into a propositional logic formula in conjunctive normal form (CNF). The size of the generated propositional logic formula is given in columns *variables* and *clauses*.

Runtimes and problem sizes are stable for the verification of different scenarios in the same component as shown in Table 4. As expected, the number of assignments, which relates to the number of statements in the program, the number of claims and the number of clauses varies greatly for different components. Surprisingly the number of variables roughly matches for different components. This effect is caused by the inclusion of generic heasder files that define large and static data structures. Unread and unwritten memory locations do not incur a clause overhead even though they imply a large variable number.

4 Evaluation

Our results indicate that requirements management and implementation quality can greatly benefit from introducing software verification. The communication between testers, developers and requirements managers can be facilitated using technical scenarios. The success of our method can be seen from the number of new errors we found by our verification approach. It is well known that requirements change quickly during early development phases. Using scenarios, the need for updating a large number of test cases could be reduced to the update of less than 50 scenarios. Figure 1 illustrates the benefit of linking scenarios rather than test cases with requirements.

The reason for better synchronization of requirements and scenarios is that the number of scenarios is significantly lower. A naive tester whould introduce a potentially large number of test cases to achieve branch coverage for a function. As these can be summarized in a single scenario, less work is required using our scenario-based verification approach. In addition to the sheer improvement in link quantity, the construction of scenarios can be driven by requirements coverage. Testing has usually different coverage goals, e.g. MCDC (Modified Condition Decision Coverage) test coverage for safety critical software. In order to obtain a coverage level, white box tests have to be constructed, guided by the need to cover certain paths and conditions. The information which requirements are affected by the constructed testcase is derived afterwards. Hence, design of the test cases is not driven by (functional) requirements and therefore maintainability is diminished.

The advantages of the method we propose manifested in several aspects in our case study: Two requirements have been identified to be assigned to the wrong component. The formalization of scenarios has further contributed to find missing requirements for one component.

A noticeable contribution of this work is to demonstrate that software verification of functional properties can be executed concurrently to a normal development process. In contrast to other case studies—which are either dealing with generic properties like runtime-errors, or are executed offline on a snapshot of a real system—our study features a full functional analysis tightly integrated into the software development process which can be iterated for every release of requirements and implementation.

During our experiments we rarely observed that scenarios have not been correctly formalized. Such a faulty formalization has at least lead to one missed error in the implementation (that was detected by testing). *Formal requirements analysis* is a technique to identify such problems on the level of requirements. In contrast to this technique, we are working on scenarios that are derived later in the development process. Even though scenarios can only be formulated after deriving a concrete design and component requirements, it may still help by allowing to check scenarios for consistency. For a given set of k scenarios the following formula is satisfiable if and only if the scenarios, all taken

Table 2. Implementation errors found by software verification

	Component	Technical Reason	Requirement Violation Granularity
1	ACC	Possible overflow of a variable	Multiple
2	ACC	Wrong overflow check	Multiple
3	ACC	Missing functionality	Single
4	B	Conversion error of input message	Single
5	B	Misuse of a macro definition	Multiple
6	B	Missing shift operation	Multiple
7	B	Overflow repair incorrect	Single
8	B	Incorrect order of read / write	Multiple
9	B	Incorrect encoding of bit-masks	Multiple
10	B	Missed requirement change	Single

Table 3. Requirement flaws found

	Component	Flaw	Found by	Impact
1	ACC	Requirement assigned to wrong component	Scenario Formalization	Component requirement link updated
2	ACC	Requirement assigned to wrong component	Scenario Formalization	Component requirement link updated

together, are consistent:

$$\bigwedge_{1 \leq i \leq k} (\mathrm{pre}_i \rightarrow ren(\mathrm{post}_i))$$

The function $ren(\phi)$ renames all symbols x to new symbols x' (not occurring in ϕ) in ϕ. Note that the effect of the implementation is completely ignored here. We just check whether the set of all pre- and post-conditions can be simultaneously satisfied. Hence, consistency of scenarios can be checked by a SAT solver. The complexity is by far smaller than checking whether a whole C implementation satisfies one scenario.

4.1 Future directions

A potential direction of research that arose during our experiments is the idea to use scenarios to derive test cases. Then, verification could catalyse the communication between the component requirement world and testers.

A second direction of reasearch is the integration of formal requirement analysis into this scenario. Our scenarios could be viewed as an implementation of the high-level formalizations. Automata-based formalizations, e.g. by Heitmeyer et al. [11] seem close enough to be checked against our C formalizations.

4.2 Limitations

Due to disclosure agreements we cannot give more information on the detailed outcome of the software tests. It would be interesting to directly compare the number of errors found using both methods. Efforts from developers performing unit testing are currently not documented. Information about the exact amount of time spent for testing thus cannot not be given. The comparision would in any case be less significant as the verification engineer stems from an academic environment infamilar with the product and the developement processes.

This case study encompasses code from three components with more than 3000 lines of source code. An extension to a whole product would certainly strengthen the obtained results. As mentioned before, some of the requirements for component B could not be formalized due to the fact that they were finalized only a couple of days before our verification project ended. The engineer chose to verify already formalized requirements instead.

4.3 Related Work

Three areas of research are related to our proposal: software verification case studies, formal requirements analysis, and work about linking testing and requirements.

It has been demonstrated in many case studies that numerous verification techniques can be applied to real world software. Well known examples include the Microsoft SLAM project [17], which led to an interface specification verification tool that is currently deployed with every driver development kit. Cook et al. present the Terminator [5] tool, which is able to check certain termination properties of windows device drivers. Other examples include

Table 4. Runtimes and problem sizes for different ACC scenarios.

Scenario	Assignments	Claims	Remaining	Variables	Clauses	Runtime [s]
S1	4003	279	29	762899	281276	16.984
S2	4003	279	29	762899	281283	16.826
S3	4003	279	29	762899	281276	16.985
S4	4010	279	29	795174	303029	19.003
S5	4003	279	29	762899	281276	16.953
S6	4003	279	29	762899	281276	16.970
S7	4003	279	29	762899	281276	16.949
S8	4003	279	29	762899	281283	16.925
S9	4003	279	29	762899	281276	16.964
S10	4010	279	29	795174	303029	19.825
S11	4003	279	29	762899	281276	16.938
S12	4003	279	29	762899	281276	16.948
S13	4003	279	29	762899	281279	16.965
S14	4003	279	29	762899	281276	16.956
S15	4003	279	29	762899	281276	16.870
S16	4000	279	29	762982	281495	16.957
S17	4000	279	29	762899	281283	16.922
S18	4000	279	29	762899	281283	16.928
S19	4003	279	29	795038	302713	18.021
S20	4122	279	29	860168	347565	59.772
S21	4122	279	29	859180	344664	21.453
S22	1526	7	7	748671	92051	12.022
S23	4017	278	28	762414	279556	27.515
S24	3976	277	27	762073	278837	18.468

Table 5. Typical runtimes and problem sizes for verifying a single scenario for different components.

Component	Assignments	Claims	Remaining	Variables	Clauses	Runtime [s]
ACC	4003	279	29	762899	281276	16.984
B	3865	291	31	759635	273611	15.247
C	843	5	5	645150	9165	6.062

abstract interpretation tools, which are successfully applied in avionics industry [7]. CBMC [4] has been successfully applied to numerous complex software systems [12, 14].

Formal requirements analysis deals with formalizations of requirements in an early design phase. Noticeable demonstrations of this technique are given by Dutertre and Stavridou [9] in the area of avionics using non-automatic theorem provers. Crow and Di Vito [8] present a summary of four case studies in space craft industry using non-automatic proof systems. An automata based approach that is more closely related to the model checking technique we presented was proposed by Heitmeier et al. [11].

Uusitalo et al. analyze best practices for linking requirements and testing in industry[16]. Graham argues that testers should be integrated into early developement phases [10]: Both sides can profit from a tight linking.

5 Discussion

Formal scenarios provide a promising alternative for maintaining links between implementation analysis and component requirements. For parts of an industrial product the technique has proven to be extremely efficient. Another advantage is that it can be applied concurrently with testing. The effectiveness is substantiated by the dectection of ten implementation errors as well as improved adaptability of quality assurance tasks to requirements changes. The link between requirements and scenarios is still informal, but due to the fact that the number of scenarios is orders of magnitude smaller than the number of test cases, adaptability is increased.

Testing and verification have proved to show a great synergy, which is reflected by the fact that both methods detect

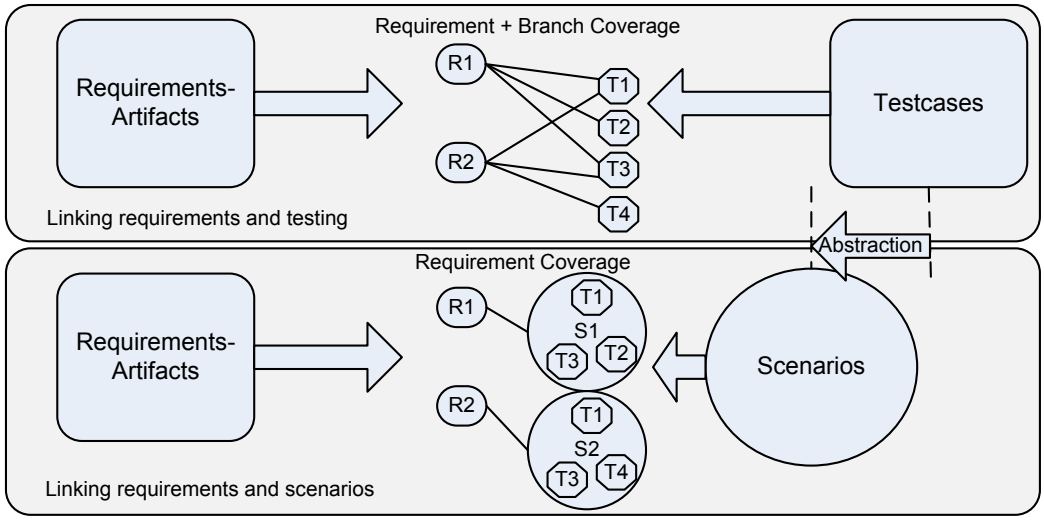

Figure 1. Linking requirements (R1 and R2) with a possibly high number of test cases (T1-T4) poses a significant synchronisation and communication challange for testers and requirements engineers. Links between requirements and formal scenarios (S1 and S2) benefit from the fact that one scenario encompasses a set of test cases.

unique errors. It is notable that verification *could* be integrated in a development process. Even though verification was performed by an external engineer, some results were obtained ealier than through testing. One reason for this efficiency is the higher manageability for scenarios linked to requirements.

Finally, we conclude that verification techniques like bounded model checking have reached a stage of development, where they can be employed with considerable benefit in an industrial setting.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys Conf., Proc.*, pages 73–85. ACM Press, 2006.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[3] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of the 9th ACM Conf. on Computer and Communications Security (CCS)*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[5] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 415–418. Springer, 2006.

[6] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[7] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proc. of the 7th ACM & IEEE Intl. Conf. on Embedded software (EMSOFT)*, pages 7–9, New York, NY, USA, 2007. ACM.

[8] J. Crow and B. Di Vito. Formalizing space shuttle software requirements: four case studies. *ACM Trans. Softw. Eng. Methodol.*, 7(3):296–332, 1998.

[9] B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23:267–278, 1997.

[10] D. Graham. Requirements and testing: seven missing-link myths. *Software, IEEE*, 19(5):15–17, Sep/Oct 2002.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[12] M. Kim, Y. Kim, and H. Kim. Ieee/acm int. conf. on automated software engineering (ase). In *Int. Conf. on Automated Software Engineering (ASE), Proc., (to appear)*. IEEE Computer Society Press, September 2008.

[13] D. R. Lindstrom. Five ways to destroy a development project. *IEEE Softw.*, 10(5):55–58, 1993.

[14] H. Post and W. Küchlin. Integration of static analysis for linux device driver verification. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods (IFM), 6th Intl. Conf., Proc.*, volume 4591 of *LNCS*, pages 518–537. Springer-Verlag, 2007.

[15] J. Ryserł and M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. In *In 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA99*, page 7, 1999.

[16] E. J. Uusitalo, M. Komssi, M. Kauppinen, and A. M. Davis. Linking requirements and testing in practice. In *RE '08: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*, pages 265–270, Washington, DC, USA, 2008. IEEE Computer Society.

[17] Various. The SLAM Project. http://research.microsoft.com/slam/, 2006.