SmartTies
Management of Safety-Critical Developments*

Serge Autexier, Dominik Dietrich, Dieter Hutter, Christoph Lüth, and Christian Maeder

Cyber-Physical Systems, DFKI Bremen, Germany

Abstract. Formal methods have been successfully used to establish assurances for safety-critical systems with mathematical rigor. Based on our experience in developing a methodology and corresponding tools for change management for formal methods, we have generalised this approach to a comprehensive methodology for maintaining heterogeneous collections of both formal and informal documents. Although informal documents, like natural language text, lack a formal interpretation, they still expose a visible structure that reflects different aspects or parts of a development and follows explicit rules formulated in development guidelines. This paper presents our general methodology for maintaining heterogeneous document collections and illustrates its instantiation in the SmartTies tool that supports the development of safety-critical systems. SmartTies utilises the structuring mechanisms prescribed in a certification process to analyze and maintain the documents occurring in safety-critical development processes.

1 Introduction

With the advent of sophisticated intelligent systems (so-called cyber-physical systems), there is an increasing need to guarantee the safety of such systems. Formal methods have been successfully used to establish such assurances by providing mathematical proofs that specifications or implementations satisfy required properties. Industrial applications revealed that a flexible, evolutionary formal development approach which efficiently supports changes is absolutely indispensable as it was hardly ever the case that the development steps were correctly designed in the first attempt.

In contrast, standards like IEC 61508 [10] or DO-178B [14] address the problem of establishing trust in such systems by regulating the development process, requiring that all design decisions and safety arguments are documented in meticulous detail. The documents arising during the development mutually depend on each other, and changes in one document typically give rise to changes in others. This makes changes cumbersome, thus decreasing flexibility. Further, the amount of these dependencies explodes with the size of the developed system.

* This work was funded by the German Federal Ministry of Education and Research under grants 01 IW 07002 and 01 IW 10002 (projects FormalSafe and SHIP)

There is a need for an efficient computer-aided document management that keeps track of the various dependencies in and between documents occurring during the development of safety-critical systems.

Existing tools do not cover this in full generality. They either cover specific aspects of the development process (like DOORS [9], which handles requirements, or the iACMTool [6], which handles UML models), or are specialised to a specific application domain and development methodology (for example, PREEvision [12] to develop safety-critical systems in the automotive industry using a model-based approach); they incorporate specialised knowledge about the underlying domain in fixed rules for maintenance.

Our goal is a generic maintenance and change management tool that can be tailored to deal with heterogeneous document collections, to maintain the corresponding dependencies and relationships, and to exploit them to propagate or to restrict the impact of changes made in the documents [8]. The contribution of this paper is the SmartTies tool, which supports the document types, operations and the workflow typically occurring in the development and certification of safety-critical software.

SmartTies is built on top of the pure document-management system DocTip, which is entirely parametric in the document type and change impact analysis rule systems, and extends it by specific document types, impact analysis rules systems, support for the development and certification workflow, as well as a web-based front-end and mediators converting between the document formats edited by the user and their internal, semantics-oriented representation. In the following, we will not explicitly distinguish between SmartTies and DocTip.

The paper is organised as follows: In Sec. 2 we introduce all the documents, relationships and consistency properties occurring in a software development process regulated by the IEC 61508 and required by a certification authority like the German TÜV. In Sec. 3 we present the different document types and relationships in order to analyse the properties of the whole document collection. Sec. 4 discusses the principles of document-type specific difference analysis and change impact propagation and Sec. 5 presents the structures, relationships and properties maintained in SmartTies as well as the supported workflow.

2 Developing Safety Critical Systems

Our running example here is the development of a system that calculates a *safety zone* for a moving, autonomous robot, thus safeguarding the robot against collisions with static obstacles. It is a very much simplified version of an actual development in the SAMS project [15] which was certified as conforming to IEC 61508 by the German TÜV. The documents occurring in this example are representative of a typical medium-sized certification effort.

Document-type specific structure. Table 1 shows the document types occurring in our example. We mainly have documents in OOXML (Office Open XML, MS-Word's native format) and C source code. All documents have an internal

Document type	Content	Structure	Format
Concept paper	Describe fundamental concepts of the system	Prose	OOXML
FTA	Fault Tree Analysis, models combinations of fault events leading to a safety failure	Table	OOXML
FMEA	Software Failure Modes and Effects Analysis, describes possible causes of failure	Table	OOXML
SRS	Safety Requirement Specification, enumerates requirements that are necessary to guarantee system safety	Table	OOXML
Test plan	Enumerates all test cases, together with their current status	Table	XML
Test suites	Contains the test driver functions	Functions	C code
Implementation	The actual implementation	Functions	C code

Table 1. Document types occurring in a safety-critical development process, together with their inherent structure and document format.

structure that results from their designation and the formalisation prescribed by the certification process. The *concept paper* introduces the underlying physical models for computing movement and braking of a vehicle that are used as given assumptions in the software design process. It consists of prose text possibly containing images and mathematical formulae. The *fault tree analysis* (FTA) decomposes the undesired event of a collision with an obstacle down to low-level fault events. The *failure mode and effects analysis* (FMEA) starts from possible failures and analyses how they may contribute to a failure of the safety function. The *safety requirement specification* (SRS) is an enumeration of functional requirements ensuring the safety of the vehicle based on the aforementioned physical models. All these documents are OOXML documents and have document type specific structure and content: a row in a table of the fault tree analysis document describes one undesired event, and a row in the FMEA describes a single failure mode, while a row in a table of an SRS document describes a safety requirement. A table in a concept paper, however, is simply a table without further document type specific semantics. The *test plan* consists of all the test cases, stored in plain XML and edited over the web front-end; *implementation* and *test suites* are MISRA-C source files, structured by the underlying programming language (here, function definitions and declarations).

Document graph. The structure gives rise to relationships within and between documents. Each basic semantic entity, such as safety requirements, fault events, failure modes, test cases, or functions, can be linked to others. This resulting graph structure, visualised exemplarily in Fig. 1, must satisfy a number of properties, which encode the restrictions on the development process prescribed by the certification standard.

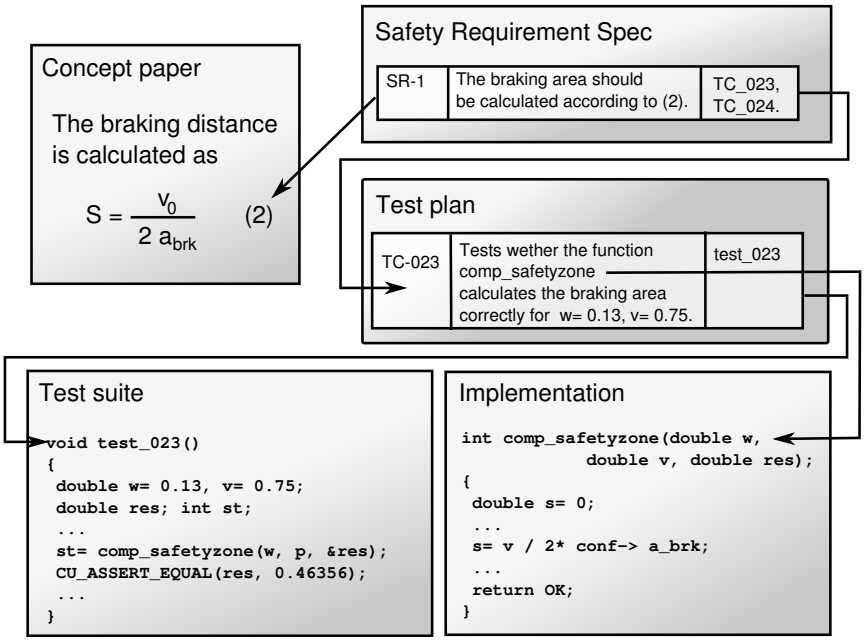

Fig. 1. Example document graph

Document collection properties. As a simple property, identifiers must be unique throughout the whole document collection. Further, each leaf fault event must reference at least one existing requirement. A requirement must either be decomposed into other requirements, or reference an existing function implementing the requirement and an existing test case in the test plan. Each test case must reference the function it is testing and the test driver function implementing the test. For sanity reasons, each test case must serve a purpose, so it must either be directly referenced from a requirement or it must be a precondition of at least one other test case. As a last example, safety requirements are the outcome of a hazard analysis documented in an FTA or FMEA, and we specify that each safety requirement has to be referenced by a fault event or failure mode. Though some automatic testing of properties exists for specific documents, checking all properties is typically done manually and automation is highly desirable. In particular, while we can check the presence of a link automatically, we cannot check that it is justified — we cannot deduce that a test case really tests the desired property. While we can assume that initially manual reasoning and review will be sufficient (if performed and documented properly), it is essential that when changes occur we can pinpoint their effects in terms of the manual reviews necessary.

2. Primäre Sicherheitsanforderungen

SR-1	Das berechnete Schutzfeld muss die gesamte beim Bremsen bis zum Stillstand wie durch das Bremsmodell beschrieben überstrichene Fläche überdecken.	IMPL-compute_safetyzone, TC-test_safetyzone, Testkonzept
SR-2	Das berechnete Schutzfeld muss eine Latenzzeit von ΔT beinhalten, in der das Fahrzeug mit unveränderter Geschwindigkeit und Richtung weiterfährt.	SR-3, SR-4
SR-3	Die Latenzzeit muss die Zykluszeit T des Systems beinhalten.	IMPL-set_config, TC_test-latency_1
SR-4	Die Latenzzeit muss die Ansprechzeit T_{brk} der Bremsen beinhalten	IMPL-set_config, TC_test-latency_2
SR-5	(gestrichen)	
SR-6	Die Bremsverzögerung muss die Abnutzung der Bremsen durch einen geschwindigkeitsabhängigen Zuschlag berücksichtigen	IMPL-set_config, TC_test-brake
SR-7	Die Höchstgeschwindigkeit v_{max} darf nicht überschritten werden.	IMPL-compute_safetyzone, TC-test_vmax

Fig. 2. Excerpt of the Safety Requirements Specification (in German)

3 Document Management

Each version of a document arising during the development process represents the state of the development, documenting and justifying design decisions made at that particular point in time. In early software development methodologies these documents were developed sequentially (waterfall model [13]). While this has the advantage that design decisions once made never have to be reconsidered, and thus assumptions can never become invalid, the underlying premiss that development can be finished successfully with the first attempt has proven highly unrealistic. Therefore, recent methodologies (such as agile development [5]) advocate an intertwined approached resulting in a parallel evolution of numerous documents. In this approach, changes occur frequently, and system support is needed to ensure they do not break the development.

Thus, we need systems which can handle and maintain change. However, as demonstrated in Sect. 2, there are number of different document types and formats, all with different editing tools, accompanied by tools such as compilers, test frameworks which run test suites and analyse the result, or verification tools to analyse and prove formal specifications. To handle change in this setting uniformly, we have developed a document broker called DocTip[1] that maintains and propagates changes and advances of individual documents to related documents. The general idea is that DocTip is notified about changes in documents made in the individual editing or analysis tools, computes their effects on other documents and initiates the necessary changes in the affected documents. DocTip is generic with respect to the document types supported and provides generic mechanisms to add new document types to the system [1, 3].

Generic Representation of Documents. We use XML as a common metalanguage to represent explicitly the structure of documents that is intrinsic to their

[1] http://www.dfki.de/cps/projects/doctip

```
<Document>
...
<srs>
<csrs component="Primäre Sicherheitsanforderungen">
  <reqspec>
    <reqid name="SR-1"/>
    <description>
      <paragraph>
        <text> Das berechnete Schutzfeld muss die gesamte beim Bremsen bis zum Stillstand
               wie durch das Bremsmodell beschrieben überstrichene Fläche überdecken.
        </text>
      </paragraph>
    </description>
    <measures>
      <paragraph> <ref kind="function" name="IMPL-compute_safetyzone"/> </paragraph>
      <paragraph> <ref kind="testcase" name="TC-test_safetyzone"/> </paragraph>
      <paragraph> <ref kind="label" docid="DOK-K-1" name="TestSafetyzone"/> </paragraph>
    </measures>
  </reqspec>
  <reqspec>
    <reqid name="SR-2"/>
    <description><paragraph><text>Das berechnete Schutzfeld muss eine Latenzzeit von </text>
      <formula style="inline">...</formula>
      <text> beinhalten, in der das Fahrzeug mit unveränderter Geschwindigkeit
             und Richtung weiterfährt.</text></paragraph>
    </description>
    <measures><paragraph><ref kind="requirement" name="SR-3"/> <text>, </text>
     <ref kind="requirement" name="SR-4"/></paragraph></measures>
  </reqspec>
        ...
</csrs>
</srs>
</Document>
```

Fig. 3. Corresponding XML version of the excerpt of the Safety Requirement Table

individual types. For instance, consider the safety requirement specifications. While written and edited in MS-Word, DocTip maintains an XML representation that explicitly segments the document in tables of safety requirements and their relations to implementation and environment descriptions. Document type specific parsers encode documents in XML and thus enable DocTip to maintain them but also decode modified XML versions back to the original document language. In SmartTies we developed encoders and decoders for the individual document types that are, for instance, used by MS-Word (which uses a different, richer layout information, but provides less content structure). The corresponding document type specific XML languages provide the structuring mechanisms for both, the generic outline of OOXML documents and the (partial) knowledge about the semantics of the individual document parts. Depending on the degree of natural language understanding and of syntactical restrictions by the document type (e.g. by using domain specific languages), we obtain a more shallow or deep XML encoding of informally written documents containing more or less chunks of non-parseable document fragments. As an example, consider Fig. 2 showing the original document as presented by MS-Word, and Fig. 3 the representing XML document making the implicit structure explicit.

Generic Document Analysis. The key idea to design change impact analysis (CIA) for informal documents is the *explicit semantics method* which represents

both the syntax parts (i.e., the documents) and the intentional semantics contained in the documents in a single, typed hyper-graph (see [4] for details). Document-type specific graph rewriting rules are used to extract the intentional semantics of documents and the extracted semantic entities are linked to their syntax source, i.e. their *origin*. The semantic graph is then analyzed to determine and propagate the impact of changes through the semantic graph, which are then projected backwards along the origin links to the syntactic nodes of the graph. A corresponding impact annotation for the syntactic part of the documents is then generated.

Generic Difference Analysis. Changes made to documents are recognised by analysing the differences between the different versions of the corresponding XML documents. Encoding all sorts of documents into different XML-based languages allows us to make use of XML-based tree-difference algorithms to compute differences between different versions of a document and represent the changes in terms of a uniform language and protocol (XML update, [4]). While we use a uniform XML diff algorithm to analyze differences of documents, this algorithm is adjusted to the individual document types by defining individual equivalence relations for each of them. These equivalence relations are used to determine which subtrees in two documents are similar and thus should be related to each other. This allows the diff algorithm to abstract from syntactical presentation issues that would otherwise prohibit the matching of related document parts. Equivalence relations are defined in terms of XML elements, attributes and subelements which identify corresponding XML subtrees.

DocTip relies on the XML update protocol to integrate changes obtained from the user interfaces or supporting analysis systems. Any change reported to DocTip is analyzed by the change management, which computes the impacts of these changes on other parts of a document or even in other documents. The propagated impacts are included in the documents maintained by DocTip, and passed along to the affected user interfaces and support systems.

In general, adding a new document type to DocTip involves the definition of the following:

- an XML language by an *XML schema* S and an additional predicate P to enforce properties of a document that are not covered by schema definitions. A document D is of type S iff D satisfies the scheme S. It is *admissible* with respect to S, P iff D satisfies S and $P(D)$ holds.
- an invertible *extraction function* ω to extract the XML representation from the actual syntax A of the document D, and to generate the actual syntax from the

$$\mathcal{A} \xleftrightarrow{\omega} \mathcal{D} \xleftrightarrow{\simeq} \mathcal{T} \underset{\pi}{\overset{\varphi}{\rightleftarrows}} \mathbf{Ker} \xhookrightarrow{\rho} \mathbf{Mod}$$

Fig. 4. Document type specific analysis: $\mathcal{A}$ are the documents of that type in their actual syntax, $\mathcal{D}$ the XML sublanguage for those documents, $\mathcal{T}$ the corresponding text-graphs, **Ker** the model kernels, and **Mod** the model graphs.

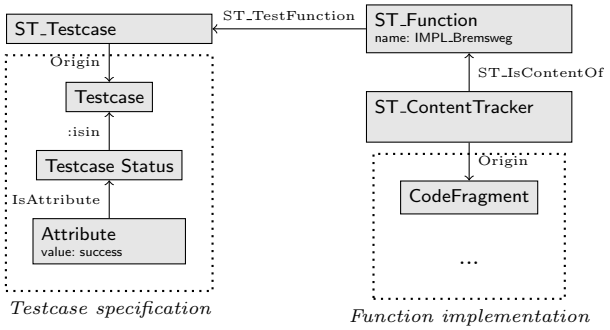

Fig. 5. Examples for text graph, model kernel and full model from the safety-critical software development domain

XML representation. The extraction process then works as follows: Given A, check whether $\omega(A)$ is admissible. If so, a *text graph* is computed. The graph structure correlates to the (parse) tree representation of XML documents. Hence, there is a one-to-one relationship between an XML document and its corresponding text graph, i.e. we can construct the XML document from its text graph and vice versa. A text graph is admissible iff its corresponding XML document is admissible. The subtrees in dotted boxes in Fig. 5 show parts of the text graphs from a testcase specification and an implementation.

- a document type specific *ontology* that describes the semantic concepts and their relationships.
- an *abstraction function* φ that computes the *model kernel* from the text graph. In contrast to the text graph, the model graph operates on semantic entities and their relationships, i.e., it consists of nodes and edges that correspond to the concepts and relationships that are defined in the corresponding ontology. The model kernel has the property that each of its entities is linked to a fragment of the text document, i.e. a node/link in the text graph, that caused its generation. E.g. a safety requirement node in the model kernel graph is linked to the corresponding text (i.e. the corresponding subtree in the XML description, representing the row in a table) defining it. Or the test case node in the model kernel graph is linked to the corresponding textual description, and the source code node in the model kernel graph is linked to the original source code (see Fig. 5). The idea is that the text graph will generate a model kernel which is expanded by semantic analysis to a fully fledged model graph.
- a document type specific *propagation function* ρ that computes the *model graph* by adding new nodes and edges to the model kernel (also from the corresponding ontology), representing derived information. E.g., in Fig. 5 the test case node in the model kernel is related to the tested source code node.
- a document type specific *projection function* π that maps derived information back to the text graph such that it can be presented to the user via ω.
- a document type specific equivalence model $\equiv$ to be able to compute the difference between two documents D and D' of the same type.

Fig. 4 summarises the functions that need to be provided for each document type, as well as their relations. The typical workflow is as follows: (1) Extract the XML representation using ω, (2) generate the text graph, (3) generate the initial model using φ, (4) compute the enriched model using ρ, (5) projecting the changes back using π, and (6) propagating the information back to the user document using ω^{-1}.

4 Change Management

In [2, 11] we presented tools to maintain structured specification and verification work in order to minimise the amount of proofs to be redone when modifying a specification. This idea is now extended from theories to heterogeneous document collections and from provers to arbitrary semantic analysis tools: we propagate the syntactical changes observed by the XML diff algorithm towards a change in the semantics and analyse these changes with respect to the deduced or user-postulated properties. In the following we will elaborate in more detail.

As explained in Sec. 3, each document A gives rise to an XML document $\omega(A)$ which induces a text graph T. Changes $A \to A'$ in the document thus result in changes $T \to T'$ of the corresponding text graph, which in turn cause changes $\varphi(T) \to \varphi(T')$ in the model kerneland therefore also changes in the preconditions of derived entities, rendering parts of the old model graph invalid but also potentially enabling the deduction of new entities.

Since we are interested in the development process of documents, it is crucial to encode explicitly what information changed from one version to another. This is because stateful information, e.g., the result of executing a test case, might invalidate due to a change, e.g., a change of the source code of the function that is tested. Therefore, recomputing the model graph from scratch is not an option, as it would not give us information about the changed parts, and therefore restrict our approach to stateless properties.

Our solution consists in specifying graph rewrite rules that adapt a given model graph based on the result of the difference analysis of the text graphs. Thus, applying the rules propagates the differences Δ_T to the kernel, such that they are explicitly represented in Δ_{Ker} (c.f. Fig. 6). Finally, the analysis function ρ is invoked in Δ_{Ker} to change derived properties in the model graph.

The transformation is successful if we reach a model graph that is consistent with the new model kernel and incorporates the same level of analysis as the old model graph. We define consistency by specifying the set of all consistent model graphs by providing a predicate P_{mod}. $P_{mod}(D)$ is true iff D is an element of this set. P_{mod} is invariant with respect to the insertion of derived knowledge, i.e. starting with a consistent model kernel the model graphs that are derived step by step by applying transformation rules (assuming an empty old model graph) will always stay consistent. Typically, P_{mod} is provided by a set of consistency rules defining (sub)graph properties that each model graph has to satisfy.

Adapting an old model graph to a changed model kernel, we have to adjust, delete or insert derived entities in the old graph to match the consistency rules

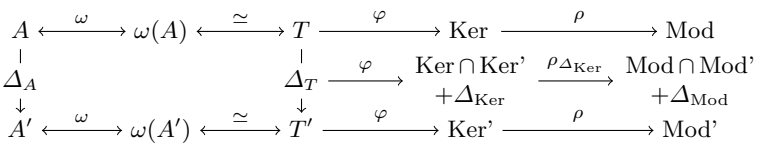

Fig. 6. Change Management: Changing a document A to A' induces changes in the text graph T, in the model kernel Ker, and in the model graph Mod. The differences Δ_T of the text graph are propagated to determine Δ_{Ker} and finally analyzed to derive the changes of the model graph.

together with the new kernel graph. This transformation process will start at differences between old and new kernel graph and will ripple along the lines of analysis of the old model graph computing implicitly the differences between old and new model subgraphs. This process obviously stops when there is no way to adapt the lines of reasoning appropriately without violating the consistency rules. As usual there are two ways to resolve such a conflict. First, we can drop the further adoption of the old model graph (i.e. throwing away knowledge about old bits that have been changed in the meanwhile). Second, we can speculate about necessary changes in the already computed model (sub)graph in order to satisfy the violated consistency rule and to propagate these required changes back to the model kernel (and further to the text graph). Which way we proceed depends on the character of the violated consistency rule.

Change Management in the Small. The graph transformation process is implemented with the help of a graph rewriting tool GrGen [7], which operates on typed and directed multi-graphs with multiple inheritance on node and edge types. In addition these types can be equipped with typed attributes and connection assertions to formulate restrictions on graphs.

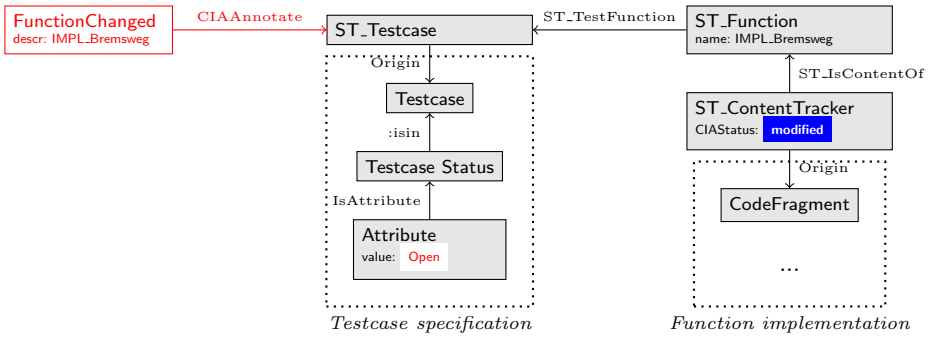

Fig. 7. Excerpt of the document graph and its changes

For example, consider the relationship between the code of IMPL_Bremsweg and the corresponding test cases that are used to validate the implementation. Changing the implementation, the corresponding tests specified in the test plan have to be redone. Fig. 7 presents the part of the model graph concerning the relation between the implementation of IMPL_Bremsweg and the test case specification. The model kernels of implementation and test cases are indicated by dotted lines. The model graph connects both kernels making the relation between both documents explicit (linking test case and code fragment via ST_Testcase, ST_Function and ST_ContentTracker).

Now suppose the implementation of IMPL_Bremsweg is changed. Comparing the XML versions of old and new version with the help of the XML-diff algorithm SmartTies localises the changes and adds both old and new version of the implementation of IMPL_Bremsweg into the model graph. The propagation of such a change is done with the help of GrGen graph rewrite rules. Since we are not interested in the details of the changes here, the GrGen rewrite rules will simply annotate the new implementation as changed by setting the CIA_Status attribute of ST_ContentTracker to "modified" and removing the old version from the model graph. In a second propagation phase this local change has to be propagated to the entire development using GrGen rewrite rules.

In general, GrGen rules specify rewrite rules on graphs allowing for pattern, replace and modify specification. A pattern matcher performs plain isomorphic subgraph matching as well as homomorphic matching for selectable sets of nodes and edges. Fig. 8 shows the rule used to propagate the modification of the implementation to the status of the tests. The block between iterated and modify constitutes the pattern of subgraphs on which the rule is applicable. Furthermore, the matches can be restricted by arithmetic and logical conditions on attributes and types, in our example we are only interested in changed implementation nodes, i.e `if { c.status == CIAStatus::modified; }`. Applying this rule to the subgraph printed in black in Fig. 7 results in the red additions: a node FunctionChanged is added to the graph and linked to ST_Testcase. Additionally, the value in the node Attribute is changed to "open", indicating the necessary rerun of the test cases. In a third phase the impacts of the change propagation to individual document parts are computed. Either they are automatically adapted

```
rule resetTestsWithChangedFunction {
 iterated {
   stc:ST_Testcase <-:ST_TestcaseFunction- stcused:ST_Function;
   stcused <-I:ST_IsContentOf- c:ST_ContentTracker;
   if { c.status == CIAStatus::modified; }
   stc -:Origin-> tc2:testcase;
   tc2 <-:isin- status:testcasestatus <-:IsAttribute- statusattr:Attribute;
   modify {
    stc <-:CIAAnnotate- a:ST_FunctionChanged;
    eval { a.description = a.description + stcused.name;
           statusattr.value = "open"; } }
   }
 modify {}
}
```

Fig. 8. Graph Transformation Rule to Actualize Tests

or if this is impossible (because, e.g., manual interaction is required) comments on necessary changes are added (e.g. as comments) to the document (cf. Fig. 10 for such annotations within the Safety Requirement Specification, SRS).

Change Management in the Large. In the following we sketch a typical scenario illustrating the cascade of changes during a development. In our running example, suppose a prototype of the system has been developed, comprising a concept paper, which contains the formula to calculate the braking distance, an FTA and an SRS which state *inter alia* that the braking distance must be calculated according to this formula, an implementation of the calculation of the safety zone, and test cases which check correctness of the calculation for various inputs.

The prototype is presented for internal review to the quality assurance department, and sure enough there is an error in the actual formula calculating the braking distance (it was $s = \frac{v_0}{2a_{brk}}$, and should have been $s = \frac{v_0^2}{2a_{brk}}$). This causes a series of corrections which ripple down the development graph (see Fig. 1):

1. The formula is corrected, and an analysis is triggered. Because there is a reference link to the formula from safety requirement SR-1, the correction in the formula will flag up an annotation in the safety requirement specification at SR-1 to check this event or requirement, respectively. Because we do not deal with the semantics of the formula, we cannot deduce what changes need to be made, but we can ask the specifier to recheck that SR-1 and its handling are still valid.
2. The specifier discovers that SR-1 as written is still valid, because they reference the formula and do not copy it verbatim, but SR-1 references test cases TC-023 and others. Test case TC-023 refers to rest function `test_023`. This test function is now wrong (or rather, the reference link is wrong), because the test data are calculated using the old (wrong) formula. The test functions are corrected, and another document analysis is run. This will invalidate the test results, as the test functions are now newer than the results.
3. The tests are re-run, and the their results uploaded into SmartTies. The changed tests covering SR-1 now fail, because the implementation in function `comp_safetyzone` uses the old formula.
4. The function `comp_safetyzone` is adapted, and assuming this is done in the correct way, the tests will now succeed again. A final document analysis asserts everything is consistent again, and we can re-present the documents for the next internal review.

Of course, in an example as small as this, a circumspect developer might make all changes in one go, but in larger developments, this type of support rippling small changes along the dependencies is the key to handling changes efficiently. Also, the initial error may have been rather obvious, but it is typical of a class errors which occur quite often but have wide-ranging consequences on the development process, namely modelling assumptions that do not quite hold in the real world (normally more subtle).

5 Document Semantics and Implementation

SmartTies supports the documents enumerated Table 1 which occur in the development of safety-critical software in a certification context. We have defined XML schemata which encode the semantic structure described rather straightforwardly. For these documents, SmartTies provides extraction functions ω as follows:

- For concept papers, FMEAs, FTAs and SRSs, the structured content is extracted by functions which parse OOXML;
- The source code is parsed by the frontend of the SAMS verification framework, and split into a sequence of external declarations;
- The test plan is kept as an XML document, and edited through the web interface.

The consistency checks and change propagation rules have been implemented using 49 graph rewrite rules and 66 graph test patterns.

As tools, SmartTies uses MS-Word for editing the informal text documents, an IDE of the user's choice for the source code, CUnit as the unit test framework (with a simple parser extracting the test results from the log file and inserting them into the test plan), and Subversion as the configuration management and version control backend.

The system architecture is web-based: the SmartTies server allows the user to upload or download documents, trigger the document analysis, and commit and update from a Subversion repository. The user accesses the system by two means: firstly, a plug-in for MS-Word allows to download and upload directly from within MS-Word, and secondly, a web interface allows to download and upload other documents, gives an overview over the current development status, and allows to start internal and external reviewing (Fig. 9). When the user triggers a document analysis, impacts in Word documents are reflected back to the user by *annotations* which show up in MS-Word as comments (see Fig. 10). This allows a seamless workflow within MS-Word.

The workflow is further supported by a document status cycling through phases from *in progress* during development to *approved* after a successful external review; SmartTies keeps track of the status, makes sure changes to it are properly documented by review reports, and versions the documents appropriately. The review process is supplemented by a simple ticketing system, which allows reviewers (in particular external) to register a list of open question which the system developers have to account for.

6 Conclusion

This paper presented an application of the generic DocTip-methodology for maintaining heterogeneous document collections in the area of safety-critical systems. While the DocTip engine is generic with respect to document types and operates purely on documents written in XML, SmartTies provides the necessary encodings of the application-depending document types in XML and the

Fig. 9. The SmartTies web interface.

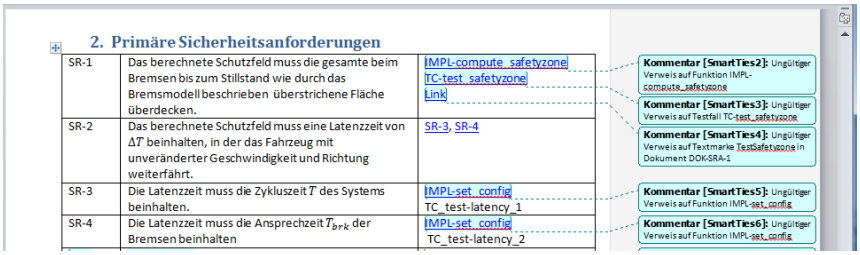

Fig. 10. Annotated Primary Safety Requirement Table (in German)

graph rewriting rules to propagate local changes in one document to the entire document collection. This allows for flexible development environments in which a user can provide or assemble specifications and corresponding propagation rules for their individually used document types. As a use case we applied SmartTies for a development of a small project with five MS word documents (concept paper, test concept description, FTA, SRS, and a user manual), a test plan with 40 test cases, ca. 650 loc CUnit test suites, and 430 loc implementation in C. We were able to successfully model the consistency rules and uses cases from Sect. 2 and Sect. 4 in our system. The text graph for the whole collection consisted of about 15000 nodes and the model graph of about 900 objects and 1500 relations. The analysis of a change using the graph rewriting rules consists of about 900-1000 graph rewriting rules and takes about 8.7s on an 2.8 GHz Intel Core i7 with 4GB RAM.

Up to now, instantiating the DocTip framework for a specific setting such as SmartTies has been laborious work, especially when formalising the impact analysis in terms of graph rewriting rules. However, we are working on general patterns for such analysis rules that will simplify this process significantly.

References

1. S. Autexier, C. David, D. Dietrich, M. Kohlhase, and V. Zholudev. Workflows for the management of change in science, technologies, engineering and mathematics. In *Conferences on Intelligent Computer Mathematics (CICM-11)*, 2011.
2. S. Autexier and D. Hutter. Formal software development in MAYA. In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning*. Springer, LNCS 2605, 2005.
3. S. Autexier and C. Lüth. Adding change impact analysis to the formal verification of c programs. In D. Méry and S. Merz, editors, *Proc. 8th International Conference on Integrated Formal Methods (iFM'10)*, LNCS. Springer, 2010.
4. S. Autexier and N. Müller. Semantics-based change impact analysis for heterogeneous collections of documents. In M. Gormish and R. Ingold, editors, *Proc. 10th ACM Symposium on Document Engineering (DocEng2010)*, 2010.
5. K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10), 1999.
6. L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka. Automated impact analysis of UML models. *Journal of Systems and Software*, 79(3):339–352, 2006.
7. R. Geiß, G. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Third International Conference on Graph Transformation (ICGT 2006)*. Springer, LNCS 4178, 2006.
8. D. Hutter. Semantic management of heterogeneous documents. In *Proc. Mexican International Conference on Artificial Intelligence*. Springer, LNAI 5845, 2009.
9. IBM. Rational DOORS. http://www-01.ibm.com/software/awdtools/doors/.
10. IEC. *IEC 61508 – Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC, Geneva, Switzerland, 2000.
11. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1–2):114–145, 2006.
12. C. Reichmann. PREEVision - bridging the gap between electrical/electronic and mechanical areas. *Automobile Konstruktion*, 1:1–4, 2011.
13. W. Royce. Managing the development of large software systems: Concepts and techniques. In *ICSE*, pages 328–339, 1987.
14. RTCA/DO-178B. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., Washington, D.C. 20036, 1992.
15. H. Täubig, U. Frese, C. Hertzberg, C. Lüth, S. Mohr, E. Vorobev, and D. Walter. Guaranteeing functional safety: design for provability and computer-aided verification. *Autonomous Robots*, 32(3):303–331, April 2012.