

Adding Change Impact Analysis to the Formal Verification of C Programs

Serge Autexier and Christoph Lüth*

Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI),
Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
`serge.autexier@dfki.de`, `christoph.lueth@dfki.de`

Abstract. Handling changes to programs and specifications efficiently is a particular challenge in formal software verification. Change impact analysis is an approach to this challenge where the effects of changes made to a document (such as a program or specification) are described in terms of rules on a semantic representation of the document. This allows to describe and delimit the effects of syntactic changes semantically. This paper presents an application of generic change impact analysis to formal software verification, using the GMoC and SAMS tools. We adapt the GMoC tool for generic change impact analysis to the SAMS verification framework for the formal verification of C programs, and show how a few simple rules are sufficient to capture the essence of change management.

1 Introduction

Software verification has come of age, and a lot of viable approaches to verifying the correctness of an implementation with respect to a given specification exist. However, the real challenge in software verification is to cope with changes — real software is never finished, the requirements may change, the implementation may change, or the underlying hardware may change (particularly in the embedded domain, where hardware is a major cost factor). Here, many existing approaches show weaknesses; it is not untypical to handle changes by rerunning the verification and see where it fails (though there exist more sophisticated approaches [1, 5], cf. Sec. 5).

To handle changes efficiently, a verification methodology which supports a notion of modularity is needed, together with the appropriate tool support which allows to efficiently delimit the impact of changes made to source code or specifications, thus leveraging the benefits of modularity. This is known as *change impact analysis*, and this paper describes how to adapt it to formal software verification. We make use of a change impact analysis tool for collections of documents developed by the first author, which supports document-type specific, rule-based semantic annotation and change propagation, and apply it to a framework for formal verification of C programs developed by the second author

* Research supported by BMBF under research grant 01 IW 07002 *FormalSafe*, and DFG under research grant Hu737/3-1 *OMoC*.

and others, which is based on annotating C programs with specifications. Our contribution here is to demonstrate that a generic change impact analysis tool can be adapted quickly to obtain a working solution for management of change in formal verification. We posit that separating change impact analysis from the actual software verification is useful, as it is a sensible separation of concerns and allows the impact analysis to be adapted and reused with other tools. This is a typical situation in verification, where often different tools are used for different aspects of the verification (e.g. abstract interpretation to check safety properties, and a theorem prover for functional correctness).

This paper is structured as follows: we first describe the two frameworks, the generic change impact analysis tool GMoC in Sec. 2 and the SAMS verification framework in Sec. 3, and show how to apply the former to the latter in Sec. 4.

2 Generic Semantic Change Impact Analysis

The motivation to develop a framework for generic semantic change impact is that an overwhelming amount of documents of different types are produced every day, which are rarely isolated artifacts but rather related to other kinds of documents. Examples of documents are filled and signed forms, research reports, or artifacts of the development process, such as requirements, documentation, and in particular specifications and program source code as in our application. These documents evolve over time and there is a need to automate the impact of changes on other parts of documents as well as on related documents. In [4], we proposed the GMoC framework for semantic change impact analysis that embraces existing document types and allows for the declarative specification of semantic annotation and propagation rules inside and across documents of different types. We give a brief and necessarily incomplete overview over the framework here, and refer the reader to [4] for the technical details.

Document Graph Model. The framework assumes documents to be XML files, represented and enriched by semantic information in a single typed graph called the *document graph*. The framework is parametrised over the specific XML format and types for the nodes and vertices of the document graph. The document graph is divided into syntactic and semantic subgraphs; the latter is linked to those syntactic parts which induced them, i.e. their *origins*. As a result the graphs have interesting properties, which can be methodologically exploited for the semantic impact analysis: it may contain parts in the document subgraph for which there exists no semantic counter-part, which can be exploited during the analysis to distinguish added from old parts. Conversely, the semantic subgraph may contain parts which have no syntactic origin, that is they are dangling semantics. This can be exploited during the analysis to distinguish deleted parts of the semantics from preserved parts of the semantics. The information about added and deleted parts in the semantics subgraph is the basis for propagating the effect of changes throughout the semantic subgraph. In order to be able to distinguish semantically meaningful changes, we further define an *equivalence*

```

<guests>
  <person confirmed="true">
    <firstName>Serge</firstName>
    <lastName>Autexier</lastName>
    <email>serge.autexier@dfki.de
  </email>
  </person>
  <person confirmed="true">
    <firstName>Normen</firstName>
    <lastName>Müller</lastName>
    <email type="prv">n.mueller@gmail.com
  </email>
  </person></guests>
  <guests>
    <person confirmed="true">
      <firstName>Normen</firstName>
      <lastName>Müller</lastName>
      <email type="prv">n.mueller@gmail.com</email>
    </person>
    <person confirmed="false">
      <firstName>Serge</firstName>
      <lastName>Autexier</lastName>
      <birthday>1/23/45</birthday>
      <email type="bus">serge.autexier@dfki.de</email>
    </person></guests>

```

Fig. 1: Semantically Equivalent Guest Lists.

relation indicating when two syntactically different XML elements are to be considered equivalent.

Abstraction and Projection Rules. To relate the two subgraphs, we define abstraction rules, which construct the semantic graph representing the semantics of a document, and projection rules, which project semantic properties computed in the semantic graph into impact annotations for the syntactic documents. The rules are defined as graph rewriting rules operating on the document graph.

Change Propagation Rules. The actual change impact is described by propagation rules, which are defined on the semantic entities, taking into account which parts of the semantics have been added and which have been deleted.

Example 1. Consider for instance a wedding planning scenario, where two types of documents occur: The first document is the guest list and the second one the seating arrangement. Both are semi-structured documents in an XML format as shown in Fig. 1; the latter depends on the former with the condition of male and female guests being paired. The change impact we consider is if one guest cancels the invitation the respective change in the guest list ripples through to the seating arrangement breaking the consistency condition of guests being rotationally arranged by gender. The idea of the semantic entities is that the semantic entity for a specific guest remains the same, even if its syntactic structure changes. For instance, the semantic entity of guest *Serge* remains the same even though the subtree is changed by updating the *confirmed* status.

Change Impact Analysis. Change impact analysis starts with taking all documents under consideration, building the syntactic document graph from the XML input, and applying in order the abstraction, propagation, and projection rule sets exhaustively. This initial step semantically annotates a document collection, and can be used to define semantic checks on document collections.

For change impact analysis between two versions of document collections, we then analyse the differences between the source and target documents. In order to ignore syntactic changes which are semantically irrelevant, we use a semantic difference analysis algorithm [13] for XML documents which is parametric in

the equivalence relation of the documents. The result of the difference analysis is an edit script, which is a set of edit operations; the effect of the equivalence models is that in general the edit scripts are less intrusive, thus preserving those parts of the documents from which the semantic parts have been computed. The edit scripts are applied on the syntactic parts of the document graph. As a result now there exists in the document graph new syntactic objects, for which no semantic counterparts exist yet as well as there are semantic parts, which syntactic origins have been deleted. On the graph we execute again the semantic annotation step, which exploits the information on added and deleted parts to compute the impact of the syntactic changes, represented by impact annotations to the documents.

Example 1 (continued). To sharpen our intuition on equivalent XML fragments, let us assume the two `guests` records shown in Fig. 1. Our focus is on the identification of guests. A guest is represented by an unordered `person` with the addition of whether this is committed or cancelled (`confirmed` attribute). Furthermore, information such as first name (`firstName`), surname (`lastName`), and e-mail address is stored. For the latter the distinction is between private address (`prv`) and business address (`bus`) encoded within a (`type`) attribute whereas the type is defaulted to be private. Optionally the date of birth is represented in a `birthday` element. Over time, the entries in the guest list change. Thus, for example, in the bottom half of Fig. 1 the order of guests is permuted and for one `person` element the status of commitment and the e-mail address type changed. In addition, the date of birth has been registered. To identify guests, we are not interested in the status regarding commitment or cancellation and we do not care if we send the invitation to the business address or home address. Therefore, we define the primary key of a guest as a combination of first name, last name and e-mail address, i.e. when comparing two `person` elements, changes in confirmed status or e-mail address type are negligible as well as the existence of birthday information. Existing XML differencing tools, however, would even with an adequate normalisation consider the two guest records to be different. The change of the confirmation status makes the two XML fragments unequal although regarding our primary key definition for `person` elements those two are equal.

Realisation. The semantics-based analysis framework has been realised in the prototype tool GMoC [2] that annotates and analyses the change impacts on collections of XML-documents. The abstraction, propagation and projection rules are graph transformation rules of the graph rewriting tool GrGen [9], which has a declarative syntax to specify typed graphs in so-called *GrGen graph models* as well as GrGen graph rewriting rules and strategies. The GMoC-tool is thus parametrised over a *document model* which contains (i) the GrGen graph model specifying the types of nodes and edges for the syntactic and semantic subgraphs for these documents, as well as the equivalence model for these documents; (ii) the abstraction and projection rules for these documents in GrGen's rule syntax; and (iii) the propagation rules for these documents, written in the GrGen rule syntax as well.

3 Formal Verification of C Programs

In this section, we give a brief exposition of the SAMS verification framework in order to make the paper self-contained and show the basics on which the change management is built in Sec. 4. For technical details, we may refer the interested reader to [10]. The verification framework is based on *annotations*: C functions are annotated with specifications of their behaviour, and the theorem prover Isabelle is used to show that an implementation satisfies the specification annotated to it by breaking down annotations to proof obligations, which are proven interactively. Thus, in a typical verification workflow, we have the role of the implementer, who writes a function, and the verifier, who writes the proofs. Specifications are typically written jointly by implementer and verifier.

Language. We support a subset of the C language given by the MISRA programming guidelines [11]; supported features include a limited form of address arithmetic, arbitrary nesting of structures and arrays, function calls in expressions, and unlimited use of pointers and the address operator; unsupported are function pointers, unstructured control flow elements such as goto, break, and switch, and arbitrary side effects (in particular those where the order would be significant). Programs are deeply embedded into Isabelle and modelled by their abstract syntax, thus there are datatypes representing, *inter alia*, expressions, statements and declaration. (A front-end translates concrete into abstract syntax, see Fig. 3.) The abstract syntax is very close to the phrase structure grammar given in [7, Annex A].

Semantics. The foundation of the verification is a denotational semantics as found e.g. in [15], suitably extended to handle a real programming language, and formalised in Isabelle. It is deterministic and identifies all kinds of faults like invalid memory access, non-termination, or division by zero as complete failure. Specifications are semantically considered to be state predicates, as in a classical total Hoare calculus. A specification of a program p consists of a precondition P and a postcondition Q , both of which are state predicates, and we define that p satisfies this specification if its semantics maps each state satisfying P to one satisfying Q :

$$\Gamma \vdash_s [P] p [Q] \stackrel{def}{=} \forall S. P S \longrightarrow def(\llbracket p \rrbracket S) \wedge Q(\llbracket p \rrbracket S) \quad (1)$$

where Γ is the global environment containing variables and the specifications of all the other functions. We can then prove rules for each of the constructors of the abstract syntax as derived theorems. Special care has been taken to keep verification modular, such that each function can be verified separately.

Specification language. Programs are specified through annotations embedded in the source code in specially marked comments (beginning with `/*@`, as in JML or ACSL). This way, annotated programs can be processed by any compiler without modifications. Annotations can occur before function declarations, where they take the form of function specifications, and inside functions to denote loop invariants. A function specification consists of a precondition (`@requires`),

```

/*@
  @requires 0 <= w
            && w < sams_config.brkdist.measurements[0].v
            && brkconfig_OK(sams_config)
  @modifies \nothing
  @ensures 0 < \result
            && \result < sams_config.brkdist.length
            && sams_config.brkdist.measurements[\result-1].v > w
            && w >= sams_config.brkdist.measurements[\result].v
  @*/
Int32 bin_search_idx_v( Float32 w);

```

Fig. 2: Example specification: Given a velocity w , find the largest index i into the global array `sams_config.brkdist.measurements` such that `measurements[i].v` is larger than w . The assumption is that at least the first entry in `measurements` is larger than w , and that the array is ordered (this is specified in the predicate `brkconfig_OK`). The function has no side effects, as specified by the `@modifies \nothing` clause.

a postcondition (`@ensures`), and a modification set (`@modifies`). Fig. 2 gives an example. The state predicates are written in a first-order language into which Isabelle expressions can be embedded seamlessly using a quotation/antiquotation mechanism; the exact syntax can be found in [10]. In contrast to programs, specifications are embedded shallowly as Isabelle functions; there is no abstract syntax modelling specifications.

Tool chain and data flow. The upper part of Fig. 3 shows a graphical representation of the current data flow in the SAMS environment. We start with the annotated source code; it can either be compiled and run, or verified. For each source file, the frontend checks type correctness and compliance with the MISRA programming guidelines, and then generates a representation of the abstract syntax in Isabelle, and for each function in that source file, a stub of the correctness proof (if the file does not exist). The user then has to fill in the correctness proofs; users never look at or edit the program representation. The resulting proof scripts can be run in Isabelle; if Isabelle finishes successfully, we know the program satisfies its specification. Crucially, the verification is modular — the proofs can be completed independently and in any order.

Proving correctness. This is because correctness is proven in a modular fashion, for each function separately. The correctness proof starts with the goal *correct* Θf , which is unfolded to a statement of the form $\Theta \vdash_s [pre] \textit{body} [post]$, where *body* is the body of the function, and *pre* and *post* the pre- and postconditions, respectively. This goal is reduced by a tactic which repeatedly applies proof rules matching the constructors of the abstract syntax of the *body*, together with tactics to handle state simplification and framing. After all rules have been applied, we are left with *proof obligations* relating to the program domain, which

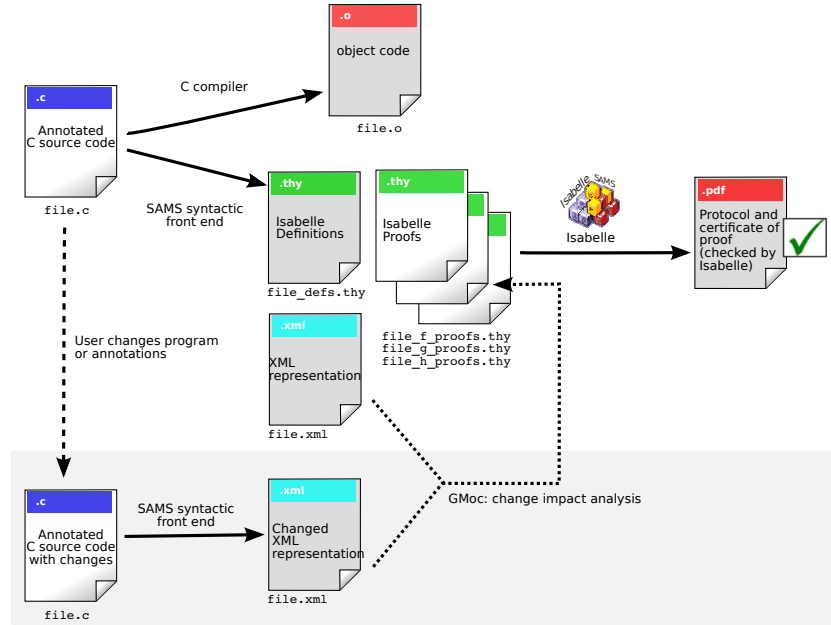


Fig. 3: Dataflow of the verification environment. Grey files are generated, white files are created and edited by the user. The lower part (highlighted in grey) is the added change impact analysis; given a change made by the user, the GMoC tool compares the two XML representations of original and changed copy, and calculates which proof scripts are affected by the change.

have to be proven interactively in Isabelle. These proofs make up the individual proofs scripts in Fig. 3.

The proof scripts can be large, and the whole framework including proof rules and tactics implementing the reduction of correctness assertions to proof obligations is even larger. However, the correctness of the whole verification process reduces to correctness of a very small trusted core, consisting of the denotational semantics of the programming language and the notion of satisfaction from equation (1); the rest is developed conservatively from that. One consequence of this is that when we implement change management as meta-rules, we can never produce wrong results (i.e. a result which erroneously states a program satisfies a specification), as Isabelle is always the final arbiter of correctness.

Change Management. The framework has been developed and used in the SAMS project to verify the correctness of an implementation of a safety function for autonomous vehicles and robots, which dynamically computes safety zones depending on the vehicle’s current speed. The implementation consists of about 6 kloc of annotated C source code, with around fifty functions. What happens now if we change one of these functions or its specification? In the current framework, any changes to the annotated source code require the front-end to be run

again, producing a new representation of the program. (The front-end will never overwrite proof scripts.) One then has to rerun the proof scripts, and see where they fail. For large programs, this is not very practical, as proof scripts can run from several minutes up to an hour. One might break down a source file into smaller parts containing only one function each, but besides not being desirable from a software engineering point of view, this does not help when we change the specification or declaration of a function, which lives in its header file, and is always read by other functions.

In practise, the change impact analysis formalised here is done manually — when a change is made, the verifier reruns those proof scripts which he suspects may break. But this is error prone, and hence we want to make use of change impact analysis techniques, and in particular the GMoC tool, to help us delimiting the effects of changes, pointing us to the proofs which we need to look at. This will extend the current tool chain as depicted in the lower part of Fig. 3.

4 Change Impact Analysis for the SAMS Environment

In order to adapt the generic change impact analysis to the specific SAMS setting, we need to instantiate the generic *document model* of Sec. 2 in three easy steps: (i) define the graph model representing the syntax and semantics of annotated C programs, together with the equivalence relation on the semantics, then (ii) define abstraction and projection rules relating syntax and semantics, and finally (iii) define the rules how changes in annotated programs propagate semantically. We also need to tool the SAMS front-end to generate output of the abstract syntax in the specified XML format, so it can be input to GMoC. All of this has only to be done once and for all, to set up the generic environment. This is because of the basic assumption of the GMoC tool that the underlying document model does not change during the document lifetime (which is the case here, as the syntax and semantics of annotated C programs is fixed).

We now describe the three steps in turn. As a running example we consider the safety function software for autonomous vehicles and robots mentioned in the previous section. In this software one source file contained the functions to compute the configuration of the safety function based on an approximation of the braking distance. It consists of three functions `bin_search_idx_v` called by the function `brkdist_straight` itself called by `compute_brkconfig`. The specification of `bin_search_idx_v` was given in Fig. 2, and the change we consider is that there had been an error in the postcondition saying `@ensures w > sams_config.brkdist.measurements[\result].v` (wrongly using `>` instead of `>=`), and this is now changed in the source code to result in the (correct) specification in Fig. 2. This changes the specification of `bin_search_idx_v`, and as a consequence the proofs of `bin_search_idx_v` and of the calling function `brkdist_straight` must be inspected again, but not the proof of any other function, such as `compute_brkconfig`. Detecting this automatically is the goal of the change impact analysis with GMoC.

```

enum CIAStatus {added,deleted,preserved}
node class CIANode extends GmocNode {status:CIAStatus = CIAStatus::added;}
abstract node class Symbol extends CIANode { name : string;}
node class Function extends Symbol {}
node class SemSpec extends CIANode { function : string;}
node class SemBody extends CIANode { function : string;}
edge class Origin extends GmocLink {}
edge class CIALink extends GmocLink { status : CIAStatus = CIAStatus::added;}
edge class Uses extends CIALink {}
edge class IsSpecOf extends CIALink {}
edge class IsBodyOf extends CIALink {}

```

Fig. 4: Semantic GrGen Graph Model for Annotated C Programs

```

equivspec annotatedC {
  element invariant {}
  unordered declaration {constituents = {InitDecl, StorageClassSpec}}
  element InitDecl {constituents = {IdtDecl, FunDecl}}
  element FunDecl {constituents = {IdtDecl}}
  element IdtDecl {constituents = {Identifier}}
  element Identifier {annotations = {id?}} ... }

```

Fig. 5: Part of the equivalence model for the XML syntax of annotated C files

4.1 Graph Model of the Documents

The *syntactic* document model is the abstract syntax of the annotated C programs, with nodes corresponding to the types of the non-terminals of the abstract syntax, and edges to the constructors. We use an XML representation of that abstract syntax, and let the SAMS front-end generate that XML representation. The *semantic* entities relevant for change impact analysis of the annotated C programs as sketched above are:

- functions of a specific name and for which we use the node type `Function`;
- the relationship which function calls which other functions directly and for which we have the `Uses` edges among `Function` nodes;
- specifications of functions, for which we use the nodes of type `SemSpec` which are linked to the respective `Function`-nodes by `IsSpecOf` edges; and
- bodies of functions, for which we use nodes of type `SemBody`, are linked to the respective `Function`-nodes by `IsBodyOf` edges.

Every semantic node has at most one link from a syntactic part, its `Origin`. Finally, all semantic nodes and edges have a `status` attribute indicating whether they are `added`, `deleted`, or `preserved`. Fig. 4 shows the GrGen graph model declaration to encode the intentional semantics of annotated C files.

Next we use the declarative syntax to specify the *equivalence model* for these files, an excerpt of which being shown in Fig. 5: first, the equivalence model specifies to ignore the filename and the file-positions contained as attributes. Thus, for most XML elements, the `annotations`-slot in the entry on the equiva-

<pre> rule detectNewFunDecls { attr : Attribute - : IsAttribute -> id : Identifier - : isin -> idt:IdtDecl - : isin -> fd:FunDecl - : isin -> d:FunDef; if { attr.name == "id";} negative { id - : Origin -> f:Function; if { f.name == attr.value; }} modify { id - : Origin -> f:Function; exec (findSpec(fd, f)); exec (findFctBody(d, f)); }} </pre>	<pre> rule detectExistingFunDecls { attr : Attribute - : IsAttribute -> id : Identifier - : isin -> idt:IdtDecl - : isin -> fd:FunDecl - : isin -> d:FunDef; if { attr.name == "id";} id - : Origin -> f:Function; if { f.status == CIAStatus::deleted;} if { f.name == attr.value;} modify { eval {f.status=CIAStatus::preserved;} exec (findSpec(fd, f)); exec (findFctBody(d, f)); }} </pre>
--	---

Fig. 6: Rules to detect new and existing functions

lence model does not contain these attribute names. Furthermore, it indicates that declarations are unordered elements and that two declarations are to be considered equal, if the children `InitDecl` and `StorageClassSpec` are equivalent. These equivalence relations specifications are also given and for `Identifiers` the recursion is over the value of the attribute `id`.

For the specifications `spec` and the bodies `FctBody` of functions no entries are specified. As a consequence, they are compared by deep tree equality in the semantic difference analysis and thus every change in these is detected as a replacement of the whole specification (respectively body) by a new version.

4.2 Abstraction and Projection Rules

The *abstraction rules* are sequences of GrGen graph rewriting rules that take the abstract syntax of annotated C programs and compute the semantic representation. In our case these are rules matching function declarations, bodies of functions and specifications of functions and — depending on whether a corresponding semantic entity exists or not — either adjust its status to `preserved` or add a new semantic entity which gets status `added`. Corresponding GrGen rules for these two cases are shown in Fig. 6. Both rules modify the graph structure as specified in the `modify` part and can invoke the call to other rules now concerned with matching the specification and body parts of found functions, which are defined analogously, except that they also add or update the `IsSpecOf` and `IsBodyOf` edges to the given `Function`-node `f` passed as an argument. Finally, there is one rule that establishes the function call relationship between functions by checking the occurrence of functions in function bodies.

Prior to the application of the abstraction rules, the status of all nodes and edges in the semantic graph are set to `deleted`. The adjustment of that information during the abstraction phase for `preserved` parts and setting the status of new semantic objects to `added` results in a semantic graph, where the status of

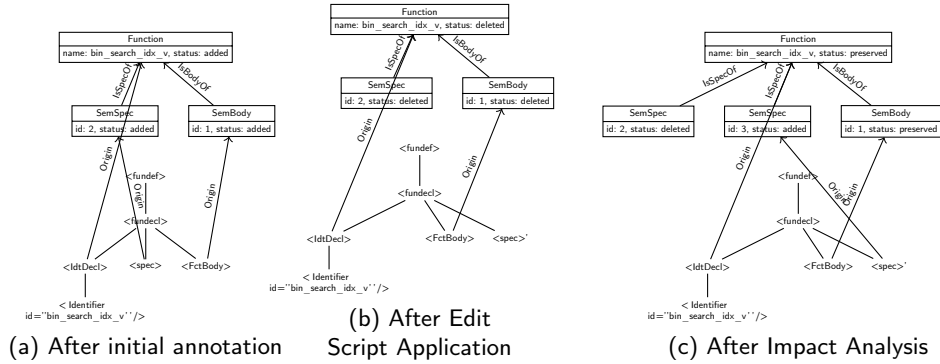


Fig. 7: Phases of the document graph during impact analysis

the nodes and edges indicates what happened to these objects. For instance, if the specification of a function f has been changed, then the function node f has one `lsSpecOf` edge to an old `SemSpec` node, which has status `deleted`, and one `lsSpecOf` edge to a new `SemSpec` node, which has status `added`.

Example 2. In our running example the abstraction rules detect the functions `bin_search_idx_v`, `brkdist_straight` and `compute_brkconfig` as well as their corresponding bodies and specifications (see Fig. 7(a) for the part for the function `bin_search_idx_v` only). Finally, they add the `Uses` edges indicating that `bin_search_idx_v` is called by `brkdist_straight`, which in turn is called by the function `compute_brkconfig` (all not shown in Fig. 7(a)).

The *projection rules* simply project the computed affect information into the syntactic documents by linking the origins of the semantic entities with respective `Impact` nodes. Since we have essentially two kinds of changes causing the re-inspection of proofs, we have two corresponding rules, of which we only present the specification change rule:

```

pattern markSpecCauseProofRequiresInspection (f:Function,cause:Function) {
  id: Identifier  - :Origin-> f;
  negative { if { f.status == CIStatus::deleted; }
             :fctSpecChangeMarked(f,cause); }
  modify { i:Impact-:affects-> id;
           eval { i.name = "ProofAffected"; i.value = "SpecChange "+cause.name;}}

```

The `Impact` nodes will be automatically serialised into an XML description of the impacts of the changes, reusing the name and value specified in the `impact` node and referencing the XML subtree corresponding to the linked node (e.g., `id:Identifier`) by its XPath in the document.

The final part of the projection phase is to actually delete those nodes and edges in the semantic graph still marked as `deleted`, which is some kind of garbage collection after completing the analysis — and before the next analysis phase starts after application of the syntactic changes on the documents.

4.3 Change Propagation Rules

In the SAMS verification environment, the verification is modular and each C function is verified separately. In this context, changes in the annotated source code affect proofs as follows:

- (**CIABodyChange**) If a function is modified but its specification not changed, then only the proof for this function needs to be checked again.
- (**CIASpecChange**) If the specification of a function f is changed, then the correctness proof of that function needs to be checked as well as the proofs of all functions that directly call f , because in these proofs the specification may have been used.

These are two simple rules that can be checked on the C files directly. In the following we describe how these checks have been automated using the GMoC-tool, presenting the formalisation of the latter in detail. First, we specify a pattern to recognise if the specification of a function has changed based on the fact that we have a function which has not status `deleted` and which has an `IsSpecOf` edge from an added `SemSpec`:

```

pattern fctSpecChanged(f:Function) {
  negative { if { f.status == CIAStatus::deleted; }}
  newss:SemSpec -:IsSpecOf-> f;
  if { newss.status == CIAStatus::added; }}

```

This pattern is used to detect functions which specifications have changed and mark these and all functions calling them as being affected by the change. This is done by the sub-rules `markSpecCauseProofRequiresInspection` and `markCallingFunctionsProofInspection` in the pattern below, where the keyword **iterated** indicates that the enclosed pattern must be applied exhaustively:

```

pattern propagateChangedSpec {
  iterated { f:Function;
    :fctSpecChanged(f);
    markself:markSpecCauseProofRequiresInspection(f,f);
    markothers:markCallingFunctionsProofInspection(f);
    modify { markself(); markothers(); }}
  modify {} }

```

A similar but simpler propagation rule exists to detect changed function bodies which marks only the function itself, but not the calling functions.

4.4 Change Impact Analysis

For the change impact analysis, the original annotated C programs are semantically annotated using the abstraction, propagation and projection rules from the previous section. Before analysing the changes caused by a new version of the annotated C programs, all impact nodes are deleted, because they encode the impacts of going from empty annotated C programs to the current versions. This is the equivalent to an adjustment of the baseline in systems like DOORS.

Next the differences to the new versions of the annotated C programs are analysed using the semantic difference analysis instantiated with the equivalence model for annotated C programs. This model is designed such that all type, constant and function declarations are determined by the name of the defined objects and any change in a specification or function body causes its entire replacement. Applying the resulting edit script thus deletes the node which was marked as the origin of a specification (resp. body, function, type or constant), and thus the corresponding node in the semantic graph becomes an orphan.

Example 2 (continued). For our example, the change in the specification of `bin_search_idx_v` from `>` to `>=` is detected by the semantic difference analysis and due to the way the equivalence model is defined, the edit script contains the edit operation to remove the entire specification and replace it with a new one. Applying that on the SDI graph deletes the old specification node in the document graph and adds a new specification to the document graph. Thus, in the semantic graph the `SemSpec` node for the function `bin_search_idx_v` now lacks an `Origin`-node. The shape of the graph now including deletion information on the semantic parts and the removed/added syntactic parts from the edit script is shown in Fig. 7(b).

On that graph the same abstraction, propagation and projection rules are applied again. All rules are change-aware in the sense that they take into account the status (`added`, `deleted`, or `preserved`) of existing nodes and edges in the semantic graph and adapt it to the new syntactic situation. As a result we obtain `Impact`-nodes marking those functions, which proofs must be inspected again according to the rules (`CIABodyChange`) and (`CIASpecChange`).

More specifically, for our example C files, the abstraction-phase introduces a new `SemSpec`, and hence the `Function`-node for `bin_search_idx_v` in the semantic graph now has one `IsSpecOf`-edge to the old `SemSpec`-node (i.e. with status `deleted`) and one to the `SemSpec`-node which has just been added (see Fig. 7(c)). This is exploited by the propagation rules to mark `bin_search_idx_v` and `brkdlist_straight` as those functions which proofs need inspection.

5 Conclusions, Related and Future Work

This paper has shown how change impact analysis can be used to handle changes in formal software verification. Our case study was to adapt the generic GMoC tool to the change impact analysis for annotated C programs in the SAMS verification framework, and demonstrate its usage with source code developed in the SAMS project. Currently, prototypes of both tools are available at their respective websites [2, 14]

Results. The case study has shown that the principles of the explicit semantic method underlying the GMoC framework indeed allow to add change impact analysis to an existing verification environment. Note that we have not shown all rules above, but only those of semantic importance. As far as the C language is concerned, further impact analysis is unlikely to be much of an improvement, as functions are the appropriate level of abstraction. One could consider analysing

the impact of changing type definitions, but because type checking takes place before the change impact analysis, this is covered by the rules described above: if we change the definition of a type, then in order to have the resulting code type-check, one will typically need to make changes in the function definitions and declarations using the type, which will in turn trigger the existing rules.

On a more technical note, the case study also showed that the current prototype implementation of GMoC does not scale well for large XML files, because interaction between GMoC, implemented in Java, and GrGen, based on .NET, is currently based on exchanging files. Thus, future work will consist of moving to an API based communication between GrGen and GMoC.

Related work. Other formal verification tools typically do not have much management of change. For example, two frameworks which are close to the SAMS framework considered here are Frama-C [8] and VCC [6], both of which use C source code annotated with specifications and correctness assertions, and both of which handle changes by rerunning the proofs. Theorem provers like Isabelle and Coq (which is used with Frama-C) have very coarse-grained change management at the level of files (i.e. if a source file changes, all definitions and proofs in that file and all other files using this, are invalidated and need to be rerun). Some formal methods tools, such as Rodin [1] and KIV [5], have sophisticated change management, which for these tools is more powerful than what a generic solution might achieve, but the separation advocated here has three distinct advantages which we believe outweigh the drawbacks: (i) it makes change impact analysis (CIA) reusable with other systems (e.g. the SAMS instantiation could be reused nearly as is with Frama-C); (ii) it allows experimentation with different CIA algorithms (amounting to changing the rules of the document model); and (iii) it allows development of the verification system to be separated from development of the change management, and in particular allows the use of third-party tools (such as Isabelle in our case) for verification. In previous work the first author co-developed the MAYA system [3] to maintain structured specifications based on development graphs and where change management support was integrated from the beginning. These experiences went into the development of GMoC which is currently also used to provide change impact analysis for the Hets tool [12] where change management was not included right from the beginning.

Outlook. The logical next step in this development would be change impact analysis for Isabelle theories, to allow a finer grained change management of the resulting theories. As opposed to the situation in C, fine-grained analysis in Isabelle makes sense, because in general proof scripts are much more interlinked than program source code, and because they take far longer time to process. However, for precisely this reason it requires a richer semantic model than C.

The change impact analysis could then be used for the Isabelle proof scripts occurring in the SAMS framework, for standalone Isabelle theories, or for Isabelle proof scripts used in other tools. This demonstrates that it is useful to keep change impact analysis separate from the actual tools, as it allows its reuse, both across different versions of the same tool (this is particularly relevant when tools are still under active development, which is often the case in academic

environment), or when combining different tools (a situation occurring quite frequently in software verification). Thus, we could have implemented the rules above straight into the frontend with not much effort, but that would still leave the necessity to handle changes for the Isabelle theories.

On a more speculative note, we would like to extend the change impact analysis to handle typical re-factoring operations (both for C, and even more speculative, for Isabelle), such as renaming a function or parameter (which is straightforward), or e.g. adding a field to a structure type t ; the latter should not impact any correctness proofs using t except those calling `sizeof` for t .

References

1. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Metha, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.
2. S. Autexier. The GMoC Tool for Generic Management of Change. Website at <http://www.informatik.uni-bremen.de/dfki-sks/omoc/gmoc.html>, 2010.
3. S. Autexier and D. Hutter. Formal software development in Maya. In *Festschrift in Honor of J. Siekmann*, LNAI 2605. Springer, 2005.
4. S. Autexier and N. Müller. Semantics-based change impact analysis for heterogeneous collections of documents. In *Proc. 10th ACM Symposium on Document Engineering (DocEng2010)*, UK, 2010.
5. M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*, LNCS 1783, pages 363–366. Springer, 2000.
6. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, LNCS 5674, pages 23–42. Springer, 2009.
7. Programming languages — C. ISO/IEC Standard 9899:1999(E), 1999. 2nd Edition.
8. Frama-C. Website at <http://frama-c.cea.fr/>, 2008.
9. R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Graph Transformations (ICGT 2006)*, LNCS 4178, pages 383–397. Springer, 2006.
10. C. Lüth and D. Walter. Certifiable specification and verification of C programs. In *Formal Methods (FM 2009)*, LNCS 5850, pages 419–434. Springer, 2009.
11. MISRA-C: 2004. Guidelines for the use of the C language in critical systems. MISRA Ltd., 2004.
12. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, LNCS 4424, pages 519–522. Springer, 2007.
13. N. Müller. *Change Management on Semi-Structured Documents*. PhD thesis, School of Engineering & Science, Jacobs University Bremen, 2010.
14. Safety Component for Autonomous Mobile Service Robots (SAMS). Website at <http://www.sams-project.org/>, 2010.
15. G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press, 1993.