Autom Softw Eng (2010) 17: 119–148
DOI 10.1007/s10515-010-0062-z

Generic patch inference

Jesper Andersen · Julia L. Lawall

Received: 24 January 2010 / Accepted: 1 February 2010 / Published online: 12 February 2010

Abstract A key issue in maintaining Linux device drivers is the need to keep them up to date with respect to evolutions in Linux internal libraries. Currently, there is little tool support for performing and documenting such changes.

In this paper we present a tool, spdiff, that identifies common changes made in a set of files and their updated versions, and extracts a generic patch performing those changes. Library developers can use our tool to extract a generic patch based on the result of manually updating a few typical driver files, and then apply this generic patch to other drivers. Driver developers can use it to extract an abstract representation of the set of changes that others have made.

Our experiments on recent changes in Linux show that the inferred generic patches are more concise than the corresponding patches found in commits to the Linux source tree while being safe with respect to the changes performed in the provided driver files.

Keywords Linux · Patches · Change detection

1 Introduction

In the case of open-source software, such as Linux, where the developers are widely distributed, it must be possible to exchange, distribute, and reason about source code changes. One common medium for such exchange is the patch (MacKenzie et al. 2003). When making a change in the source code, a developer makes a copy of the

J. Andersen · J.L. Lawall (✉)
Department of Computer Science (DIKU), University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark
e-mail: julia@diku.dk

J. Andersen
e-mail: jespera@diku.dk

code, modifies this copy, and then uses `diff` to create a file describing the line-by-line differences between the original code and the new version. He then distributes this file, known as a *patch*, to subsystem maintainers and mailing lists for discussion. Once the patch has been approved, other developers can apply it to their own copy of the code, to update it to the new version.

Patches have been undeniably useful in the development of Linux and other open-source systems. However, it has been found that they are not very well adapted for one kind of change, the *collateral evolution* (Padioleau et al. 2006). A collateral evolution is a change entailed by an evolution that affects the interface of a library, and comprises the modifications that are required to bring the library clients up to date with this evolution. Collateral evolutions range from simply replacing the name of a called library function to more complex changes that involve multiple parts of each affected file. Such changes may have to be replicated across an entire directory, subsystem implementation, or even across the entire source code. In the case of Linux, it has been shown that collateral evolutions particularly affect device drivers, where hundreds of files may depend on a single library (Padioleau et al. 2006).

The volume and repetitiveness of collateral evolutions strain the patch-based development model in two ways. First, the original developer has to make the changes in every file, which is tedious and error prone. Second, developers that need to read the resulting patch, either to check its correctness or to understand what it will do to their own code, may have to study hundreds of lines of patch code, which are typically all very similar, but which may contain some subtle differences. An alternative is provided by the transformation system Coccinelle, which raises the level of abstraction of patches to *semantic patches* (Padioleau et al. 2008). A semantic patch describes a change at the source code level, like an ordinary patch, but is applied in terms of the syntactic and semantic structure of the source language, rather than on a line-by-line basis. Semantic patches include only the code relevant to the change, can be abstracted over irrelevant subterms using metavariables, and are independent of the spacing and line breaks of the code to which they are applied. The level of abstraction of semantic patches furthermore implies that they can be applied to files not known to the original developer—in the case of Linux, the many drivers that are maintained outside the Linux source tree.

Despite the many advantages of semantic patches, it may not be reasonable to expect developers to simply drop the patch-based development model when performing collateral evolutions. For the developer who makes the collateral evolution, there can be a gap between the details of an evolution within a library and the collateral evolution it entails. Therefore, he may still find it natural to make the required changes by hand in a few typical files, to better understand the range and scope of the collateral evolution that is required. Furthermore, the standard patch application process is very simple, involving only replacing one line by another, which may increase confidence in the result. Thus, developers may find it desirable to continue to distribute standard patches, with or without an associated semantic patch.

What is then needed is a means of mediating between standard patches and semantic patches, by inferring semantic patches from standard patches. In this paper, we propose a tool, `spdiff`, that infers a restricted form of semantic patch, which we refer to as a *generic patch*, from a collection of standard patches implementing a

common set of transformations. The Linux developer who makes a change in a library that affects the library's interface can perform the collateral evolution in a few files based on his knowledge about how drivers typically make use of the library, and then apply `spdiff` to produce a generic patch that can be applied to the other files automatically. Complementarily, the developer who needs to read an existing standard patch implementing a collateral evolution can apply `spdiff` to the patch to obtain a more concise, abstract representation of the common changes that are performed, as well as information about any deviations from these common changes, which may represent bugs or special cases of which he should be aware. If the developer maintains proprietary code outside the Linux kernel source tree, he may furthermore use the inferred generic patch to apply the necessary changes.

Concretely, the contributions of this paper are:

- We provide a formalization of what constitutes a concise and abstract generic patch. The formalization does not rely on particular features of our generic patches and thus could be instantiated for other transformation languages.
- We give an algorithm, `spfind`, that infers concise and abstract generic patches for C code. We have implemented the algorithm `spfind` in a tool `spdiff`.
- We show examples of the generic patches inferred by `spdiff` for some recent collateral evolutions in Linux.

The rest of this paper is organized as follows. Section 2 presents a motivating example that illustrates some of the issues taken into account by our approach. Sections 3 through 5 formally present our algorithm for inferring generic patches, by first defining a core term language, then developing the necessary elements of a theory of patches on this language, and finally defining the algorithm itself. Section 6 illustrates the application of our `spdiff` tool to various recent collateral evolutions performed in the Linux source tree. Section 7 describes related work and Sect. 8 concludes.

2 Motivating example

To motivate the design of `spfind`, we begin with a simple example of a collateral evolution from March 2007[1] and consider the issues involved in inferring a generic patch for it. The collateral evolution required replacing uses of the general-purpose memory copying function `memcpy` that manages network buffers by calls to a special-purpose function, `skb_copy_from_linear`.

Figure 1 shows extracts of two files affected by this collateral evolution and the updates to these files. The lines prefixed with - and + indicate where code was removed and added, respectively. When lines annotated with - and + are adjacent, as in this example, the code on the + lines essentially replaces the code that matched the - lines in the original version. Furthermore, the line that is prefixed with ! has

[1] Git SHA1 identification codes
`1a4e2d093fd5f3eaf8cffc04a1b803f8b0ddef6d` and
`d626f62b11e00c16e81e4308ab93d3f13551812a`.
All patches in this paper can be obtained from
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

```
 static int ax25_rx_fragment(
                ax25_cb *ax25,
                struct sk_buff *skb)
 {
   struct sk_buff *skbn, *skbo;

   if (ax25->fragno != 0) {
   ...
   /* Copy data from the fragments */
   while ((skbo = skb_dequeue(
                  &ax25->frag_queue))
          != NULL) {
-    memcpy(skb_put(skbn, skbo->len),
-           skbo->data,
-           skbo->len);
+    skb_copy_from_linear_data(
+           skbo,
+           skb_put(skbn, skbo->len),
+           skbo->len);

     kfree_skb(skbo);
   }
   ...
 }
 static int ax25_rcv(
            struct sk_buff *skb, ...
 {
   ...
   if (dp.ndigi == 0) {
     kfree(ax25->digipeat);
     ax25->digipeat = NULL;
   } else {
     /*Reverse the source SABM's path*/
!    memcpy(ax25->digipeat, &reverse_dp,
!           sizeof(ax25_digi));
   }
   ...
 }
```

File: net/ax25/ax25_in.c

```
 static
 struct sk_buff *dnrmg_build_message(
        struct sk_buff *rt_skb,
        int *errp)
 {
   struct sk_buff *skb = NULL;
   ...
   if (!skb)
     goto nlmsg_failure;
   ...
-  memcpy(ptr, rt_skb->data,
-              rt_skb->len);
+  skb_copy_from_linear_data(
+         rt_skb, ptr, rt_skb->len);
   ...
 nlmsg_failure:
   if (skb)
     kfree_skb(skb);
   ...
 }
```

File: net/decnet/netfilter/dn_rtmsg.c

```
-  memcpy(skb_put(skbn, skbo->len),
-         skbo->data,
-         skbo->len);
+  skb_copy_from_linear_data(
+       skbo,
+       skb_put(skbn, skbo->len),
+       skbo->len);

-  memcpy(ptr, rt_skb->data,
-              rt_skb->len);
+  skb_copy_from_linear_data(
+         rt_skb, ptr, rt_skb->len);
```

Set of changes for the two files

Fig. 1 Extracts of the two files and the set of changes for the two files (*bottom right*)

superficially the same form as the others, in that it represents a call to `memcpy`, but it is not affected by the collateral evolution. In the first file, two calls to `memcpy` are present initially and only one is affected, and in the second file there is only one such call and it is affected.

A summary of the changes is shown in the bottom-right part of Fig. 1. The summary reveals that although there are differences in how the two files were modified, there are also compelling similarities in the modified parts of the code:

1. All calls to `memcpy` where the second argument references the field `data` from a `sk_buff` structure are changed into calls to `skb_copy_from_linear_data`. On the other hand, in the call to `memcpy` marked with a `!`, the second argument does *not* reference the field `data`.
2. The first argument becomes the second.
3. The field reference to `data` in the original second argument is dropped. The resulting expression becomes the first argument of the new function call.
4. The third argument of `memcpy` is copied as-is to the third argument of the new function call.

The changes made to the two mentioned files can be summarised compactly as the following generic patch derived using our inference tool:

```
@@
    expression X0;
    struct sk_buff *X1;
    expression X2;
@@
-   memcpy(X0,X1->data,X2)
+   skb_copy_from_linear_data(X1,X0,X2)
```

where X0, X1, and X2 serve as placeholders (metavariables) for concrete arguments. The metavariables are declared to match expressions (X0 and X2) and an expression of type struct sk_buff * (X1). Intuitively, the generic patch is an abstract representation of the changes made: in the context of a call to memcpy where the first and third arguments are arbitrary expressions and the second is of type struct sk_buff * and references the data field of the sk_buff structure, then change the called function to skb_copy_from_linear_data, move the first argument to the second position, remove the data field reference of the second argument and make it the first argument in the new function call, and copy the third argument as-is. Thus, the combined requirements on the context in which to make a transformation ensure that only the calls marked with – are affected and leave out the call to memcpy marked with !, as required.

There are two main issues to be considered when inferring generic patches: (1) compactness and (2) safety.

Compactness The most trivial way to construct a generic patch is simply to enumerate the changes, as done for the example earlier in this section. The result, however, would be no better than a standard patch, and it would generally not be applicable to files other than the ones used for the inference. Finally, it would generally not be readable as high-level documentation of the changes performed. We prefer, therefore, a more compact description of the changes. We produce the more compact description by replacing subterms that are not affected by the transformation by metavariables. The use of metavariables is illustrated in the generic patch above where e.g., X0 is used rather than the concrete terms skb_put(skbn,skbo->len) (in the file ax25_in.c) and ptr (in the file dn_rtmsg.c).

Safety The safety of a generic patch requires that only things that were actually changed in the original file should be changed by the inferred generic patch. In our example, one of the calls to memcpy was not changed. We saw that we could ensure safety by imposing structural and type-based restrictions on the second argument to memcpy: only those calls where the second argument had the correct type and referenced the data field should be affected.

In the next two sections we develop the machinery needed to present an algorithm that can automatically infer compact and safe generic patches such as the one shown above. In the example above, there was only one change, but the method we describe is capable of deriving a generic patch that can perform multiple changes at code sites scattered throughout the source program, and always ensures that the derived generic patch correctly describes a set of transformations that apply to all the files given as input.

3 Setup

While our approach targets C code, we formalise it in terms of a simpler language, which we call the language of TERMs. The syntax of the language of TERMs is given as:

Definition 1 (Syntax of Terms)

$$\text{TERM} ::= \text{ATOM} \mid \text{ATOM}(\text{TERM}^+)$$

In this definition, and subsequently, t^+ indicates one or more comma-separated occurrences of the nonterminal t. Furthermore, terms will be written as a and $a(ts)$, for atomic and compound terms respectively, where a represents an ATOM and ts represents a TERM^+.

Generic patches describe updates on terms. A generic patch is created out of patterns, as defined below.

Patterns A pattern is a TERM that may additionally contain *metavariables*, which are placeholders for concrete terms. The syntax of patterns is as follows:

Definition 2 (Syntax of Patterns)

$$p ::= \text{ATOM} \mid \text{ATOM}(p^+) \mid \mathit{Meta} \mid \mathit{Meta}(p^+)$$

where *Meta* denotes a set of metavariables. In the examples, we use uppercase letters to denote metavariables.

A pattern p matches a term t if there is a substitution θ of metavariables for terms such that applying the substitution to the pattern yields a term that is syntactically equivalent to t, i.e., $\theta p = t$ where θp denotes application of θ to p. A metavariable may occur more than once in a pattern, in which case all occurrences of the metavariable must match the same concrete term. For example, the pattern `f(X,X)` matches any concrete term that is a call to `f` with two syntactically equal arguments.

Term replacement patches A term replacement patch describes how to transform any (sub)terms that match a given pattern. The syntax of term replacement patches is as follows:

Definition 3 (Syntax of Term Replacement Patches) A term replacement patch, *trp*, has the form defined by the following grammar:

$$\mathit{trp} ::= p \rightsquigarrow p$$

In a term replacement patch, $p_1 \rightsquigarrow p_2$, p_1 is a pattern that should match a subterm of the input term, and p_2 is a pattern that describes the term that should replace the matched term.

$$(a)\quad \frac{\exists\theta : \theta p = t \qquad \theta p' = t' \qquad MV(p') \subseteq MV(p)}{(p \rightsquigarrow p')(t) = t', \top}$$

$$(b)\quad \frac{\begin{array}{c}\neg\exists\theta : \theta p = a(t_0, \ldots, t_n) \\ (p \rightsquigarrow p')(t_i) = t_i',\ f_i \text{ for all } 0 \leq i \leq n \\ f = \bigsqcup f_i\end{array}}{(p \rightsquigarrow p')(a(t_0, \ldots, t_n) = a(t_0', \ldots, t_n'), f}$$

$$(c)\quad \frac{\neg\exists\theta : \theta p = a}{(p \rightsquigarrow p')(a) = a, \bot}$$

Fig. 2 Application of a term replacement patch

The application of a term replacement patch to a term is defined by the rules shown in Fig. 2.[2] Rule a addresses the case where a term t matches the pattern p according to some substitution θ. The matching term t is replaced with $\theta p'$. This rule further requires that the metavariables in the right-hand side pattern $MV(p')$ be a subset of those in the left-hand side pattern $MV(p)$. The remaining rules apply only if rule a does not apply. Rule b recursively traverses a non-atomic term top-down to all its subterms. Rule c processes an atomic term that does not match the pattern. If there is no matching subterm, the application of a term replacement patch behaves as the identity function. Term replacement patch application terminates because the only recursive application is in rule b, in which the term replacement patch is only applied to proper subterms of the original term.

The application of a term replacement patch additionally returns a flag f, which is $\top$ when a match has been found, and $\bot$ when no match has been found. These are ordered as $\bot \sqsubseteq \top$. Note that even if there is a match, the resulting generated term might be the same as the original one, e.g., if the term replacement patch specifies that the two arguments of a function should be switched, and they are actually textually equal. The flag distinguishes between the case where there is a match and the term is unchanged ($\top$) and the case where there is no match at all ($\bot$).

Generic patches A generic patch is a sequence of one or more term replacement patches, as defined by the following grammar:

Definition 4 (Syntax of Generic Patches) A generic patch is either a term replacement patch or a sequence of generic patches.

$$gp ::= p \rightsquigarrow p \mid gp; gp$$

[2]Note that although term replacement patches have the form of rewrite rules, they are not applied iteratively as is typically done in term rewriting systems. Iteration would be possible, but would significantly complicate the semantic patch inference process. Furthermore, iteration is not provided by Coccinelle, and this has not been found to be a significant limitation in practice.

Subsequently, whenever we say "patch," we mean generic patch unless stated otherwise. Also, we say that a generic patch is *non-abstract* if no pattern used in the generic patch contains any metavariables.

The rules for applying a generic patch are shown below. The application of a term replacement patch $p_1 \rightsquigarrow p_2$ is defined according to the rules of Fig. 2, but here the application only succeeds if the pattern matches somewhere, as indicated by the flag $\top$. A sequence of patches $gp_1; gp_2$ first applies gp_1 to the term and then applies gp_2 to the result.

$$\begin{aligned} [\![p \rightsquigarrow p']\!]t &= t'' && \text{if } (p \rightsquigarrow p')(t) = t'', \top \\ [\![gp_1; gp_2]\!]t &= [\![gp_2]\!]([\![gp_1]\!]t) \end{aligned}$$

Note that the side condition $(p \rightsquigarrow p')(t) = t'', \top$ implies that every term replacement patch in a generic patch must be able to successfully apply somewhere within the argument term t for the generic patch to apply at all.

If the result of applying $gp_1; gp_2$ to a term is independent of the ordering of gp_1 and gp_2, the patches are said to be *commutative*.

Two generic patches are equivalent with respect to a set of terms T if and only if application of the patches has the same effect on all terms:

$$gp_1 \equiv gp_2 \iff \forall t \in T : [\![gp_1]\!]t = [\![gp_2]\!]t$$

Two generic patches are equivalent with respect to a set of pairs of terms (which we shall also call a *changeset*) C if and only if application of the patches has the same effect on all *left-hand side* terms in C:

$$gp_1 \equiv_C gp_2 \iff \forall (t, t') \in C : [\![gp_1]\!]t = [\![gp_2]\!]t$$

Whenever the changeset, C, is clear from the context, we will write $gp_1 \equiv gp_2$ instead of $gp_1 \equiv_C bp_2$.

4 Theory of subpatches

To satisfy the criteria of safety and compactness, we would like to infer a generic patch that expresses the *largest* possible *common* transformation applied to each term, without performing any undesired transformations (safety). In this section, we provide a formal definition of what it means for the transformation performed by a generic patch to be largest and common.

4.1 Ensuring safety

Safety of a generic patch requires that it does not perform undesired changes. Formally, safety is defined relative to a pair of terms (t, t') with the assumption that some change has been made in t to turn it into t'. A patch gp is said to be safe relative to a pair of terms when the transformations it makes when applied to t need not be *undone* in order to reach the final term t'. To illustrate this concept, the following example shows a patch that performs a safe transformation and another patch that does not perform a safe transformation relative to a pair of terms.

Example 1 (Illustration of safety) Consider the following terms:

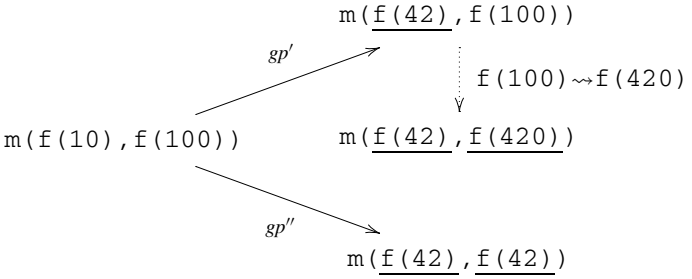
$$\begin{aligned} t &= \texttt{m(f(10),f(100))} \\ t' &= \texttt{m(f(42),f(100))} \\ t'' &= \texttt{m(f(42),f(42))} \\ t''' &= \texttt{m(f(42),f(420))} \end{aligned}$$

and the following patches:

$$\begin{aligned} gp' &= \texttt{f(10)} \rightsquigarrow \texttt{f(42)} \\ gp'' &= \texttt{f(X)} \rightsquigarrow \texttt{f(42)} \end{aligned}$$

Observe that $[\![gp']\!]t = t'$ and $[\![gp'']\!]t = t''$. We now consider whether gp' or gp'' is safe with respect to the pair (t, t''').

The following diagram illustrates the application of and gp'' to t, and compares the result to t''' (center right) where the subterms that have been affected by a patch are underlined. Arrows between terms are labeled with the patch that transforms the source term to the target term.



After the application of gp', only the subterm `f(100)` (top right) of t' needs to be modified in order to reach t''' (center right). As can be seen by the fact that `f(100)` is not underlined it was not modified by gp'. Thus, gp' is safe relative to (t, t'''). After application of gp'', on the other hand, we need to modify the second generated occurrence of `f(42)` into `f(420)` in order to reach t'''. Thus, we need to *undo* some of the changes made by gp'', Thus, gp'' is not safe relative to (t, t''').

To capture the "no-undoing" property that we have motivated informally in Example 1 above, we define a distance metric on terms. The distance between two terms is defined as the cost of performing the minimal number of insertions and deletions of subterms needed to turn one term into the other. The cost of inserting or deleting a subterm is defined as the size of the subterm. We denote the size of a term t by $|t|$. The function calculating the size of a term is given by induction on the structure of terms:

$$\begin{aligned} |a| &= 1 \\ |c(t_0, \ldots, t_n)| &= 1 + \sum_{i=0}^{n} |t_i| \end{aligned}$$

The distance between two terms t and t' is denoted $\delta(t, t')$. The definition of term distance is given in Definition 5 below. The definition makes use of an auxiliary function $\Delta(ts, ts')$ that takes two lists of terms ts and ts' and uses a dynamic programming scheme to find the minimal distance between the two lists. We make liberal use of the TERM^+ notation introduced in Definition 1 and let $(t : ts)$ denote a list of terms that starts with t and has remainder ts. Also, we let ϵ denote an empty list of terms although that is technically not a valid TERM^+.

Definition 5 (Term distance)

$$\delta(t, t') = \begin{cases} 0 & \text{if } t = t' \\ v & \text{if } t = a(ts) \wedge t' = a'(ts') \wedge \\ & v = \begin{cases} \Delta(ts_1, ts_2) & \text{if } a = a \\ 2 + \Delta(ts_1, ts_2) & \text{otherwise} \end{cases} \\ |t| + |t'| & \text{otherwise} \end{cases}$$

where

$$\Delta((t : ts), (t' : ts')) = \min \begin{Bmatrix} \delta(t, t') + \Delta(ts, ts'), \\ \Delta((t : ts), ts') + |t'|, \\ \Delta(ts, (t' : ts')) + |t| \end{Bmatrix}$$

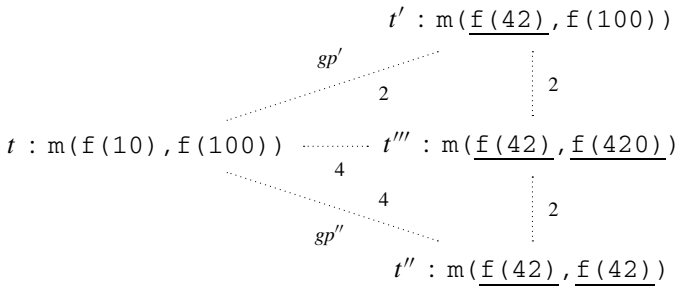
$$\Delta(\epsilon, (t_0, \ldots, t_n)) = \Delta((t_0, \ldots, t_n), \epsilon) = \sum_{i=0}^{n} |t_i|$$

Example 2 (Term distances) Reconsider the terms from Example 1 (repeated below for ease of reference)

$$\begin{aligned} t &= \texttt{m(f(10),f(100))} \\ t' &= \texttt{m(f(42),f(100))} \\ t'' &= \texttt{m(f(42),f(42))} \\ t''' &= \texttt{m(f(42),f(420))} \end{aligned}$$

One can see that $\delta(t, t''') = 4$, $\delta(t, t') + \delta(t', t''') = 2 + 2 = 4$, and $\delta(t, t'') + \delta(t'', t''') = 4 + 2$.

We now connect the idea of term distance to the idea of a safe transformation. In Example 1 we motivated why gp should be considered safe with respect to the pair (t, t''') and gp' not safe by showing the changes required to reach the target term t''' after application of the patches: after application of gp only non-changed subterms needed to be changed to reach t''' while after application of gp' one needs to modify an already modified subterm (`f(42)`) to reach t'''. A similar picture is shown in the diagram below; One can see that more changes (insertions and deletions) are performed when following the path $t \to t'' \to t'''$ (with summarised cost 6) than when following the path $t \to t' \to t'''$ (with summarised cost 4). In fact, exactly the same changes are required to first change t to t' and subsequently change t' to t''' as the changes required when transforming t into t''' directly.



Based on Definition 5 we now formally define the concept of a *safe transformation part* of a pair of terms (t, t''). A safe transformation part safely performs all or part of a transformation on t, such that no undoing is required to subsequently reach t''.

Definition 6 (Safe Transformation part) For any pair of terms (t, t') and patch gp:

$$gp \preceq (t, t'') \iff [\![gp]\!]t = t' \Rightarrow \delta(t, t') + \delta(t', t'') = \delta(t, t'')$$

Note that the definition of $[\![gp]\!]t = t'$ implies that gp does apply to t.

Extension to changesets We extend the notion of safety to a changeset by simply quantifying over all the pairs of terms in the changeset.

Definition 7 (Common Patch) A patch gp is a common patch for a set of pairs of terms C if it is safe relative to each pair of terms in the set C:

$$gp \preceq C \iff \forall (t, t'') \in C : gp \preceq (t, t'')$$

We will sometimes say that a patch is a *safe* common patch to emphasise that the patch is indeed a safe patch with respect to all pairs of terms in the changeset C.

4.2 Ensuring compactness

In addition to seeking a safe generic patch, we also seek a generic patch that *compactly* represents the changes made. In order to define that one generic patch gp' is more compact than another generic patch gp, we define an ordering of patches relative to a pair of terms: $gp \preceq_{(t,t'')} gp'$. When $gp \preceq_{(t,t'')} gp'$ the transformations expressed by gp are also contained in gp' and we say that gp is a subpatch of gp'.

Definition 8 (Patch Ordering Relation)

$$gp \preceq_{(t,t'')} gp' \iff gp' \preceq (t, t'') \wedge \exists t' : [\![gp']\!]t = t' \wedge gp \preceq (t, t')$$

Thus, we consider gp a subpatch of gp' (or equivalently, gp' a superpatch of gp) if and only if gp performs a safe part of the transformation that gp' performs, as expressed by $gp \preceq (t, t')$ above, and gp' is a safe part of (t, t'').

A patch gp is a trivial subpatch of gp' if gp is equivalent to the identity patch with respect to the pair of terms (t, t').

The subpatch definition can be generalised to a set of pairs of terms as follows:

$$gp \preceq_C gp' \iff \forall (t, t'') \in C : gp \preceq_{(t,t'')} gp'$$

Example 3 (Super- and subpatches) In this example we show that when a patch gp' is a superpatch of gp, it can actually be *syntactically* smaller than gp.

Consider the following terms:

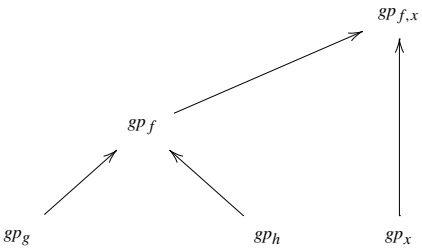
$$\begin{array}{ll} t_f = \texttt{f(1)} & t'_f = \texttt{f(1,1)} \\ t_g = \texttt{g(f(1))} & t'_g = \texttt{g(f(1,1))} \\ t_h = \texttt{h(f(1))} & t'_h = \texttt{h(f(1,1))} \\ t_x = \texttt{x(f(2))} & t'_x = \texttt{x(f(2,2))} \end{array}$$

$$\begin{array}{l} t_m = \texttt{m(g(f(1)) ,h(f(1)) +x(f(2)))} \\ t'_m = \texttt{m(g(f(1,1)),h(f(1,1)+x(f(2,2))))} \end{array}$$

as well as the following patches:

$$\begin{array}{ll} gp_f & = t_f \rightsquigarrow t'_f = \texttt{f(1)} \rightsquigarrow \texttt{f(1,1)} \\ gp_g & = t_g \rightsquigarrow t'_g = \texttt{g(f(1))} \rightsquigarrow \texttt{g(f(1,1))} \\ gp_h & = t_h \rightsquigarrow t'_h = \texttt{h(f(1))} \rightsquigarrow \texttt{h(f(1,1))} \\ gp_x & = t_x \rightsquigarrow t'_x = \texttt{x(f(2))} \rightsquigarrow \texttt{x(f(2,2))} \\ gp_{f,x} & = \texttt{f(X)} \rightsquigarrow \texttt{f(X,X)} \end{array}$$

The subpatch hierarchy is given below. An arrow from gp_i to gp_j indicates that relative to the term pair (t_m, t'_m), gp_i is a subpatch of gp_j (and gp_j a superpatch of gp_i).



From the hierarchy one can see that although gp_f is syntactically smaller than both gp_g and gp_h it is a superpatch of both. Finally, we see that $gp_{f,x}$ is highest in the subpatch hierarchy and that it covers both the transformations made by gp_f and gp_x. Therefore it can be considered more compact than the combination of the two smaller patches.

The largest safe common subpatch

Definition 9 Suppose C is a set of pairs of terms $\{(t_0, t_0''), \ldots, (t_n, t_n'')\}$ representing the original and updated code after some manual collateral evolution has been performed. A largest safe common subpatch for C is then a patch gp satisfying the following properties:

$$gp \preceq C \tag{1}$$

$$\forall gp' : gp' \preceq C \Rightarrow gp' \preceq_C gp \tag{2}$$

Property (1) expresses that gp is a safe common patch and property (2) expresses that gp is largest among the common subpatches. Since there can be more than one patch satisfying the requirements above, we let $LCP(C)$ be the set of largest safe common patches for a set of pairs of terms C. The set $LCP(C)$ is thus the set of the most compact and safe generic patches relative to a set C.

Example 4 (Largest Safe Common Subpatch) The following is an example of a largest safe common subpatch for a set of pairs of terms $C = \{(t_1, t_1'), (t_2, t_2')\}$. This example shows that the largest safe common subpatch need not be unique.

$t_1 =$
```
int foo(void) {
  int x;
  f(117);
  x = g(117);
  return x;
}
```

$t_1' =$
```
int foo(void) {
  int x;
  f(117,GFP);
  x = h(g(117));
  return x+x;
}
```

$t_2 =$
```
int bar(int y) {
  int a;
  a = f(11)+g(y);
  return a;
}
```

$t_2' =$
```
int bar(int y) {
  int a;
  a = f(11,GFP)+g(y);
  return a+a;
}
```

The changes made to the two terms t_1 and t_2 are enumerated below. In the term t_1 three changes are made: the call to `g(117)` is embedded in another call to `h`, the call to `f(117)` gets an extra argument `GFP`, and the expression returned is added to itself. In the term t_2 only two changes are made: the call to `f(11)` gets an extra argument `GFP` and the expression returned is added to itself. Schematically:

Updates applied to t_1:
```
-  f(117)
+  f(117, GFP)

   -  g(117)
   +  h(g(117))

   -  return x;
   +  return x+x;
```

Updates applied to t_2:
```
      -  f(11)
      +  f(11,GFP)

      -  return a;
      +  return a+a;
```

Given the above definition of C, the set $LCP(C)$ consists of exactly two generic patches:

$$\left\{\begin{array}{l} \texttt{f(X)} \rightsquigarrow \texttt{f(X,GFP)} \;\; ; \;\; \texttt{return Y} \rightsquigarrow \texttt{return Y+Y}, \\ \texttt{return Y} \rightsquigarrow \texttt{return Y+Y} \;\; ; \;\; \texttt{f(X)} \rightsquigarrow \texttt{f(X,GFP)} \end{array}\right\}$$

The difference between the two is the order in which the two term replacement patches are applied. However, the result of applying either patch to each term in C is the same. Note that the transformation on `g` does not appear in $LCP(C)$, because there is no transformation on `g` in the term t_2.

The example above motivates the following theorem of equivalence of the set of largest safe common subpatches.

Theorem 1 *For all sets of pairs of terms C, all of the patches in the set $LCP(C)$ are extensionally equivalent with respect to the application function*:

$$\begin{array}{l} \forall C : \forall gp, gp' : gp \in LCP(C) \Rightarrow \\ \qquad\qquad\qquad\quad gp' \in LCP(C) \Rightarrow \\ \qquad\qquad\qquad\quad (t_i, t_i'') \in C \Rightarrow \\ \qquad\qquad\qquad\quad gp \equiv gp' \end{array}$$

Proof of Theorem 1 (sketch) Given a set of pairs of terms C, let $B = \{gp \mid gp \preceq C\}$. The goal is to show that the pair consisting of the quotient set $B/_{\equiv}$ (where $\equiv$ denotes equivalence of generic patches with respect to C) and the subpatch ordering $\preceq_C$ $(C/_{\equiv}, \preceq_C)$ forms a complete join semi-lattice and that the least upper bound is in fact the set $LCP(C)$. Then, since any element of $B/_{\equiv}$ is a *set of equivalent* generic patches, Theorem 1 follows.

□

Inverse monotonicity Given a set of pairs of terms C, let B be the set of largest common subpatches for C: $B = LCP(C)$. Adding more pairs to the set C will decrease the size of B:

$$\forall C, C' : C \subseteq C' \Rightarrow LCP(C') \subseteq LCP(C)$$

In particular $LCP(C')$ can become *empty*. This can happen in two ways: 1) the transformations in the pairs added to C have nothing in common with the transformations in C, or 2) the transformations in $LCP(C)$ are not and cannot be refined to be *safe* for the new pairs of terms.

Summary In this section, we have defined when a generic patch transforms terms in a set of example terms in a correct manner. We have also defined when a generic patch is a subpatch of another. Using the subpatch relation, we have defined the set of largest safe common subpatches, $LCP(C)$. Each patch in $LCP(C)$ (recall that according to Theorem 1 they are all equivalent) is a high-level expression of all common changes applied in all pairs of terms in C. In Sect. 6, we will show how inference of largest safe common subpatches can help obtain a succinct form of documentation of changes made in Linux device drivers.

5 The `spfind` algorithm and implementation

We now present an algorithm for finding the set of largest common subpatches given a set of pairs of terms, C, as well as details about the implementation of the algorithm.

We first present a very simple algorithm for computing largest common subpatches that leaves out any performance concerns and other details that must be addressed when implementing the algorithm. This very simple algorithm is mainly to be considered as an outline of the algorithm's basic structure. Based on an observation about a "sufficient level of abstraction" we then refine the algorithm in two steps. The refined algorithm returns a subset of what the simple algorithm returns such that all patches returned are of minimal abstractness. Finally, we present the two main issues we have addressed in our implementation: non-uniform input data and constant-time comparison of terms.

5.1 A simple algorithm

We now present the simple version of our algorithm to compute the set of largest common subpatches for a given changeset. The algorithm is denoted `spfind`. Overall, `spfind` works in two steps: (1) finding term replacement patches (i.e. patches of the form $p \rightsquigarrow p'$) and (2) growing larger sequential patches (i.e., patches of the form $bp_1; \ldots; bp_n$) from the set of found term replacement patches.

The pseudocode of the `spfind` algorithm is written in the style of a functional programming language, such as OCaml or Standard ML. In the pseudocode, we frequently make use of set-comprehensions to compute sets.

Definition 10 (`spfind`—simple version)

```
simple_pairs C =
  { p->p' | (lhs,rhs) ∈ C,
            t is a subterm of lhs,
            t' is a subterm of rhs,
            ∃θ,p,p': θp = t , θp' = t',
            p⇝p' ⪯ (lhs,rhs)
  }

computeNext C B cur = {p⇝p' | p⇝p' ∈ B, (cur;p⇝p') ⪯ C}

gen C B cur =
  let next = computeNext C B cur in
  if next == {}
  then {cur}
  else {gp | p⇝p' ∈ next, gp ∈ gen C B (cur;p⇝p')}

spfind C =
  let B = simple_pairs C in
  {gp | gp ∈ gen C B bp, bp ∈ B}
```

The main functions in the algorithm are `simple_pairs` and `gen`. The function `simple_pairs` constructs term-replacement patches and the function `gen` grows larger sequential generic patches.

Construction of term-replacement patches The `simple_pairs` function takes a set of pairs of terms C and constructs the set $\{gp \mid gp \preceq C\}$ by considering abstractions of all subterms in the given set of pairs of terms C. In the pseudocode, patterns `p` and `p'` (that may contain metavariables) more or less "magically" appear in the line with existential quantification. In the refined algorithm in Sect. 5.2 we describe a method to construct such patterns from a set of terms. Another, more serious, issue is that even for small examples, the `simple_pairs` function can return *many* term replacement patches–an upper bound for $p \rightsquigarrow p'$ is $\mathcal{O}(2^{|p|*|p'|})$ where $|\cdot|$ denotes the number of subterms in a pattern. We illustrate this in Example 5 below.

Example 5 Let the following be given

```
t1 = g(f(42))          t2= g(f(117))
t1'= g(f(42,42))       t2'= g(f(117,117))
C  = { (t1,t1') ,      (t2, t2') }
```

Part of the result of `simple_pairs C` is show below. Recall that capital letters denote metavariables.

```
simple_pairs C =
{
   g(f(X)) ⇝ g(f(X,X)),   g(Y(X)) ⇝ g(f(X,X)),
   Z(f(X)) ⇝ g(f(X,X)),   Z(Y(X)) ⇝ g(f(X,X)),
   g(Y(X)) ⇝ g(Y(X,X)),   Z(f(X)) ⇝ Z(f(X,X)),
   Z(Y(X)) ⇝ Z(Y(X,X)),   ...
}
```

The above result-set for `simple_pairs` illustrates that many very similar patches can be returned. In this example the complete set contains $\mathcal{O}(2^{5*6})$ term replacement patches where 5 is the number of subterms of `g(f(42))` and 6 is the number of subterms of `g(f(42,42))`. A few of those are not safe for the changeset, but most are. Thus, we are motivated to find a way to limit the size of this set to return fewer representative patches. This will be done in Sect. 5.2.

Growing sequential generic patches The `gen` function takes three inputs: a changeset `C`, a set of term-replacement patches `B`, and a generic patch `cur`. The function then tries recursively to extend the generic patch `cur` with term replacement patches from the set `B`. Roughly, the `gen` function generates all sequential patches that are *safe and largest with respect to* `B`, composed of permutations of elements of `B`, where no element occurs more than once in each generated sequential patch. Thus, a call to `gen C B cur` returns a set of patches that have the shape $\texttt{cur}; \texttt{p}_1 \rightsquigarrow \texttt{p}'_1; \ldots; \texttt{p}_n \rightsquigarrow \texttt{p}'_n$ where $\texttt{p}_i \rightsquigarrow \texttt{p}'_i \in \texttt{B}$ and $\texttt{cur}; \texttt{p}_1 \rightsquigarrow \texttt{p}_1; \ldots; \texttt{p}_n \rightsquigarrow \texttt{p}'_n \preceq \texttt{C}$.

The main function of the `spfind` algorithm calls `gen` for each term replacement patch found by `simple_pairs`.

Relationship between algorithm and specification Unfortunately, the `spfind` algorithm is neither sound nor complete in the strict sense. There are patches in $LCP(C)$ that `spfind` will not find and indeed some patches found by `spfind` are not largest. An example illustrates both cases and hints at why the lack of soundness and completeness is not as bad as it sounds.

Example 6 (No soundness or completeness) Assume the following definitions are given:

```
t1  = h(f(1),1)
t1' = h(f(2),3)
t2  = t(f(1),42,1,117))
t2' = t(f(2),42,3,117))
C   = {(t1,t1'), (t2,t2')}
```

We notice that $gp_1 = \texttt{f(1)} \rightsquigarrow \texttt{f(2)}$ is safe for `C` while $gp_2 = \texttt{1} \rightsquigarrow \texttt{3}$ is not because it updates too many occurrences of `1`. However, $gp_1; gp_2$ *is* safe for `C` because now, when gp_2 is applied, there is only one occurrence of `1` and that occurrence is supposed to be updated to `3`. In fact, $gp_1; gp_2$ is one of the largest common subpatches for C.

All patches returned by `spfind` are either term replacement patches or sequences of term replacement patches all of which are individually safe for the changeset given. Therefore, `spfind C` will not contain a patch that is composed of $\texttt{1} \rightsquigarrow \texttt{3}$. It can also be shown that `spfind C` cannot contain a patch that is equivalent to $gp_1; gp_2$ because the occurrences of `1` in `t1` and `t2` that have to be changed have no common term structure expressible in a generic patch.

Therefore, we can conclude that `spfind` is not complete. The result of `spfind C` will contain gp_1 which is not in $LCP(C)$. We can therefore also conclude that `spfind` is not sound in the strict sense that for all patches found by `spfind C`, there must exist an *equivalent* patch in $LCP(C)$.

Example 6 gives a counterexample that shows that `spfind` is neither sound nor complete. However, it *is* the case that the patches returned by `spfind C` are either equivalent to some in $LCP(C)$ or they are each a non-trivial subpatch of some patch in $LCP(C)$. Thus, each patch found by `spfind C` is safe for `C` and when applied, it will have *some* safe effect on the terms.

5.2 Towards a refined algorithm

There are two problems in the simple version of the `spfind` algorithm:

Term replacement patches: In the function `simple_pairs`, we use existential quantification to introduce patterns that contain metavariables. There are two problems to tackle: (1) how do we actually find such patterns and (2) is there a *sufficient* subset of this set of patterns?

Search-space pruning: The `gen` function uses the `computeNext` function to find a subset of the set of term replacement patches `B` that can be used to extend the generic patch being generated (`cur`). For each term replacement patch in this subset, `gen` will try to extend the current generic patch with the particular element. Thus, if we can limit the size of the `next` subset in `gen`, fewer recursive calls to `gen` ensue.

Sufficient term replacement patches Consider once more the terms from Example 5:

```
t1 = g(f(42))          t2= g(f(117))
t1'= g(f(42,42))       t2'= g(f(117,117))
C  = { (t1,t1') ,      (t2, t2') }
```

As shown in Example 5 there are many potential term replacement patches that are safe for the set `C`. Let $B = \{p \rightsquigarrow p' \mid p \rightsquigarrow p' \preceq \texttt{C}\}$. In order to define a sufficient subset of B, we observe that some of the term replacement patches are needlessly abstract. A term replacement patch is needlessly abstract if there is an equivalent term replacement patch that is less abstract. For the set `C` above, the most abstract patch is `X(Y(Z(Q)))` $\rightsquigarrow$ `X(Y(Z(Q,Q)))` and the least abstract is `g(f(X))` $\rightsquigarrow$ `g(f(X,X))`. It is easy to see that in this example all elements of B are equivalent. The important property of two equivalent term replacement patches is that the patches that can be grown from one are equivalent to the patches that can be grown from the other:

Lemma 1 (Equivalent patches implies equivalent suffix extensions)

$$\forall trp_1, trp_2, gp : trp_1 \equiv trp_2 \Rightarrow trp_1; gp \equiv trp_2; gp$$

The conclusion one can draw from Lemma 1 is that for a set of equivalent term replacement patches, we only need to return one of them.

Finding term replacement patches Given the set $\{p \rightsquigarrow p' \mid p \rightsquigarrow p' \preceq C\}$ we can construct the sufficient subset of term replacement patches based on the above observations. Implemented naively, we would still need to search the entire set $\{p \rightsquigarrow p' \preceq C\}$ and that is potentially very time-consuming.

With the goal of avoiding the construction of the complete initial set, we now define a fusion operator on term replacement patches, $trp_1 \ast\ast trp_2$. The patch fusion operator constructs a new term replacement patch encompassing both of the given patches. The *patch* fusion operator relies on a *pattern* fusion operator which we define below. The definition of patch fusion is then given in Definition 12.

In order to perform patch fusion, we need to be able to fuse two patterns into a new pattern that is a superpattern of both given patterns. A simple definition of the pattern fusion function could be: $p * p' = X$, but this definition is useless for our purpose. Instead we would like the pattern resulting from $p * p'$ to abstract as little as possible and still obtain a pattern that matches both p and p'.

Definition 11 (Pattern fusion relation) The pattern fusion relation is given below. To make the rules less verbose, we assume that the rules are considered in numerical

order. That is, if the patterns p and p' are the same, rule (1) is to be used. Otherwise, if the two patterns use the same top-level constructor, a, and have the same arity, n, rule (2) is used. Otherwise, rule (3) is used. Note that the environment used in the rules maps a metavariable to a *pair* of terms.

$$(1)\frac{}{\theta \vdash p * p = p} \qquad (2)\frac{0 < i \leq n \qquad \theta \vdash p_i * p'_i = p''_i}{\theta \vdash a(p_1, \ldots, p_n) * a(p'_1, \ldots, p'_n) = a(p''_1, \ldots, p''_n)}$$

$$(3)\frac{\theta(X) = (p, p')}{\theta \vdash p * p' = X}$$

One can show that for any two patterns p and p' it is always the case that $\exists \theta, p'' : \theta \vdash p * p' = p''$ for some θ and p''. Furthermore, the resulting pattern is unique up to alpha-equivalence of patterns: $\theta_1 \vdash p * p' = p_1 \wedge \theta_2 \vdash p * p' = p_2 \Rightarrow p_1 \cong p_2$ where $p \cong p'$ denotes that p and p' are alpha-equivalent. One can show uniqueness by induction on the derivations and using the fact that the rules defining $p * p'$ are deterministic. Thus, we will write $p * p' = p''$ with the meaning: $\exists \theta : \theta \vdash p * p' = p''$ and treat the relation as a *function* in the following. One can furthermore show that the pattern fusion function is commutative $p * p' = p' * p$ and associative $(p * p') * p'' = p * (p' * p'')$. Commutativity and associativity can be used to show that given a set of terms $T = \{t_1, \ldots, t_n\}$, a pattern that matches all terms and is the least abstract is given by: $t_1 * t_2 \cdots * t_n$.

Using the pattern fusion function we can define fusion of term replacement *patches* $gp \ast\ast gp'$ as the fusion of the embedded patterns in the patches and a renaming of metavariables. The definition can be seen in Definition 12 below.

Definition 12 (Fusion of term replacement patches)

$$\begin{array}{l} p_1 \rightsquigarrow p'_1 \ast\ast p_2 \rightsquigarrow p'_2 = p_3 \rightsquigarrow (\theta_3 p'_3) \Longleftrightarrow \\ \quad \theta_1 \vdash p_1 * p_2 = p_3 \quad \theta_2 \vdash p'_1 * p'_2 = p'_3 \\ \quad \theta_2 \ \mathit{renamed\_by}\ \theta_1 = \theta_3 \end{array}$$

θ *renamed_by* θ' renames metavariables in θ using those in θ' to produce a new environment θ''. Thus, θ'' is a mapping from metavariables to metavariables.

$$\begin{array}{l} \theta \ \mathit{renamed\_by}\ \theta' = \theta'' \Longleftrightarrow \\ \quad \forall X \in \mathit{dom}(\theta) : \theta''(X) = \begin{cases} Y & \text{if } \exists Y : \theta'(Y) = \theta(X) \\ Z & \text{otherwise, where } Z \notin \mathit{dom}(\theta') \end{cases} \end{array}$$

It is instructive to consider an example of term replacement patch fusion to see what is going on.

Example 7 (Patch Fusion) Let the following be given:

```
gp1  = f(42,h(117))  ⇝  f(43,h(117+117))
gp2  = f(42,q(118))  ⇝  f(43,q(118+118))
```

We now compute `gp1 ** gp2`, step by step. First we need to fuse the two left-hand sides and the two right-hand sides.

```
[left-hand sides]
f(42,h(117)) * f(42,q(118)) = f(42,X0(X1))

[right-hand sides]
f(43,h(117+117)) * f(43,q(118+118)) = f(43,X1(X3+X3))
```

The environments associated with each fusion are as follows:

```
env   = { X0 ↦ (h,q), X1 ↦ (117,118) }
env'  = { X1 ↦ (h,q), X3 ↦ (117,118) }
```

When fusing patterns we are allowed to select any metavariable names as long as the inference rules in Definition 11 can be used to derive the desired fused patterns. Thus, when fusing the left hand side (non-abstract) patterns we used the metavariables `X0` and `X1` while when fusing the right hand side (non-abstract) patterns, we used `X1` and `X3`. Indeed, the environments need not be consistent with each other. For example, `X1` in the left hand side pattern maps to `(117,118)` while in the right hand side pattern `X1` maps to `(h,q)`. This selection of names is mainly done to illustrate how θ *renamed_by* θ' works and why it is needed.

```
env'' = env' renamed_by env = { X1 ↦ X0, X3 ↦ X1}
```

One can verify that `env″` satisfies the requirements set out in the definition of p *renamed_by* p' in Definition 12 above. Finally, we can construct the fused term replacement patch as:

```
gp1 * gp2 = f(42,X0(X1)) ⇝ env''(f(43,X1(X3+X3)))
          = f(42,X0(X1)) ⇝ f(43,X0(X1+X1))
```

Search-space pruning Even with the above mentioned refinements of the simple pairs function, the `gen` function can end up constructing an exponential number of generic patches. We now consider how to further reduce the number of generated generic patches.

The `gen` function in the simple version of the `spfind` algorithm is given a set B of term replacement patches and tries to grow sequential patches starting from each patch in B. If all those patches are safe for the changeset C also given to `gen`, the `gen` function would have to compute the powerset of B. Suppose however, that all term replacement patches in B are *commutative*. The `gen` function will first try to grow generic patches starting from some element gp in B. After having found all generic patches that start with gp, the `gen` function then tries to grow generic patches starting with some other element gp' from B. Since we assumed all patches in B to be commutative, all generic patches starting with gp found by `gen` are equivalent to

those found to be starting with gp'. The following corollary is a generalization of the observation just made that whenever the `gen` function has already found some safe generic patch for C denoted gp, there is no need to try to grow from a subpatch gp'. The corollary holds even when there are non-commutative patches in the set B as otherwise assumed above.

Corollary 1 (Search-space pruning) *For any generic patches* gp, gp', gp'':

$$gp \in LCP(C) \wedge gp' \preceq_C gp \wedge gp'; gp'' \preceq C \Rightarrow gp'; gp'' \preceq_C gp$$

The corollary follows directly from the definitions of $gp \in LCP(C)$ (Definition 9) and $gp \preceq_C gp'$ (Definition 8) because $gp \in LCP(C)$ implies that for any other gp', if $gp' \preceq C$ then $gp' \preceq_C gp$.

5.3 The refined `spfind` algorithm, `spfind_refined`

We now proceed to describe the refined `spfind` algorithm making use of the solutions to the two problems identified for the simple algorithm.

The `spfind_refined` algorithm is split into two main parts just like the simple version, `spfind`. The former part (the `simple_pairs` function) can be seen in Fig. 3 and the latter (the `gen` function as well as the entry point of the algorithm) in Fig. 4.

Generation of term replacement patches In order to generate a set of term replacement patches based on a given changeset, we first find a set of sets of non-abstract patches and then fuse them to obtain term replacement patches with metavariables.

```
simple_pairs_one (t,t') =
  let loop l r =
    if l == r
    then {}
    else case (l,r) of
      | (a1(ts1), a2(ts2)) ->
          let R = {tu | t1 ∈ ts1, t2 ∈ ts2,
                        tu ∈ simple_pairs_one t1 t2
                  } in
          {l⇝r} ∪ R
      | otherwise -> {l⇝r}
  in
    {tu | tu ∈ loop t t', tu≼(t,t')}

simple_pairs C =
  let loop tu_lists = case tu_lists of
      | [] -> {}
      | [tu_list] -> tu_list
      | (tu_list::tu_lists) ->
           let tu_merged = {tu | tu ∈ loop tu_lists} in
           {tu ** tu_merged | tu ∈ tu_list, tum ∈ tu_merged,
                              (tu ** tum) ≼ C}
      lists = map find_simple_one changeset
in
  {p⇝p' | p⇝p' ∈ loop lists}
```

Fig. 3 Generation of term replacement patches

The function `simple_pairs_one` defined at the top of Fig. 3 finds non-abstract term replacement patches. The function works by taking a pair of terms and finding non-abstract patches that all satisfy the safety criterion: $t_i \rightsquigarrow t_i' \preceq (t, t')$. The function does so by traversing the given pair of terms simultaneously. If the two terms are not equal, a non-abstract patch is added to an intermediate result set. If the two terms are also compound, the function calls itself recursively on the embedded subterms. Once the traversal is done, all non-safe patches are filtered out and only the safe ones are returned.

The `simple_pairs` function (bottom Fig. 3) is given a changeset $C = \{(t_1, t_1'), \ldots, (t_n, t_n')\}$ and applies the `simple_pairs_one` function to each pair obtaining a set of sets of non-abstract term replacement patches:

$$M = \left\{ \left\{ \begin{array}{c} t_{1,1} \rightsquigarrow t'_{1,1} \\ t_{1,2} \rightsquigarrow t'_{1,2} \\ \vdots \\ t_{1,k} \rightsquigarrow t'_{1,k} \end{array} \right\}, \ldots, \left\{ \begin{array}{c} t_{n,1} \rightsquigarrow t'_{n,1} \\ t_{n,2} \rightsquigarrow t'_{n,2} \\ \vdots \\ t_{n,q} \rightsquigarrow t'_{n,q} \end{array} \right\} \right\}$$

where $t_{i,j} \rightsquigarrow t'_{i,j}$ corresponds to the jth non-abstract patch for the ith pair in C. The goal is now to use the patch fusion operator $trp \ast\ast trp'$ to construct more abstract patches. In the following, let `tu_merged` denote the fused patches so far. We describe the three cases of the local `loop` function of `simple_pairs` below. When `loop` is done, `tu_merged` will contain a set of term replacement patches that are safe for all pairs in the changeset.

M is empty: If *M* is empty, the changeset *C* was also empty and thus `tu_merged` should also be empty.

M is a singleton: In this case there is no other set of patches to fuse with, so we simply let `tu_merged` be the only list in *M* at this point. Doing so ensures that the patches returned are minimally abstract.

```
computeNext C B cur = { p⇝p' | p⇝p' ∈ B , cur;p⇝p' ≤ C }

gen C B acc_res cur =
  let next = computeNext C B cur in
  if next == {}
  then {cur} ∪ acc_res
  else fold (λacc_res bp ->
              if ∃gp ∈ acc_res:(cur;bp)≤Cgp
              then acc_res
              else gen C B acc_res (cur;bp)
            ) acc_res next

spfind_refined C =
  let B = simple_pairs C in
  fold (λacc_res bp ->
              if ∃gp ∈ acc_res:bp≤Cgp
              then acc_res
              else gen C B acc_res bp
            ) [] B
```

Fig. 4 Growing sequential generic patches and entry point of the refined algorithm

M contains two or more sets of patches: Let M_i denote one set of patches $t_{i,j} \rightsquigarrow t'_{i,j}$ for some i and M' denote $M \backslash M_i$. The `loop` function first fuses the patches in M' and then combines each of the fused patches with one from M_i. If the fused patch is not safe for the changeset, it is not included in `tu_merged` and otherwise it is. Using the safety check to limit the size of `tu_merged` in the mentioned way is crucial for the running time of the `loop` function. If we had deferred the safety check to outside of the `loop` function, it would need to compute $|M_1| \times |M_2| \times \cdots \times |M_n|$ fused patches.

Growing sequential generic patches The essential difference between the refined `gen` function shown in Fig. 4 and the simple version is that before each recursive call to `gen`, the refined version checks that the extended patch (`cur;bp`) is not a subpatch of one it already found in the accumulated results, `acc_res`. Based on Corollary 1 we can see that extending `cur` with `bp` will only allow us to find patches that are subpatches of what is already in `acc_res`. Therefore, the recursive call is not performed and instead `acc_res` is taken as the result.

5.4 Implementation

We have implemented the `spfind_refined` algorithm in a tool called `spdiff` using the OCaml programming language in roughly 10.000 lines of code. In the implementation two further issues have been addressed. The first issue is the problem of non-uniform input data and the second is a performance issue.

Non-uniform input data We have so far assumed that the given changesets C are "ideal" in the sense that it is possible to find a patch that is safe for *all* pairs in C. When applying our tool to updates in Linux device drivers we found that this was often not the case. Rather, one patch was found safe for a number of pairs while another patch was found safe for a different set of pairs. In this case, the transformations in the former set of pairs may be simply disjoint from the transformations in the latter set of pairs. Our tool would then say that it could find no globally safe update.

To relax the requirement that all changes have to be safe for all pairs we introduced a user-specified *threshold*. The threshold states for how many pairs a change has to be safe before it is considered safe for the given changeset C.

Example 8 (Introducing thresholds) Suppose the changeset defined below is given to `spdiff`:

```
t1 = f(1)     t2 = g(2)+f(1)     t3 = q(f(1),g(2))
t1'= f(1,1)   t2'= g(2+2)+f(1)   t3'= q(f(1,1),g(2+2))

bp1= f(1)->f(1,1)
bp2= g(2)->g(2+2)

C  = {(t1,t1'), (t2,t2'), (t3,t3')}
```

If we run `spdiff` with a threshold of 3, we get no result back because there is no change that is safe for *all* pairs. When we run `spdiff` with a threshold of 2, we

get two possible solutions because `bp1` is safe for {`(t1,t1')`, `(t3,t3')`} and `bp2` is safe for {`(t2,t2')`, `(t3,t3')`}. It is worth noting that `bp1` is not safe for (`t2`, `t2'`) because application of `bp1` will transform `f(1)` which should not be transformed, while `bp2` is unsafe for (`t1`, `t1'`) simply because it does not apply to it.

```
$ spdiff C 2 = ...

Number of solutions: 2

1: f(1)->f(1,1)
2: g(2)->g(2+2)
```

Improving performance Consider the definition of term distance (Definition 5). The function makes frequent use of equality checks on terms. A simple implementation of term equality would compare the two given terms structurally:

```
let t_equal t1 t2 = case (t1, t2) of
            | a1, a2 -> a1 == a2
            | a1(ts1), a2(ts2) ->
                a1 == a2 && |ts1| == |ts2| &&
                ∀ t1∈ts1, t2∈ts2: t_equal t1 t2
```

Clearly, the time complexity of `t_equal` is $\mathcal{O}(n)$ where n is the size of the smaller of the two terms. Our initial implementation used the above equality check and it was not very fast. The big question is: How can we do better?

Pointer-equality is more efficient to compute than structural equality. Concretely, we have made use of a technique called *hash-consing* (or *value numbering*) as implemented in the module `Hashcons` by Conchon and Filliâtre (2006). Their implementation of hashconsing works very well in our setting and allows constant time equality checks.

6 Examples

We now provide a few examples of the use of `spdiff`, based on some recent patches committed to Linux that we have identified using the `patchparse` collateral evolution mining tool (Padioleau et al. 2006). For each standard patch that we have tested, we have constructed the set of pairs of terms, C, from the image of the Linux source tree just before the standard patch was applied and just after.

Adapt to structure changes The following commits, dated November 9, 2007, begin with the log message "convert to use the new `SPROM` structure".[3]

[3]The patches can be obtained from
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

```
95de2841aad971867851b59c0c5253ecc2e19832
458414b2e3d9dd7ee4510d18c119a7ccd3b43ec5
7797aa384870e3bb5bfd3b6a0eae61e7c7a4c993
```

These commits comprise over 650 lines of patch code, and affect 12 files in the `drivers/net` directory or its subdirectories, at 96 locations. In the role of an expert in the affected files, we selected three files from the first commit that illustrate the set of required changes. From these files, `spdiff` infers the following generic patch:

```
X0->sprom.r1 ⇝ X0->sprom ;
sprom->r1.X0 ⇝ sprom->X0
```

The inferred generic patch fully updates all 12 original files in the same way the standard patches did. By careful examination of the standard patches, a person could construct the inferred generic patch by hand. However, there would be no guarantee that the constructed patch is safe. To check safety manually, one would have to consider (1) whether the constructed patch updates the proper locations correctly but does not update locations that were not to be modified, and (2) whether the constructed patch is only a part of the update that is to be performed to a particular file. A standard patch by itself does not provide enough information to do this, because it does not provide a complete view of the unmodified code.

Furthermore, the inferred generic patch updates some other files that were present at the time of the original patches but were overlooked. These files were in other directories and were not updated until February 18, 2008, by another developer.

Structure changes Commit `c32c2f63a9d6c953aaf168c0b2551da9734-f76d2` from February 14, 2008 affects 9 files at 12 locations. The message attached to the commit is "`d_path: Make seq_path() use a struct path argument`". The standard patch attached to the commit is approximately 160 lines. The patch inferred by `spdiff` is:

```
seq_path(X1,X2->X3.mnt,X2->X3.dentry,X4)
  ⇝ seq_path(X1,&X2->X3,X4)
```

The inferred generic patch fully updates all but one of the original files. The only file that is not fully updated is the file `fs/namespace.c` in which a declaration `struct path mnt_path;` is also added.

Renaming of function calls The following commits, dated December 20, 2007, begin with some variant of the log message "`Kobject: convert drivers/* from kobject_unregister() to kobject_put()`".

```
c10997f6575f476ff38442fa18fd4a0d80345f9d
78a2d906b40fe530ea800c1e873bfe8f02326f1e
197b12d6796a3bca187f22a8978a33d51e2bcd79
38a382ae5dd4f4d04e3046816b0a41836094e538
```

These commits comprise almost 800 lines of patch code, and affect 35 files at 79 locations. Based on the changes in the 17 files in the first of the above commits, `spdiff` derives the following generic patch:

```
kobject_unregister(X0)⇝kobject_put(X0)
```

The inferred generic patch fully updates all but 3 files in the same way the standard patch did. The remaining files each include an additional change that goes beyond the collateral evolution.

Modifying declarations Commit `c11ca97ee9d2ed593ab7b5523def7787-b46f398f` and 12 others from around December 7, 2007 change 21 files at 26 locations. The log messages are "use `LIST_HEAD` instead of `LIST_HEAD_INIT`". The standard patches total almost 300 lines. The inferred generic patch is:

```
struct list_head X0 = LIST_HEAD_INIT(X0);
  ⇝ LIST_HEAD(X0);
```

The inferred generic patch fully updates all 21 files. The original developer, on the other hand, initially overlooked one case and had to create a second patch on the same file to correct it. Furthermore, 6 files that contained relevant declarations at the time the patches were committed were not updated by the original patches, and of those 5 files were still not updated as of several months later. All of these files are fully updated by the generic patch.

Use `kzalloc` We conclude with an example that illustrates the limitations of our approach. Over the past couple of years, around 100 patches have been committed that convert the combination of calls to `kmalloc` and `memset` to `kzalloc`. One such commit, from September 6, 2005 is `dd3927105b6f65afb7dac176-82172cdfb86d3f00` which affected 6 files at 27 locations. The transformation it performs can be represented as follows.

```
x = kmalloc(size, flags);          ⇝  x = kzalloc(size, flags);
...                                   ...
memset(x, 0, size);
```

Our tool is, however, not able to infer any safe generic patch in this case. Indeed, in this case, it is not safe either to modify all of the calls to `kmalloc` or to drop all of the calls to `memset`. Instead, the transformation is only valid when the call to `kmalloc` precedes the call to `memset` and when they share the same expression `x`. The language of generic patches is not able to express the temporal ordering of terms nor the sharing of metavariables between disjoint code fragments, and thus it gives no answer in this case.

Assessment These examples show that for a variety of collateral evolutions, `spdiff` infers generic patches that are much more concise, and we believe much more readable, than the corresponding standard patches. In several cases, the original standard patches did not perform part of the collateral evolution in some relevant files. In this situation, a developer could use `spdiff` to infer a generic patch from the provided standard patches and to complete the collateral evolution. Using the Coccinelle transformation system, the generic patch can be applied everywhere in the Linux source tree.

While all of the inferred generic patches are simple enough that a person could construct them by hand by inspecting the standard patches, it would require more

work to confirm that the manually constructed patch is indeed safe for all of the input files. Safety is not evident from the standard patches which contain only the code that was changed, not any similarly structured code in the same files that was not changed. In order to confirm safety, one would need to apply the constructed patch to all original input files and check that for each file, the constructed patch applies *correctly* to a *subset* of the locations that need to be modified in the file.

Our final `kzalloc` example illustrates a limitation of generic patches. The richer language of semantic patches provided by Coccinelle can express the properties needed to treat such examples (Padioleau et al. 2008). We are currently extending `spdiff` to infer proper semantic patches (Andersen 2010).

7 Related work

Our approach considers the problem of finding a single generic patch that correctly updates a collection of programs. We know of no work that addresses this problem directly. Several approaches, however, have considered how to concisely capture the changes between the original and modified versions of a *single* program. In this section we relate our approach to a number of other approaches that detect program changes.

Chawathe et al. describe a method to detect changes to structured information based on an ordered tree and its updated version (Chawathe et al. 1996). Their goal is to derive a compact description of the changes. To this end, a notion of a minimum cost edit script is defined. An edit script is basically a sequence of operations where each operation has an associated cost determined by some measure of structural similarity between the trees. As such, the minimum cost edit script will be the most compact description of the changes made to the original tree with respect to the edit operations. Edit operations, however, always explicitly denote the node to transform and thus the approach is not sufficient for our context where we would like one transformation specification that applies to even unknown code.

Neamtiu et al. (2005) consider the problem of identifying changes to C programs. Their method infers changes, additions and deletions of various program elements based on structural matching of syntax trees. Two trees that are structurally identical but have differences in their nodes are considered to represent matching program fragments. In contrast to the work by Chawathe et al., each simple change (e.g. renaming of a variable) is only reported once. Thus, the description of the changes made can be more compact than what is possible with the minimum cost edit scripts of Chawathe et al. However, similarities in changes involving larger trees are not detected, and consequently very similar changes made across all functions are reported as separate changes, whereas we need to generalise arbitrary descriptions of changes.

Kim et al. (2007) propose a method to infer "change-rules" from two versions of the same program. Their goal is to construct a small set of change rules that capture many changes. Change rules express changes related to program headers (method headers, class names, package names, etc.). The basic shape of a change rule is similar to that of our term replacement patches: $\forall x \in \texttt{scope} : \texttt{transformation}$, meaning that every match described by the scope is modified by the transformation.

The scope, described using a variant of regular expressions, ranges over the textual representations of the previously mentioned headers. By using regular expressions as an abstraction mechanism the scope can be extended to e.g. all calls to a method that starts with the prefix `foo`. Thus, change rules can express that a given transformation was applied to a set of entities, which is more compact than simply enumerating all entities. Our term replacement patches are similar to change rules but apply to any program element rather than just to program headers. Finally, our use of metavariables allow equality constraints among program elements as well as applicability to more than one input program.

Weißgerber et al. present a technique to identify likely refactorings in the changes that have been performed in Java programs (Weissgerber and Diehl 2006). Like Kim et al., they search for a fixed set of transformation types (in this case, rename a method, add a parameter, etc.). Each transformation type has an associated precondition that enables the transformation. They first collect various signature information about the old and new versions of a given file, and then use this information to determine whether the precondition of any of the transformation types is satisfied. If a precondition is satisfied, the transformation is considered a refactoring candidate. They furthermore use clone-detection to check that the change performed by a candidate is semantics preserving. Because we consider arbitrary changes, such checks are not relevant in our case. The transformation types given by Weißgerber et al. do not support any kind of abstraction mechanisms such as our metavariables. Thus, two detected changes cannot be generalised into a more compact description that covers both of them, as could potentially be done by the method given by Kim et al. and by our work.

The `patchparse` collateral evolution mining tool (Padioleau et al. 2006) scans patch files for frequently occurring changes, modulo a simple strategy for abstracting away from terms that are shared between the original and modified code. `Patchparse` is sufficient to detect some of the rules in our examples, such as the example where calls to `kobject_unregister` was replaced by calls to `kobject_put`, for which it reports:

```
kobject_unregister(ARG0) replaced by kobject_put(ARG0)
```

However, its strategy for detecting common terms is essentially top-down, and thus it reports the following result for the `seq_path` example:

```
seq_path(ARG0, CODE, CODE, ARG3) replaced by seq_path(ARG0, CODE, ARG3)
```

In this case, `Patchparse` was not able to relate the second and third arguments in the original call to the second argument in the new code, and thus it falls back on characterizing these arguments as arbitrary code (`CODE`), which is not sufficient to specify the transformation. `Patchparse` furthermore does not ensure that the transformation represented by the proposed collateral evolution is either safe or compact. It can, however, be beneficially used, as we have done in this paper, to narrow down the set of patches considered when using `spdiff` to infer generic patches based on standard patches already submitted to Linux.

Aspect-oriented programming (Kiczales et al. 1997) can be viewed as specifying a form of transformation rule, which integrates so called "cross-cutting concerns" into a

program. Using our approach to infer aspects, however, would require that the aspect be manually added into the source program at once, so we could observe the appropriate changesets. It is not clear that programs that could benefit from aspect-oriented programming are created in this way, however; one could just as well imagine that the code that should be aspectified was introduced over time, and thus appropriate original and changed versions of the code would not be available to our approach. Similarly, specifications for static analysis tools such as Semmle/Code[4] could not be inferred using our approach, because no transformation is involved in that case. Our approach is indeed best suited for evolution or bug fixing, where multiple common changes are made at once.

8 Conclusion

The contributions of this paper are as follows: (1) We provide a formalisation of the largest common subpatch notion independent of the transformation language used. (2) We give an algorithm that infers safe and compact subpatches relative to a transformation language of generic patches as well as a tool that implements the algorithm. (3) We have shown examples of inferred generic patches from recent collateral evolutions in Linux where our tool infers generic patches that are more compact than the standard patches that were originally applied to the Linux source tree and that allow us to update relevant files where the collateral evolution was not performed.

Currently, the `spfind` tool requires that a change be made in all of the provided files. In practice, however, particularly when `spfind` is used to better understand a collection of existing standard patches, it can be useful to be able to detect changes that occur in only a subset of the files. We have addressed this issue by allowing the user to provide a threshold for the frequency of occurrences required for a change to be considered for inclusion in the generic patch.

Finally, the language of generic patches covers only a subset of the collateral evolutions performed in the Linux kernel. We have extended our method to a larger subset of the richer language of semantic patches provided by Coccinelle. Work is underway to document this extension (Andersen 2010).

Acknowledgements We would like to thank Gilles Muller for his feedback on an earlier version of this paper. We would also like to thank the anonymous reviewers for this version and the previous conference version of this paper for their helpful comments. And finally, this work was supported in part by the Danish Research Council for Technology and Production Sciences.

References

Andersen, J.: Semantic patch inference. Ph.D. dissertation, University of Copenhagen, Feb. 2010

Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 493–504. ACM Press, New York (1996)

[4] Semmle/code, http://semmle.com/semmlecode/.

Conchon, S., Filliâtre, J.-C.: Type-safe modular hash-consing. In: ACM SIGPLAN Workshop on ML, Portland, Oregon, September 2006, supersedes (Filliâtre, 2000). [Online]. Available: http://www.lri.fr/filliatr/ftp/publis/hash-consing2.ps

Filliâtre, J.-C.: Hash consing in an ML framework. LRI, Université Paris Sud, Research Report 1368, September 2000. [Online]. Available: http://www.lri.fr/~filliatr/ftp/publis/hash-consing.ps.gz

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: ECOOP'97—Object-Oriented Programming, 11th European Conference. Jyväskylä, Finland. Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer, Berlin (1997)

Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, pp. 333–343. IEEE Computer Society, Los Alamitos (2007)

MacKenzie, D., Eggert, P., Stallman, R.: Comparing and Merging Files With Gnu Diff and Patch, Network Theory Ltd., Jan. 2003, unified Format section, http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html

Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005)

Padioleau, Y., Lawall, J.L., Muller, G.: Understanding collateral evolution in Linux device drivers. In: The first ACM SIGOPS EuroSys Conference (EuroSys 2006), Leuven, Belgium, Apr. 2006, pp. 59–71

Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in Linux device drivers. In: Eurosys 2008, Glasgow, Scotland, Mar. 2008, pp. 247–260

Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, pp. 231–240. IEEE Computer Society, Los Alamitos (2006)