Generic Patch Inference

Andersen, Jesper; Lawall, Julia Laetitia

Published in:
Automated Software engineering 2008

DOI:
10.1109/ASE.2008.44

Publication date:
2008

Document version
Publisher's PDF, also known as Version of record

Generic Patch Inference

Jesper Andersen
DIKU, University of Copenhagen,
Universitetsparken 1, 2100 Copenhagen Ø, Denmark
Email: jespera@diku.dk

Julia L. Lawall
DIKU, University of Copenhagen,
Universitetsparken 1, 2100 Copenhagen Ø, Denmark
Email: julia@diku.dk

***Abstract*—A key issue in maintaining Linux device drivers is the need to update drivers in response 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 pairs 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 pairs of driver files.

I. 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 [1]. When making a change in the source code, a developer makes a copy of the 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 ideal for one kind of change, the *collateral evolution* [2]. 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 [2].

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* [3]. 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, such as 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 its 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 changes that are performed, as well as information about any deviations from these 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 that infers concise and abstract generic patches. We have implemented the algorithm `spfind` in a tool `spdiff`.
- We show examples of the generic patches inferred by `spdiff` from some recent collateral evolutions in Linux.

The rest of this paper is organized as follows. Section II presents a motivating example that illustrates some of the issues taken into account by our approach. Sections III through V 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 VI illustrates the application of our `spdiff` tool to various recent collateral evolutions performed in the Linux source tree. Section VII describes related work and Section VIII concludes.

II. 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. Furthermore, the line that is prefixed with ! has 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 and only one was affected and in the second file only one such call was affected.

A summary of the set of changes is shown in Figure 2. The summary reveals that although there are differences in how the two files were modified, there are also compelling similarities:

1) All calls to `memcpy` where the second argument is a reference to the field `data` 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 is *not* a reference to the field `data`.
2) The first argument becomes the second.
3) The field reference to `data` in the second argument is dropped. The resulting expression, which has type

[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`

Fig. 1. Extracts of the two files.

```
-  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);
```

Fig. 2. Set of changes for the two files of Figure 1.

`struct sk_buff *`, 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:

```
-    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. Intuitively, the above is an abstract representation of the changes made: in the context of a call to `memcpy` where the first and third argument are arbitrary expressions and the second references the `data` field, 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, 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.

As illustrated by this example, there are two main issues to be considered when inferring such 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, which we produce by replacing subterms that are not affected by the transformation by metavariables, as illustrated by *e.g.*, the use of `X0` rather than the concrete terms `skb_put(skbn,skbo->len)` (in `ax25_in.c`) and `ptr` (in `dn_rtmsg.c`) in the generic patch in the above example.

Safety: The safety of a generic patch requires that only things that 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 a structural restriction on the second argument to `memcpy`: only those calls where the second argument referenced the `data` field should be affected.

In the next two sections we develop the machinery needed in order to state an algorithm that can automatically infer safe and compact generic patches such as the one shown above. In the example, there was only one change, but the method we describe is capable of dealing with multiple changes, and always ensures that the derived generic patch correctly describes a transformation that applies to all the files given as input.

III. Setup

While our approach targets C code, we formalise it in terms of a simpler language, which we call the language of Terms. This language only distinguishes between atomic and compound terms, as is sufficient for the presentation of the algorithm. The language is defined as follows:

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.

Updates on terms are described by generic patches. 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 the metavariables for terms such that applying the substitution to the pattern yields a term 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: Patterns are then combined into term replacement patches. A term replacement patch describes how to transform any (sub)terms that match a given pattern. The syntax of a term replacement patch is as follows:

Definition 3 (Syntax of Term Replacement Patches)

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

The application of a term replacement patch to a term is defined by the rules shown in Figure 3.[2] Rule a is concerned with the case where a term t matches the pattern p according to some substitution θ. The matching term is replaced with $\theta p'$. The remaining rules traverse the term top-down along all branches of the term (rule b) until reaching a subterm at which rule a applies or until reaching a leaf (rule c). (Note that rules b and c only apply if rule a does not.) If there is no matching subterm, the application of a term replacement patch behaves as the identity function.

[2]Note that although term replacement patches have the form of rewrite rules, they are not applied iteratively as done in term rewriting systems.

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

$$(b) \quad \frac{(p \rightsquigarrow p')(t_i) = t'_i, f_i \text{ for all } 0 \leq i \leq n \qquad F = \bigsqcup f_i}{(p \rightsquigarrow p')(a(t_0, \ldots, t_n) = a(t'_0, \ldots, t'_n), F}$$

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

Fig. 3. Application of a term replacement patch.

The application of a term replacement patch additionally returns a flag $\top$, when a match has been found, or $\bot$, when no match has been found. These are ordered as $\bot \sqsubseteq \top$. Note that a match may not actually cause the generated term to be different from 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 equivalent.

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.*

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

Subsequently, whenever we say "patch," we mean generic patch unless stated otherwise.

The rules for applying a generic patch are shown below. The application of a term replacement patch $p_1 \rightsquigarrow p_2$ is defined in terms of the rules of Figure 3, 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 the result of that application is used as input to the application of gp_2.

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

IV. 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. 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.

A. 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 transformation has been made in t to turn it into t''. A patch gp is safe relative to a pair of terms when the transformations it makes when applied to t are a part of those that turn t into t''. 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:

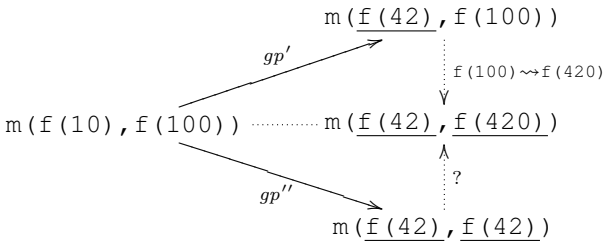
$$\begin{array}{lcl} t & = & \texttt{m(f(10),f(100))} \\ t' & = & \texttt{m(f(42),f(100))} \\ t'' & = & \texttt{m(f(42),f(42))} \\ t''' & = & \texttt{m(f(42),f(420))} \end{array}$$

and the following patches:

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

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 gp' and gp'' to t, where subterms that have been affected by a patch are underlined, and the relation to the result, t''' (center right).



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 *undo* some of the changes made by gp'', *i.e.*, the second *underlined* occurrence of `f(42)` in t'' (bottom right). Thus, gp'' is not safe relative to (t, t''').

The definition of safety is based on the observation that in transforming t into t'' it is not necessary to modify any subterm more than once. Thus, if t' is the result of applying gp to t, then any subterm of t that is changed in t' should not be changed again, and any subterm of t that is not changed in t' can be changed in order to reach the third term t'''. The rules in Definition 5 capture this property. We then say that gp is safe relative to (t, t'').

Definition 5 (One-step reachable) *Suppose a patch gp and a pair of terms (t, t'') are given. Assume that $[\![gp]\!]t = t'$ for some t'. We say that t'' is one-step reachable if t'' can be obtained from t' without modifying any of the subterms that gp modified with respect to t. We denote this $t \to t' \to t''$.*

$$\frac{t = t' \vee t' = t''}{t \to t' \to t''}$$

$$\frac{\begin{array}{c} a(ts) \neq a'(ts') \qquad a'(ts') \neq a''(ts'') \\ a = a' \vee a' = a'' \\ \forall (t, t', t'') \in (ts, ts', ts'') : t \to t' \to t'' \end{array}}{a(ts) \to a'(ts') \to a''(ts'')}$$

Based on Definition 5 we define a *safe transformation part* of a pair of terms (t, t'') as a patch such that any affected subterm of t is affected in a safe manner.

Definition 6 (Safe Transformation part)

$$gp \preceq (t, t'') \iff \forall t' : [\![gp]\!]t = t' \Rightarrow t \rightarrow t' \rightarrow t''$$

Common patch: A patch is said to be common to a set of pairs of terms C if it is safe relative to each of the pairs of C.

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

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

B. 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 a 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' if and only if gp performs a safe part of the transformation that gp' performs, as expressed by $gp \preceq (t, t')$, and gp' is a safe part of (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 2 (Compactness and subpatches) In this example we show that 1) when a patch gp' is a larger than gp it can actually be *syntactically* smaller than gp and that 2) larger patches can be more compact by covering the changes expressed by several smaller patches.

Consider the following terms:

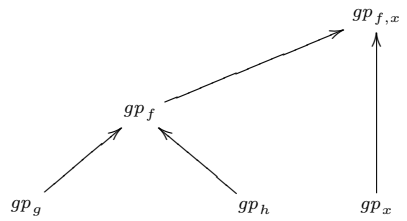
t_f = `f(1)`
t_g = `g(f(1))`
t_h = `h(f(1))`
t_x = `x(f(2))`
t'_f = `f(1,1)`
t'_g = `g(f(1,1))`
t'_h = `h(f(1,1))`
t'_x = `x(f(2,2))`

t_m = `m(g(f(1)),h(f(1))+x(2))`
t'_m = `m(g(f(1,1)),h(f(1,1)+x(2,2)))`

as well as the following patches:

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

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.



From the hierarchy it is evident 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 the largest patch and that it covers both the transformations made by gp_f and gp_x. Therefore it is more compact than the combination of the two smaller patches.

The largest common subpatch: Suppose C is a set of pairs of terms $\{(t_0, t''_0), ..., (t_n, t''_n)\}$ representing the original and updated code after some manual collateral evolution has been performed. A largest 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 common safe 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 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 3 (Largest Common Subpatch) The following is an example of a largest common subpatch for a set of pairs of terms $\{(t_1, t'_1), (t_2, t'_2)\}$. This example shows that the largest common subpatch need not be unique.

t_1 =

```
void 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 =

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

t'_2 =

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

The changes made to the two terms 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 C, we can verify that the set $LCP(C)$ consists of exactly two generic patches:

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

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.

The example above motivates the following theorem of equivalence of the set of largest 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:*

$$\begin{array}{ll} \forall C : \forall gp, gp' : & gp \in LCP(C) \Rightarrow \\ & gp' \in LCP(C) \Rightarrow \\ & (t_i, t_i'') \in C \Rightarrow \\ & [\![gp]\!]t_i = t_i' \wedge [\![gp']\!]t_i = t_i' \end{array}$$

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 $B/_\sim$ (where $\sim$ denotes equivalence of generic patches with respect to C) and the subpatch ordering $\preceq_C$ $(C/_\sim, \preceq_C)$ form a join semi-lattice and that the least upper bound is in fact the set $LCP(C)$. Since any element of $B/_\sim$ is a set of equivalent generic patches, theorem 1 follows. ∎

Monotonicity: Given a set of pairs of terms C, suppose B is such that $LCP(C) = B$. That is, B is the set of largest common subpatches for 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 transformation in the pairs added to C has nothing in common with those in C, or 2) the transformations in $LCP(C)$ are not *safe* for the new pairs of terms. In this case the transformation found in $LCP(C)$ is applied in the new pairs, but the transformation language is not expressive enough to make a safe description of the transformation.

```
 0:  simple_pairs C =
 1:    B := I := ∅;
 2:    foreach (ti,ti'') ∈ C:
 3:      Bi := {p⇝p' | p⇝p'⪯(ti,ti'')};
 4:      B := Bi ∪ B;
 5:    foreach Bi ∈ B:
 6:      I := Bi ∩ I;
 7:    return I

 8:  gen (C, bps, current_bp) =
 9:    next := next_bps (C, bps, current_bp);
10:    if next = ∅
11:    then
12:      return {current_bp};
13:    else
14:      foreach bp ∈ next:
15:        nbp := current_bp;bp;
16:        Si := gen (C, next, nbp);
17:        S := Si ∪ S;
18:      foreach Si ∈ S:
19:        R := Si⋆R
20:      return R;

21:  let spfind C =
22:    return gen (C, simple_pairs(C), ⊥)
```

Fig. 4. The algorithm behind `spfind`.

V. The spfind Algorithm

In this section we present the algorithm `spfind` on which our tool is based. Given a set of pairs of terms, this algorithm infers a collection of largest common subpatches, as defined in the previous section.

Overview of the algorithm: The algorithm follows a generate-and-test scheme. 1) For each pair of terms, we construct the set of all possible parts that have the form of term replacement patches (*i.e.*, not sequential patches). Next, we collect the term replacement patches that occur in the intersection of all of the generated sets of term replacement patches for every pair of terms. 2) Finally, we combine the term replacement patches in this intersection into larger sequential patches to obtain largest common subpatches for the given set of term pairs. As discussed in the previous section, there can be more than one largest common subpatch.

Using pseudo-code, Figure 4 states the essential algorithm behind our tool. The algorithm is split into two parts: 1) generation of term replacement patches in the function `simple_pairs` (lines 0-7) and 2) combination of patches into larger ones in the function `gen` (lines 8-20). In the following we describe these functions.

A. Generating term replacement patches

The key part of `simple_pairs` is the set comprehension in line 3, which uses the safe transformation part relation $gp \preceq C$, to construct a set of term replacement patches.

Generating and abstracting over term replacement patches: In order to implement the set comprehension described in line 3 of `simple_pairs` we need a way to generate all possible term replacement patches for a pair of terms. To do this, the algorithm first finds all the term replacement patches without metavariables (concrete parts) and then generalises each concrete part in every possible way by introduction of metavariables.

To generate all the concrete parts for a pair of terms (t, t'') the algorithm simply traverses the two terms in parallel. For each pair of subterms it returns the concrete part $t_l \rightsquigarrow t_r$ only when $t_l \rightsquigarrow t_r \preceq (t, t'')$, using Definition 6 to perform the part-of check.

For each concrete part $t_l \rightsquigarrow t_r$, the algorithm then generates a set of generalised patches by introducing metavariables for subterms of t_l and t_r respectively such that $\exists\theta : \theta p = t_l$ and $\theta p' = t_r$ and $p \rightsquigarrow p' \preceq (t, t'')$. The pseudocode for introducing metavariables is tedious but straightforward and consequently not shown. As an example, however, consider the concrete term replacement patch `f(42,117)` $\rightsquigarrow$ $f(42, 117 + 42)$. Below we show a non exhaustive list of possible abstracted versions:

```
f(42,X0) ⇝ f(42,X0+42)
f(X0,117)⇝ f(42,117+X0)
f(X0,X1) ⇝ f(X0,X1+X0)
   ...
X0(X1,X2)⇝ X0(X1,X2+X1)
```

Intersection: The final step of `simple_pairs` is to take the intersection of all the generated sets of term replacement patches. Because the generalization process ensures that all metavariables have been generated systematically, if `I` is the intersection of all sets of term replacement patches generated as described above, then it holds that:

$$\forall gp \in I : gp \preceq \mathtt{C}$$

where `C` is the set of pairs of terms input to the `simple_pairs` function. Recall that the definition of $gp \preceq$ `C` simply requires that *gp* is a safe transformation part for each pair in `C`. Since `I` is the intersection of all the sets of term replacement patches constructed from each pair of terms in `C` ($\mathtt{B}_i$ in line 6), we can conclude that $\forall(t, t') \in \mathtt{C} : gp \preceq (t, t')$ as required.

B. Creating sequential patches

The `gen` function (lines 8 to 19 in Figure 4) constructs larger patches based on the term replacement patches returned by the `simple_pairs` function.

A technically correct version of the `gen` function would simply generate all possible permutations of the term replacement patches given, and then extract the largest among the constructed patches. This approach, however, is unnecessarily inefficient. Instead, the algorithm iteratively constructs larger patches by extending simple patches into sequential ones and removing smaller ones when combining sets of results. To remove smaller patches, the algorithm uses Definition 8 for $gp \preceq_C gp'$.

Finding possible next patches: Given a set of pairs of terms, a prefix patch (denoted `current_bp` in Figure 4) and a set of term replacement patches, the function `next_bps` returns the set of patches that can safely be applied *after* `current_bp`. For each of these possible next patches (denoted `bp`), the algorithm extends `current_bp` into `current_bp;bp` (in line 15) and recursively calls `gen` with the extended patch. If there are no more next patches to apply, the algorithm simply returns the singleton set consisting of the current prefix patch. The function `next_bps` is defined as:

```
let next_bps (C, bps, current) =
   { bp ∈ bps | current;bp ≼ C }
```

Joining sets of patches: Given sets of constructed patches, lines 18-19 combine them to obtain a resulting set that has the property that all patches in the set are largest:

$$\mathtt{S} \star \mathtt{S'} = \{gp \in S \cup S' \mid \forall gp' \in S \cup S' : gp' \preceq_C gp\}$$

Correctness of algorithm: We now state the relation between the set of largest common subpatches $LCP(C)$ and the algorithm, although a formal proof of correctness is beyond the scope of this paper.

The algorithm given in Figure 4 is sound and complete with respect to the definition of set of largest common patches. Given a set of pairs of terms C, $\mathtt{spfind}(C) \subseteq LCP(C)$ and $LCP(C) \subseteq \mathtt{spfind}(C)$.

The implementation: We have implemented the algorithm in a tool called `spdiff`.[3] Because LCP(C) can be large and involve uninteresting or overly generic variants, we have desiged the tool to be sound but not complete, although the tool always produces non-empty results when $LCP(C) \neq \emptyset$. In the following we consider in more detail what these uninteresting or overly generic variants may be.

In Example 3 we have already seen a case where $LCP(C)$ contains multiple elements because of differences in the order of the constituent term replacement patches. $LCP(C)$ can also be very large when there are too many opportunities for generalization. Given a set C of pairs of terms, the set $LCP(C)$ may be very large in case the terms in C are structurally very similar. To see why, suppose the following generic patch is in the set $LCP(C)$:

```
foo(bar(w+x),117+y)
⇝ foo(bar(w+x),117+y) + z
```

The essential part of the change is that in the right hand side, a `z` is added. Potentially, however any subexpression can be abstracted, so there are $O(n^2)$ abstractions, *gp*, of the generic patch such that $gp \in LCP(C)$. Given that the above generic patch is in the set $LCP(C)$, it may be that the following generic patches are also in this set (recall that metavariables are written as uppercase letters).

```
foo(X(w+x),117+y)   ⇝   foo(X(w+x),117+y) + z
X(bar(w+x),117+y)   ⇝   X(bar(w+x),117+y) + z
X(bar(w+x),117+Q)   ⇝   X(bar(w+x),117+Q) + z
          X(Y,Q)    ⇝   X(Y,Q) + z
                    ...
```

Although `X(Y,Q)` $\rightsquigarrow$ `X(Y,Q)+z` is in the set $LCP(C)$ it is not very likely that this generic patch is going to be safe to apply to an unknown input term, because it specifies no concrete information about either the function that is applied or its arguments. Thus, the tool discards such a result, if a more informative result is available.

[3] Binary available as http://www.diku.dk/~jespera/spdiff.opt

Based on the observation expressed in Theorem 1 that all generic patches in the set $LCP(C)$ perform the same transformation to all given terms, the tool can return a subset $LCP(C)$ of generic patches that seem *likely* to also apply correctly to unknown terms. Concretely, define the size of a generic patch $|gp|$ as the size of each pattern it contains, where the size of a pattern is given by:

$$\begin{array}{lcl} |\text{ATOM}| & = & 1 \\ |\text{ATOM}(t_0,...,t_n)| & = & 1+|t_0|+\cdots+|t_n| \\ |Meta| & = & 0 \\ |Meta(t_0,...,t_n)| & = & |t_0|+\cdots+|t_n| \end{array}$$

The size of a pattern is thus the number of ATOMs in the pattern. Let $\texttt{spdiff}(C)$ denote the result of running the tool with input C. The relation between the result of $\texttt{spdiff}(C)$ and $LCP(C)$ is then

$$\begin{array}{l} \texttt{spdiff}(C) = R \wedge LCP(C) = B \Rightarrow \\ R \subseteq B \wedge \\ \quad R = \{gp \mid gp \in B, \forall gp' \in B : |gp'| \leq |gp|\} \end{array}$$

The size of the previous generic patch is then:

$$\begin{array}{l} |\texttt{X(Y,Q)} \rightsquigarrow \texttt{X(Y,Q)+z}| = \\ \quad |\texttt{X(Y,Z)}| + |\texttt{X(Y,Z)+z}| = 0+1 = 1 \end{array}$$

The other generic patches listed above have a size that is strictly larger than 1. Thus, the above generic patch is not part of the set returned by the tool.

VI. 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 [2]. 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".[4]

```
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 a person could construct the inferred generic patch by hand. However, there would be no guarantee that the constructed patch is safe, as this is not evident in the standard patch. To check safeness 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.

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 `c32c2f63a9d6c953aaf168c0b2551da9734f76d2` 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 `c11ca97ee9d2ed593ab7b5523def7787b46f398f` 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 have still not been updated today (four months later). All of these files are fully updated by the generic patch.

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

Use `kzalloc`: 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 is `dd3927105b6f65afb7dac17682172cdfb8-6d3f00` from September 6, 2005 which affected 6 files at 27 locations. The transformation it performs can be represented as follows.

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

Our tool is, however, not able to infer any common generic patch in this case because the language of generic patches is not able to express the temporal ordering of terms or the sharing of metavariables between disjoint code fragments.

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 inferred are all 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. Safeness is not evident from the standard patches, so to confirm safeness, 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 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 [3]. We plan to extend `spdiff` to treat such cases in the future.

VII. 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 programs changes.

Chawathe et al. [4] describe a method to detect changes to structured information based on an ordered tree and its updated version. 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 similarities 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. consider the problem of identifying changes to C programs [5]. Their method infers changes, additions and deletions of various program elements based on structure 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 be able to generalise descriptions of changes.

Kim et al. propose a method to infer "change-rules" from two versions of the same program [6]. 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: $\forall x \in$ `scope` : `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 [7]. Like Kim et al., they search for a fixed set of transformation types (in this case, rename method, add 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 preconditions 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 can not 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 [2] 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 `kobject_unregister/kobject_put` example, 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, it 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 `spfind` to infer generic patches based on standard patches already submitted to Linux.

VIII. 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 performs inference of largest common 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 allowed 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 plan to address this issue in the near future, *e.g.*, 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 can cover only a subset of the collateral evolutions performed in the Linux kernel. In the future we plan to extend our method to the richer language of semantic patches provided by Coccinelle.

IX. Acknowledgments

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 helpful comments. And finally, this work was supported in part by the Danish Research Council for Technology and Production Sciences.

References

[1] D. MacKenzie, P. Eggert, and R. Stallman, *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`.

[2] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in Linux device drivers," in *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, Leuven, Belgium, Apr. 2006, pp. 59–71.

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

[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1996, pp. 493–504.

[5] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[6] M. Kim, D. Notkin, and D. Grossman, "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: IEEE Computer Society, 2007, pp. 333–343.

[7] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–240.