Semantic Patch Inference

Jesper Andersen[*]
DIKU
University of Copenhagen
jespera@diku.dk

Anh Cuong Nguyen
Dept. of Computer Science
National University of Singapore
anhcuong@comp.nus.edu.sg

David Lo
School of Information Systems
Singapore Management University
davidlo@smu.edu.sg

Julia Lawall
INRIA/LIP6-Regal
Julia.Lawall@lip6.fr

Siau-Cheng Khoo
Dept. of Computer Science
National University of Singapore
khoosc@comp.nus.edu.sg

ABSTRACT

We propose a tool for inferring transformation specifications from a few examples of original and updated code. These transformation specifications may contain multiple code fragments from within a single function, all of which must be present for the transformation to apply. This makes the inferred transformations context sensitive. Our algorithm is based on depth-first search, with pruning. Because it is applied locally to a collection of functions that contain related changes, it is efficient in practice. We illustrate the approach on an example drawn from recent changes to the Linux kernel.

1. INTRODUCTION

Software frequently makes use of libraries in order to implement common tasks. When a library is updated, the interface that it exposes may change as well. A change in the interface may in turn necessitate updates in all code that uses the library. Such updates have been given the name *collateral evolutions* [16], and have been studied in the context of Linux code. They involve changes such as renaming called functions, adding and removing arguments, reorganizing structures, and changing usage protocols. To perform a collateral evolution, a software maintainer must first understand the evolution and then consider how it applies to his code. Collateral evolutions thus entail a major software maintenance burden.

To ease the task of understanding and applying collateral evolutions, Padioleau et al. have proposed a notion of *semantic patches* [15]. A standard patch is a description of how to transform a piece of software, in terms of the lines to add and remove [12]. Such patches can be generated automatically from the original and updated versions of each modified source file. They are simple, immediately applicable, and familiar to software developers. Nevertheless, they are limited to describing a change at specific lines in the software. Semantic patches generalize standard patches by abstracting over details such as comments and whitespace, as well as over irrelevant subterms. A single semantic patch can thus be used to update many code sites. As such, a semantic patch can be used both to provide formal documentation of the transformations needed in a collateral evolution and to implement the collateral evolutions itself.

[*]Current address: Endomondo ApS
jesper.andersen@endomondo.com

Despite their advantages, constructing a semantic patch may be a stumbling block for the developer who makes a change in a library interface, because it requires that the developer think about code changes in an abstract way. To reduce this burden, we have developed the tool `spdiff` to infer some kinds of semantic patches from a small collection of manually performed modifications in the source code. `spdiff` thus supports the current development process, but requires the developer only to perform the collateral evolution in a few typical files. The inferred semantic patch can then be applied to the remaining files, and published for use by other developers whose code may be affected by the collateral evolution. Previous work on `spdiff` is limited to inferring semantic patches that amount to a sequence of context-free atomic transformations, which we refer to as *generic patches* [1]. Since then, we have extended this approach to generate a larger class of semantic patches, describing the relationship between multiple disjoint terms that determines how the overall transformation should be performed.

We present the extension in this demo. After giving an overview of the principles behind `spdiff` (Section 2), we present an example (Section 3). We next briefly assess the relationship to related work (Section 4), and then conclude (Section 5).

2. PRINCIPLES

Semantic patches generalize patches by using *metavariables* to represent arbitrary subterms and dots ("...") to represent arbitrary code sequences. Conceptually, our semantic patch inference process raises the level of abstraction of a collection of standard patches, by introducing metavariables and dots in order to obtain semantic patches that satisfy the following properties. Given a *change set*, $CS = \{(t_1, t'_1), ..., (t_n, t'_n)\}$, consisting of a set of pairs of terms, the resulting semantic patches should be:

- *Safe*, in that the resulting semantic patches can be applied to all left hand side terms $\{t_i \mid (t_i, t'_i) \in CS\}$ such that the result for each t_i is related to the corresponding right hand side term t'_i.

- *Concise*, in that the resulting semantic patches should capture as much of the common changes across all of the pairs of terms in CS as possible.

These properties are designed to account for the fact that it may not be possible to find *concise* semantic patches that completely and *safely* implement all of the transformations indicated by CS. In this case, the transformations expressed by the resulting semantic patches may be partial.

We have previously shown how to infer safe and concise semantic patches that perform only atomic transformations [1]. In this work, we extend these results to infer transformations that depend on the presence of multiple code fragments that are connected by paths in a function's control-flow graph. The inference algorithm is divided into two steps: 1) Find *maximal patterns* than are common to the source terms t_i, and 2) Integrate transformation specifications into these semantic patterns, such that the resulting semantic patches implement the transformations represented by the change set CS.

2.1 Finding maximal semantic patterns

A semantic pattern is a term in which subterms and sequences of subterms may be abstracted as metavariables and dots ("...''), respectively. A semantic pattern is thus like a semantic patch but with no added or removed lines. Semantic patterns are ordered such that a semantic pattern is related to all semantic patterns that have the same structure but are possibly more abstract.

The inferred semantic patterns all have the form $Pt(\ldots Pt)^*$, where Pt is a fragment of code that may contain metavariables. The *length* of such a semantic pattern is the number of terms Pt. The inference of semantic patterns proceeds in three steps: 1) Incrementally infer the $Pt_1 \ldots Pt_n$, starting from length 1 and increasing to longer lengths; in these semantic patterns all metavariables are unique, 2) Remove non-maximal patterns according to the above ordering, and 3) Merge metavariables within the inferred maximal patterns to express common subterms. Resulting patterns are required to match at least th terms t_i in the given changeset CS, where the threshold th is provided by the user.

Pattern extension involves concatenating length 1 patterns to the end of an existing semantic pattern. Several pruning strategies are employed to ensure termination and reduce complexity. If a pattern does not match at least th fragments in the source code, it is not extended further, as any extension would also have insufficient matches. A pattern is also not extended if a super-pattern matching the same fragments of code has been considered before. If a pattern cannot be extended further, we add it to the set of candidate patterns, R. Because patterns are extended by concatenation at the end, R may end up containing patterns that are suffixes of other patterns in R. Removing these suffixes yields the set of maximal patterns R_{max}.

Finally, the metavariables of the elements of R_{max} are merged as much as possible, by assigning a common variable name to the maximal set of metavariables that always map to the same terms in the change set CS under consideration. The result is then fed into the next step: *patch construction.*

2.2 Constructing patches from patterns

Once a set of maximal semantic patterns has been found, the next step is to extend them with transformation information to construct semantic patches. In order to do so, we first extract small bits of change information, referred to as *chunks*, from the change set by considering the differences between the control flow graphs of each pair of terms in the change set. A chunk basically indicates whether a particular node of a control flow graph of a term was removed or if code was added before or after the node in the corresponding control flow graph of the updated term. Given a set of chunks and semantic patterns, we construct semantic patches by iteratively trying to pair a chunk with a semantic pattern to see whether the resulting semantic patch is common to enough pairs in the change set. If this is the case, the resulting semantic patch is added to a work-queue to allow it to be extended with further chunks. A semantic patch containment relation allows the algorithm to return only the maximal semantic patches. These maximal semantic patches satisfy the *safety* and *conciseness* properties mentioned above.

3. REAL-WORLD USAGE

`Spdiff` is most useful when a developer wishes to extract a context-sensitive semantic patch from the result of applying a single collateral evolution of a set of code fragments. The developer can then use the patch to upgrade other code that is affected by the same collateral evolution. We simulate this scenario by using `spdiff` to derive a semantic patch from a few pairs in a given change set. We then apply the patch to the rest of the change set and assess the correctness of the results.

The original commit. The commit with `git` SHA1 identification code 01f2705daf5a36208e69d7cf95db9c330f843af6[1] was the first in a series of standard patches committed to Linux starting in May 2007 that implemented a refactoring of a commonly occurring pattern for clearing pages found in filesystem code. This pattern consisted of the following operations: 1) map a page into the kernel virtual memory using `kmap_atomic`, 2) clear this page using `memset`, 3) call `flush_dcache_page` to ensure that the cleared memory gets written to disk, and 4) unmap the memory region using `kunmap_atomic`. The refactoring introduced a new macro, `zero_user_page`, that does all of these operations. In the above commit, core kernel functions where memory was cleared in this way were modified to use the new macro. In subsequent commits, the remaining functions were updated similarly.

The inferred semantic patch. We focus on 8 pairs of original and updated functions that were modified by the above commit. We first randomly selected 4 pairs of original and updated functions from the commit to create a change set. The selected functions are `do_direct_IO`, `xip_truncate_page`, `memclear_highpage_flush` and `do_lo_send_aops`. Running `spdiff` with the above change set produces the semantic patch shown below.

```
@@
expression X0, X1;
struct page *X2;
```

[1]http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git

```
char *X3;
@@
- X3 = kmap_atomic(X2, KM_USER0);
  ...
- memset(X3 + X0, 0, X1);
  ...
- kunmap_atomic(X3, KM_USER0);
+ zero_user_page(X2, X0, X1, KM_USER0);
```

Note that the generated semantic patch does not include the call to `flush_dcache_page`. This call occurs in varying positions relative to the call to `kunmap_atomic`, and is even absent in one case, and thus it is excluded from the initial semantic pattern. Thus, it does not occur in the generated semantic patch.

Applying this semantic patch to the files mentioned in the original commit causes it to perform a safe part of the changes that were made by hand, in 7 out of 8 cases. The remaining case is represented by the following excerpt of the standard patch:

```
@@ -2108,10 +2100,8 @@ int cont_prepare_write(

- kaddr = kmap_atomic(new_page, KM_USER0);
- memset(kaddr+zerofrom, 0, PAGE_CACHE_SIZE-zerofrom);
- flush_dcache_page(new_page);
- kunmap_atomic(kaddr, KM_USER0);
+ zero_user_page(page, zerofrom, PAGE_CACHE_SIZE - zerofrom,
+                KM_USER0);
```

In this standard patch, the developer has introduced a bug. The error is that in the added code at the end of the standard patch, the first argument to `zero_user_page` is `page`, while, as shown by the other calls, it should have been `new_page`. The updated function can indeed still be compiled because the variable `page` is a parameter of the function being modified. At run-time, however a file system corruption occurs. This fault is described and fixed in the commit ff1be9ad61-e3e17ba83702d8ed0b534e5b8ee15c, which was submitted by another developer, 11 days after the bug was introduced.

This error would have been avoided if the change were made using the semantic patch, because the semantic patch specifies that the first argument to the newly inserted function call should be the same as the first argument to the call to `kmap_atomic`. Since in *all* other updated functions the name of the variable given as the first argument to `kmap_atomic` is indeed `page`, the bug is probably resulted from careless copy-paste editing [9]. Linux code indeed frequently, but not always, uses stereotypical names for values of a given type, and thus there is a high potential for this sort of error.

This example also demonstrates that the inferred semantic patch can be much more concise than the corresponding standard patch derived from manual changes. First, by using a semantic patch, the developer needs to derive only one patch for the entire collection of files, rather than making the change in every file in the collection. In addition, the semantic patch captures only the changes that are common across all the pairs in the change set. In this example, the complete inferred semantic patch consists of only 6 lines of code, while every change in the original commit is represented by at least 10 lines.

4. RELATED WORK

Our inference of semantic patterns is closely related to clone detection [4, 6, 7, 8, 10], as has become widely available in refactoring tools. A clone is a contiguous or nearly contiguous block of code found to occur, often modulo inessential elements such as constants or variable names, at least twice in a program. Much of the work on clone detection has focused on introducing approximations to improve performance, in order to allow the approach to scale to very large programs. Our inference of semantic patterns on the other hand, cannot use such approximations, because the result is intended to be used directly as part of a transformation specification. The complexity of our semantic pattern detection is thus greater than that of scalable clone detection, but this is compensated for by the fact that the data size is much smaller, as the intended usage scenario is that it is given only the functions in which the developer has manually made some changes of interest. While most clone detection strategies focus on contiguous terms in the program syntax, Gabel et al. [4] consider contiguous subgraphs of a program dependence graph (PDG). Their approach can skip over auxiliary code such as debugging statements that is mixed with the clone code at the source code level. `Spdiff`, in contrast, finds arbitrarily separated fragments of common code, regardless of dependencies between the common code and the intervening code.

There are a number of works on a pattern-mining based approach to inferring specifications from a piece of software or a set of clients accessing a common library [5, 11, 17]. `Spdiff`, does not identify common patterns within a single version of the software, but instead extracts a description of the differences between two versions of the software, to produce patches.

Chawathe et al. [2] describe a method to detect changes in structured information based on a ordered tree representation of the original and update version. Their goal is to derive a compact description of the changes between the original and updated tree. To this end, a notion of a minimum cost edit script is defined. An edit script is 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 possible. 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 many programs and potentially even unknown code.

Neamtiu et al. infer changes, additions and deletions of various elements of C programs based on structural matching of syntax trees [14]. 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. [2], 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 `spdiff` generalizes descriptions of changes.

`Sydit` [13] infers change rules from a single example of original and updated code. Code on which the changed terms are control or data dependent can optionally be included, according to various strategies, to make the resulting rules context sensitive. Nevertheless, because only one example of the change is provided, the abstraction and context-inclusion strategies are necessarily fixed *a priori*, and without awareness of the actual requirements of the transformation. Thus, it was found that including more context was often detrimental, because rules became overspecified, reducing their applicability. In contrast, `spdiff` has available several examples of a change, and thus can adapt the degree of abstraction and the amount of included context information according to the needs of each provided change set, taking into account the safety and conciseness properties presented in Section 2.

5. CONCLUSION

Much effort in program differencing has been devoted to the detection of changes between different versions of the same program [2, 3, 14]. In contrast, our `spdiff` tool detects common changes in a *collection* of program fragments resulting from one collateral evolution. As a result, the semantic patch generated by `spdiff` can be used to upgrade other code that is affected by the same collateral evolution. As illustrated by our example, an inferred semantic patch can be much more concise than the corresponding standard patch. Semantic patches are also useful to reason about common changes.

The full semantic patch language makes it possible to express multiple interdependent rules that can affect code scattered across multiple functions. We leave the problem of inferring such rules to future work.

Availability. `spdiff` is available at:
`http://www.diku.dk/hjemmesider/ansatte/jespera`.

6. REFERENCES

[1] J. Andersen and J. L. Lawall. Generic patch inference. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, L'Aquila, Italy, sep 2008. IEEE.

[2] 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*, pages 493–504, New York, NY, USA, 1996. ACM.

[3] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, November 2007.

[4] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 321–330, Leipzig, Germany, 2008.

[5] L. Z. Hao Zhong and H. Mei. Inferring specifications of object oriented APIs from API source code. In *Proceedings of 15th Asia-Pacific Software Engineering Conference (APSEC'08)*, pages 221–228, 2008.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, pages 96–105, Minneapolis, MN, May 2007.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[8] H. Li and S. Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM'09)*, pages 169–178, Savannah, GA, USA, 2009.

[9] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, San Francisco, CA, Dec. 2004.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[11] D. Lo, S. C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, San Jose, California, USA, 2007.

[12] 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`.

[13] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, San Jose, CA, USA, June 2011.

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

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

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

[17] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 84–88, 2006.

APPENDIX

A. DEMONSTRATION PROCEDURE

To demonstrate the usability of `Spdiff`, we will use four change sets extracted from Linux kernel. During the demonstration, we will show by examples how these change sets can be converted into semantic patches. We highlight some possible examples that will be presented in the demonstration in the following paragraphs. In each of these examples, Spdiff infers a semantic patch within one second.

The first change set adds ata_pad_free before kfree. We have three pairs of code in the change set as shown in Figure 1. Spdiff infers a semantic patch as shown in Figure 2.

```
@@ -3,7 +3,6 @@
  struct device *dev = ap->host->dev;
  struct sil24_port_priv *pp = ap->private_data;
  sil24_cblk_free(pp, dev);
-ata_pad_free(ap, dev);
  kfree(pp);
 }
  static void mv_port_stop(struct ata_port *ap)
@@ -15,7 +14,6 @@
  mv_stop_dma(ap);
  spin_unlock_irqrestore(&ap->host->lock, flags);
  ap->private_data = NULL;
-ata_pad_free(ap, dev);
  mv_priv_free(pp, dev);
  kfree(pp);
 }
 @@ -34,7 +32,6 @@
   ap->private_data = NULL;
  dma_free_coherent(dev, AHCI_PORT_PRIV_DMA_SZ,
    pp->cmd_slot, pp->cmd_slot_dma);
-ata_pad_free(ap, dev);
  kfree(pp);
 }
```

Figure 1: Change Set 1

```
[Main] *REAL* semantic patches inferred: 1
[spatch:]
@@
X0 *  X1;
struct device *  X3;
expression X1;
X0;
expression X3;
struct ata_port *  X2;
X9;
X2;
@@
  head:def[void X9 ( struct ata_port *  X2) {};]
  ...
  struct device *  X3 = X2->host->dev;
  ...
  X0 *  X1 = X2->private_data;
  ...
- ata_pad_free(X2,X3);
  ...
  kfree(X1);
supporting functions (3/3)
< sil24_port_stop mv_port_stop ahci_port_stop  >
```

Figure 2: Spdiff Result 1

The second change set adds cm_cleanup_timewait because timewait_info is overwritten. We have two pairs of code in the change set as shown in Figure 3. Spdiff infers a semantic patch as shown in Figure 4.

The third change set upgrades the code with new function devm_kzalloc. We have four pairs of code in the change set as shown in Figure 5. Spdiff infers a semantic patch as shown in Figure 6.

```
@@ -3,7 +3,6 @@
  int wait_time;
  unsigned long flags;
  spin_lock_irqsave(&cm.lock, flags);
-cm_cleanup_timewait(cm_id_priv->timewait_info);
  list_add_tail(&cm_id_priv->timewait_info->list,
&cm.timewait_list);
  spin_unlock_irqrestore(&cm.lock, flags);
  cm_id_priv->id.state = IB_CM_TIMEWAIT;
@@ -19,7 +18,6 @@
  cm_id_priv->id.state = IB_CM_IDLE;
  if (cm_id_priv->timewait_info) {
  spin_lock_irqsave(&cm.lock, flags);
-cm_cleanup_timewait(cm_id_priv->timewait_info);
  spin_unlock_irqrestore(&cm.lock, flags);
  kfree(cm_id_priv->timewait_info);
  cm_id_priv->timewait_info = NULL;
```

Figure 3: Change Set 2

```
[Main] *REAL* semantic patches inferred: 1
[spatch:]
@@
struct cm_id_private *  X8;
unsigned long X5;
expression X5;
X0;
X8;
@@
  head:def[void X0 ( struct cm_id_private *  X8) {};]
  ...
  unsigned long X5;
  ...
  spin_lock_irqsave(&cm.lock,X5);
  ...
- cm_cleanup_timewait(X8->timewait_info);
  ...
  spin_unlock_irqrestore(&cm.lock,X5);
  ...
  X8->timewait_info=NULL;
supporting functions (2/2)
< cm_enter_timewait cm_reset_to_idle  >
```

Figure 4: Spdiff Result 2

The fourth change set adds mthca_array_cleanup before mthca_alloc_cleanup. We have three pairs of code in the change set as shown in Figure 7. Spdiff infers a semantic patch as shown in Figure 8.

```
@@ -4,7 +4,7 @@
  struct fimc_md *fmd;
  int ret;
-fmd = kzalloc(sizeof(struct fimc_md),
GFP_KERNEL);
+ fmd = devm_kzalloc(&pdev->dev, sizeof(struct fimc_md),
GFP_KERNEL);
  if (!fmd)
  return -ENOMEM;
@@ -68,6 +68,5 @@
 err2:
  v4l2_device_unregister(&fmd->v4l2_dev);
 err1:
-kfree(fmd);
  return ret;
 }
@@ -8,7 +8,7 @@
  /* mdev does not exist yet so no mxr_dbg is used */
  dev_info(dev, "probe start\n");
-mdev = kzalloc(sizeof *mdev,
GFP_KERNEL);
+ mdev = devm_kzalloc(&pdev->dev, sizeof *mdev,
GFP_KERNEL);
  if (!mdev) {
  mxr_err(mdev, "not enough memory.\n");
  ret = -ENOMEM;
@@ -50,7 +50,6 @@
  mxr_release_resources(mdev);
 fail_mem:
-kfree(mdev);
 fail:
  dev_info(dev, "probe failed\n");
@@ -13,7 +13,7 @@
  if (!pdata->encoder.module_name)
  dev_info(&pdev->dev, "Running without decoder\n");
-lw = kzalloc(sizeof(*lw), GFP_KERNEL);
+ lw = devm_kzalloc(&pdev->dev, sizeof(*lw),
GFP_KERNEL);
  if (!lw) {
  err = -ENOMEM;
  goto err;
@@ -53,7 +53,6 @@
  platform_set_drvdata(pdev, NULL);
  v4l2_device_unregister(&lw->v4l2_dev);
 err_register:
-kfree(lw);
 err:
  dev_err(&pdev->dev, "Failed to register: %d\n", err);
@@ -18,7 +18,7 @@
  return ret;
  }
-vpbe_dev = kzalloc(sizeof(*vpbe_dev),
GFP_KERNEL);
+ vpbe_dev = devm_kzalloc(&pdev->dev, sizeof(*vpbe_dev),
GFP_KERNEL);
  if (vpbe_dev == NULL) {
  v4l2_err(pdev->dev.driver, "Unable to allocate memory"
   " for vpbe_device\n");
@@ -31,7 +31,6 @@
  if (cfg->outputs->num_modes > 0)
  vpbe_dev->current_timings = vpbe_dev->cfg->
outputs[0].modes[0];
  else {
-kfree(vpbe_dev);
  return -ENODEV;
  }
```

Figure 5: Change Set 3

```
[Main] *REAL* semantic patches inferred: 1
[spatch:]
@@
identifier X8;
X9;
struct platform_device *  X1;
X10 *  X11;
expression X5;
X0;
X1;
@@
head:def[signed int X0 ( struct platform_device *  X1) {;]
...
- X11=kzalloc(X5,GFP_KERNEL);
+ X11=devm_kzalloc(&X1->dev,X5,GFP_KERNEL);
...
  X11->X8=X9;
...
- kfree(X11);
supporting functions (4/4)
< vpbe_probe fimc_md_probe mxr_probe timblogiw_probe  >
```

Figure 6: Spdiff Result 3

```
@@ -1,6 +1,5 @@
 void mthca_cleanup_cq_table(struct mthca_dev *dev)
  {
-mthca_array_cleanup(&dev->cq_table.cq, dev->limits.num_cqs);
  mthca_alloc_cleanup(&dev->cq_table.alloc);
 }
  void mthca_cleanup_qp_table(struct mthca_dev *dev)
@@ -11,14 +10,12 @@
  for (i = 0; i < 2; ++i)
  mthca_CONF_SPECIAL_QP(dev, i, 0, &status);

-mthca_array_cleanup(&dev->qp_table.qp, dev->limits.num_qps);
  mthca_alloc_cleanup(&dev->qp_table.alloc);
 }
  void mthca_cleanup_srq_table(struct mthca_dev *dev)
 {
   if (!(dev->mthca_flags & MTHCA_FLAG_SRQ))
  return;
-mthca_array_cleanup(&dev->srq_table.srq, dev->limits.num_srqs);
  mthca_alloc_cleanup(&dev->srq_table.alloc);
 }
```

Figure 7: Change Set 4

```
[Main] *REAL* semantic patches inferred: 1
[spatch:]
@@
identifier X4;
identifier X1;
struct mthca_dev *  X0;
identifier X6;
X7;
X0;
@@
  head:def[void X7 ( struct mthca_dev *  X0) {};]
  ...
- mthca_array_cleanup(&X0->X1.X4,X0->limits.X6);
  ...
  mthca_alloc_cleanup(&X0->X1.alloc);
supporting functions (3/3)
< mthca_cleanup_cq_table mthca_cleanup_qp_table
mthca_cleanup_srq_table  >
```

Figure 8: Spdiff Result 4