Automating the Porting of Linux to the VirtualLogix Hypervisor using Semantic Patches

François Armand, Jean Berniolles, Julia L. Lawall, Gilles Muller

▶ To cite this version:

François Armand, Jean Berniolles, Julia L. Lawall, Gilles Muller. Automating the Porting of Linux to the VirtualLogix Hypervisor using Semantic Patches. Embedded Real Time Software and Systems (ERTS2008), Jan 2008, Toulouse, France. hal-02270310

HAL Id: hal-02270310

https://hal.science/hal-02270310

Submitted on 24 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automating the Porting of Linux to the VirtualLogix Hypervisor using Semantic Patches

François Armand,[1] Jean Berniolles, Julia L. Lawall,[2] Gilles Muller[3]

[1]VirtualLogix, Paris, France

[2]DIKU, University of Copenhagen, Copenhagen, Denmark

[3]Ecole des Mines de Nantes, Nantes, France

Abstract: Virtualization is a promising technology for running multiple operating systems (OS's) on a single processor. Preparing an OS for use with virtualization, however, involves making some changes to the OS code, which must be repeated for each version, whether a new release or a client customization. Typically such changes are expressed as patches, but patches are often not portable from one version to another, and thus manual adjustments are needed as well. In this paper, we consider the use of the automated transformation system Coccinelle to perform the changes required to port several versions of Linux to the VLX hypervisor. Coccinelle provides a notion of semantic patches, which are more abstract than standard patches, and thus are potentially applicable to a wider range of OS versions. We have applied this approach in the context of Linux versions 2.6.13, 2.6.14, and 2.6.15, for the ARM architecture.

Keywords: Linux, virtualization, VLX, automated program transformation

1. Introduction

Virtualization is becoming a key technology in embedded systems such as mobile phones and set-top boxes. In this context, virtualization enables hosting several operating systems (OSes) on a single uni-core or multi-core processor via a hypervisor that provides each OS with its own execution environment or virtual machine by partitioning and/or virtualizing hardware resources. The main benefit of this technology is a simpler and less expensive hardware architecture, not requiring multiple dedicated processors, that also reduces energy consumption. However, since most processors for embedded systems do not provide any support for hardware assisted virtualization, one must rely on paravirtualization: guest OSes have to be modified to be run efficiently on top of the hypervisor.

The process of adapting an OS for virtualization is quite similar to that of porting an OS to a new target platform. In particular, virtualizing Linux amounts to changing the interface between the hardware and the lowest part of the OS, known as the Hardware Abstraction Layer (HAL). Not only is such a port time-consuming and error-prone, requiring many changes to the OS source code, but it has to be undertaken each time there is a need to upgrade the OS to a more recent version. In the case of Linux, there is currently a new minor release every three months. Furthermore, in the context of a company like VirtualLogix that develops hypervisors, each client that maintains a Linux version with its own proprietary extensions must itself do the virtualization porting task. Therefore, there is a need for documenting and automating the task of porting Linux.

In recent work, in the context of a French ANR project, we have developed the program transformation tool Coccinelle [5] for documenting and automating evolutions induced by the changes of Linux internal APIs. We refer to such changes as *collateral evolutions*. Coccinelle offers a language for creating *semantic patches* that describe modifications to be applied to all the clients of a library, in a generic manner [5]. To this end, a semantic patch is insensitive to variations in spacing and control-flow, and to other code equivalences. Coccinelle has been successfully used to perform evolutions in over 5000 driver files, drawn from Linux 2.5 and 2.6.

In this paper, we report on our experience in using Coccinelle for automating the port of Linux to VLX, the VirtualLogix hypervisor. Because semantic patches have been designed for expressing interface changes, we conjectured that the HAL induced evolutions could be captured as a set of semantic patches. The result would enable a nearly complete automation of the ports of subsequent Linux subversions. With respect to the code written in C, the Linux 2.6.13 HAL modifications for ARM processors affect 16 files that involve 1800 code sites. Documenting these modifications using semantic patches and applying the transformations automatically using the Coccinelle tool was done in 5 weeks (including the time to learn Coccinelle) by the second author as part of his Masters degree. The semantic patches in total are 2000 lines long and contain 79 rules. To validate the approach, we applied the same set of rules to Linux 2.6.14, and were able to get a running version of Linux within a few days.

The rest of the paper is structured as follows. Section 2 gives an overview of Coccinelle and Section 3 presents the Coccinelle transformation language by example. Sec-

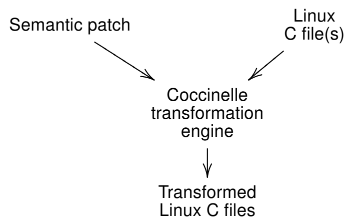

Figure 1: Using Coccinelle

tion 4 describes our experiments in automating the port of Linux to VLX. Section 5 presents related work and Section 6 concludes.

2. Overview of Coccinelle

Coccinelle is an automated program transformation system. Automated program transformation is now starting to see wide use in the form of the refactoring engines provided by integrated development environments such as Eclipse. Coccinelle, however, diverges from such systems in that the set of transformations is not built-in, and instead must be specified by the programmer, using the semantic patch language SmPL. This requires some investment from the programmer, but provides greater flexibility. SmPL is based on the standard patch syntax [3], in which transformations are expressed in terms of code fragments. Initial reactions from Linux developers who have read SmPL code suggest that they find it understandable.[1]

Figure 1 illustrates the usage of Coccinelle. Initially, the programmer develops a semantic patch that expresses the desired transformation. The semantic patch may be programmed directly, but in practice it is often useful to derive it from the differences found in some sample files. Specifically, one can update some typical files by hand, apply `diff` to compute the differences between the old and new versions, and finally generalize the result to create a semantic patch. The semantic patch is then given to the Coccinelle transformation engine along with the files that are to be transformed. The transformation engine applies the semantic patch to the files, and can either modify them in place, create new copies, or produce a record of the changes as a Linux patch, as desired. Coccinelle preserves the coding style of the original files, by preserving the preprocessor code, spacing, and comments, so that the resulting files are amenable to further development and maintenance.

In many of our experiments, we have applied Coccinelle to the entire Linux kernel. Because of the use of indexing, provided by `glimpse` [4], and other related strategies to reduce the amount of code that is considered in detail, this process typically requires only a few minutes. It is also possible to limit the application to a single directory or a set of files if more control is required over where the transformation is performed.

[1] http://www.emn.fr/x-info/coccinelle/#feedback

3. SmPL Samples

In this section, we illustrate SmPL semantic patches through a series of examples. All examples are drawn from collateral evolutions that have been needed in recent versions of Linux.

3.1 Basic structure

A semantic patch consists of a sequence of *rules*, each of which begins with some context information, delimited by a pair of `@@`s, and then describes a transformation. This structure is based on that of a standard patch, which consists of a sequence of *regions*, each of which begins with the affected line numbers, delimited by a pair of `@@`s, and then describes a transformation as a sequence of lines to be added and removed. In the case of a semantic patch, the context information contains a variety of information about how the transformation should be carried out, as described below, and the description of the transformation lists the subterms that should be added and removed, as well as any required relationships between them.

As a first example, we consider the case of a set of interrupt handling flags, which in Linux 2.6.18 received new names. These changes affected over 500 files, including the file `drivers/net/tg3.c`. Figure 2 shows a standard patch for updating `tg3.c` accordingly. This patch was created by first making a copy of the file, then making the changes at the various places where it was needed, and finally using the `diff` tool with appropriate options to capture the differences between the old and new versions. In the resulting patch, each change is indicated by the old and new versions, marked by - and +, respectively, of the complete line in which it occurs. Because each line containing a use of the interrupt handling flags is different, the patch consists of many regions, potentially one for every affected statement. The patch is furthermore applicable to only a single file, the file from which it was created.

Figure 3 shows a semantic patch for making the same set of changes.[2] A semantic patch mentions only the specific terms that are affected, *i.e.*, the old interrupt flags and their new counterparts. Because there is no extra code specific to a given Linux file, this semantic patch can apply to any Linux file. Indeed, we can apply this single semantic patch to the entire Linux source tree to update all of the files affected by the collateral evolution, including `tg3.c`, at once.

A semantic patch is applied to a source file by applying each rule in turn to each of the source file functions and top-level declarations. No inter-procedural analysis is per-

[2] There are actually 11 interrupt flags that change names; for conciseness, we show only the ones that are relevant to the file `tg3.c`. The full semantic patch, which is available at our website, is similar.

```
--- drivers/net/tg3.c
+++ drivers/net/tg3.c
@@ -6702,12 +6702,12 @@
        fn = tg3_msi;
        if (tp->tg3_flags2 & TG3_FLG2_1SHOT_MSI)
            fn = tg3_msi_1shot;
-           flags = SA_SAMPLE_RANDOM;
+           flags = IRQF_SAMPLE_RANDOM;
    } else {
        fn = tg3_interrupt;
        if (tp->tg3_flags & TG3_FLAG_TAGGED_STATUS)
            fn = tg3_interrupt_tagged;
-           flags = SA_SHIRQ | SA_SAMPLE_RANDOM;
+           flags = IRQF_SHARED | IRQF_SAMPLE_RANDOM;
    }
    return (request_irq(tp->pdev->irq, fn, flags, dev->name, dev));
 }
@@ -6726,7 +6726,7 @@
    free_irq(tp->pdev->irq, dev);

    err = request_irq(tp->pdev->irq, tg3_test_isr,
-             SA_SHIRQ | SA_SAMPLE_RANDOM, dev->name, dev);
+             IRQF_SHARED | IRQF_SAMPLE_RANDOM, dev->name, dev);
    if (err)
        return err;
```

Figure 2: Patch updating `tg3.c` to use the new interrupt flags

```
@@ @@
- SA_SHIRQ
+ IRQF_SHARED

@@ @@
- SA_SAMPLE_RANDOM
+ IRQF_SAMPLE_RANDOM
```

Figure 3: Semantic patch for updating any file to use the new interrupt flags

formed. The transformation indicated by a rule is only applied if the rule matches completely. Furthermore, a rule is never applied to the result of its own application, so termination is guaranteed.

3.2 Metavariables and dots

The example of the previous section shows that a semantic patch abstracts away from the specific code surrounding a change site. It is, however, often necessary to abstract away from some of the code contained within the affected terms, as well. For example, in Linux 2.6.23, the function `kmem_cache_create` lost its last argument. To describe the transformation without being specific about the contents of the other arguments of this function, which may be different at each usage site, we introduce metavariables, *E1* through *E5*, to represent them. Metavariables are declared between the `@@`s at the beginning of the semantic patch rule. The declaration of a metavariable describes what kind of term it can match, *e.g.*, `identifier`, `expression`, `type`, *etc.* In the case of the `kmem_cache_create` semantic patch, all metavariables represent `expressions`. The semantic patch is shown in Figure 4.

The semantic patch in Figure 4 removes a complete call to `kmem_cache_create` and then adds it back. Another approach is to simply remove the last argument, as shown

```
@@ expression E1, E2, E3, E4, E5, E6; @@
- kmem_cache_create(E1, E2, E3, E4, E5, E6)
+ kmem_cache_create(E1, E2, E3, E4, E5)
```

Figure 4: Semantic patch for updating calls to `kmem_cache_create`

```
@@ expression E1, E2, E3, E4, E5, E6; @@

  kmem_cache_create(E1, E2, E3, E4, E5
-                                       , E6
                   )
```

Figure 5: Semantic patch for updating calls to `kmem_cache_create`, specifying removal only

```
@@ expression E; @@

  kmem_cache_create(...
-                  , E
                   )
```

Figure 6: Semantic patch for updating calls to `kmem_cache_create`, specifying only the last argument

in Figure 5. It is also possible to specify that the last argument should be dropped without being precise about how many arguments precede it, by using three dots, *i.e.* "`...`", as shown in Figure 6. Dots, however, may not appear in added code.

3.3 Reusing metavariables

The previous examples merely use metavariables as placeholders. A semantic patch can also use metavariables to ensure that various parts of the code matched by a semantic patch have the same structure.

As an example, we consider the reorganization of the management of work queue structures, as was performed in Linux 2.6.20. A work queue is initialized using the macro `INIT_WORK`. Prior to Linux 2.6.20, this macro took three arguments: the address of the work structure, a pointer to the function that should be invoked to perform the desired work, and a pointer to the data that this function should use. Starting in Linux 2.6.20, this macro was redefined such that it no longer took the data argument; now the work function was passed the work structure when the work was to be performed. This change implied that not only did all calls to `INIT_WORK` have to lose their last argument, as for `kmem_cache_create`, but it was also necessary to store the original data somewhere so that the work function could retrieve it. In one common case, the data is the structure that contains the work structure. In this case, the revised work function can apply the macro `container_of` to the work structure to retrieve the desired data. The semantic patch of Figure 7 implements the transformations required in this case.

The first rule in Figure 7 detects the case where the work structure passed as the first argument to `INIT_WORK` is a field of the structure passed in as the data in the

```
@ device_arg @
type struct_type;
struct_type *device;
identifier fn, fld;
@@
  INIT_WORK(&device->fld, fn
-            , device
           );
@@
identifier dataq, device_arg.fn, device_arg.fld;
type device_arg.struct_type;
fresh identifier workq;
@@
  fn (
-      struct_type *dataq
+      struct work_struct *workq
     )
  {
+      struct_type *dataq = container_of(workq,struct_type,fld);
       ...
  }
```

Figure 7: A semantic patch implementing collateral evolutions in the use of a work structure

third argument. This rule defines several metavariables: *struct_type* is an arbitrary type, *device* is an expression of type pointer to *struct_type*, *fn* is an identifier that is the name of the work function, and *fld* is another identifier that is the name of the field holding the work structure. As in the case of the semantic patch for `kmem_cache_create`, the transformation part of the rule specifies that the last argument to `INIT_WORK` should be dropped. The transformation part furthermore specifies that the expression matching *device* must appear in both the first and third arguments. Note that although `INIT_WORK` is a macro, it appears without expansion in both the semantic patch and the generated code. Coccinelle does not expand macros or other preprocessor directives, to maintain the code structure.

The second rule in Figure 7 updates the definition of the work function. This rule handles the case where the type of the original parameter *dataq* of the work function is the same as the type of the data value. The parameter *dataq* is replaced by a parameter with a fresh name of type `work_struct`, and *dataq*, which is likely to be used within the body of the work function, is converted to a local variable, which is initialized using a call to `container_of`. This rule again uses "`...`", here to represent the arbitrary code in the body of the function. In general, "`...`" can be used whenever there is a region of arbitrary code about which nothing further needs to be specified.

The second rule uses a number of metavariables that were defined in the first rule: *fn* for the name of the work function, *fld* for the name of the field containing the work structure, and *struct_type* for the type of the data argument. To indicate that these metavariables should have the values established by the first rule, we give a name to the first rule, `device_arg`, within the rule's initial `@@`s, and then use this name in declaring the *fn*, *fld*, and *struct_type* metavariables in the second rule. In general, the first rule may apply at multiple sites in the file, *i.e.*, if there are multiple calls to `INIT_WORK`. For each such call, there may be a different work function that must be updated. Coccinelle

```
@@ expression E; @@

- if ((E->flags & (1 << TTY_DO_WRITE_WAKEUP))
-     && (E->ldisc.write_wakeup != NULL) ) {
-       ... when = \( printk(...); \| dbg(...); \)
-       E->ldisc.write_wakeup(E); }
+ tty_wakeup(E);
  ...
- wake_up_interruptible(&E->write_wait);
```

Figure 8: A semantic patch introducing the use of `tty_wakeup`

thus instantiates the second rule for each possible tuple of values of *fn*, *fld*, and *struct_type* and then applies each instantiation to the entire source file.

The semantic patch shown in Figure 7 addresses only one of many possible cases, depending on where the data for the work function should be stored. The full semantic patch addresses cases where it is stored in a global variable, where it is already a field of an existing structure, where it must be added as a field of an existing structure, *etc.*

3.4 More about dots

In several semantic patches, we have used "`...`" to represent some arbitrary code. When "`...`" appears at the top-level in a sequence of statements, it means not an arbitrary block of code, but an arbitrary path in the function's control-flow graph. This path is furthermore the shortest one that has the given endpoints.

As an example, consider the semantic patch shown in Figure 8. In the first four lines, this semantic patch matches a rather complicated conditional. This conditional must then be followed along all execution paths by a call to `wake_up_interruptible`. The matching of "`...`" in terms of control-flow paths means that the conditional and the call need not be in a straight-line sequence, but a matching call to `wake_up_interruptible` must always be executed after the conditional,[3] to ensure that the transformation to use `tty_wakeup`, which encompasses both the conditional and the call to `wake_up_interruptible`, preserves the original behavior. Furthermore, the use of the shortest path ensures that when the pattern appears multiple times within a single function, the right conditional is matched up with the right `wake_up_interruptible`.

The semantic patch in Figure 8 also uses "`...`" at the statement level in matching the initial conditional. Here "`...`" is qualified with a `when` clause requiring that the matched code contain only printing or debugging statements. Because all of this code will be deleted, it is important to ensure that it does not have any effect on the behavior of the rest of the function. It is also possible to specify code that should not appear as a subterm within the code matched by "`...`", by using `when` with `!=` rather

[3] Coccinelle relaxes this requirement for control-flow paths identified as error paths.

than =.

Finally, the semantic patch in Figure 8 explicitly compares `E->ldisc.write_wakeup` to `NULL`, while Linux code often checks for the non-nullity of a pointer by just testing the value of the pointer itself. In general, for an arbitrary pointer-typed expression *X*, all of *X* `== NULL`, `NULL ==` *X*, and *X* are equivalent. Coccinelle abstracts away from these minor syntactic differences by a collection of rules known as *isomorphisms*. Isomorphisms are specified using a syntax similar to that of semantic patches, and new isomorphisms can be defined by the user.

3.5 Assessment

SmPL provides a simple, code-based notation for expressing the kinds of collateral evolutions that are often needed in Linux code. The language is also in the spirit of the standard patch syntax, which is already familiar to Linux developers. We have written over 60 semantic patches implementing collateral evolutions that have been required in recent versions of Linux [5]. These collateral evolutions affect over 5000 files from various Linux versions, and our semantic patches have been sufficient to update over 93% of these files. In addition, we have identified over 150 files in which the human programmer made a mistake in performing the collateral evolution, but the semantic patch performs the transformation correctly. Finally, we have recently submitted several patches to Linux based on our work that complete previous collateral evolutions.

4. Applying Coccinelle to VLX

We next consider how Coccinelle can be used to perform the changes required to port an existing version of Linux to VLX. The actual porting work was done by a Masters students who had no previous experience with Coccinelle, Linux kernel code, or VLX. This task is, furthermore, quite different from that of the collateral evolutions considered previously. Collateral evolutions entail an API-level change at many places in a single version of the Linux kernel. In the case of porting an OS, our goal is to make many kinds of changes, both high and low level, across multiple versions of the Linux kernel. We find that while the approach is adequate for applying the port to closely related Linux versions, it is not sufficient when the versions start to diverge significantly in functionality.

In the rest of this section, we first briefly describe VLX, then enumerate the kinds of changes that are required to port an existing version of Linux to VLX, and finally describe our results in applying semantic patches derived from Linux 2.6.13 to later versions of Linux.

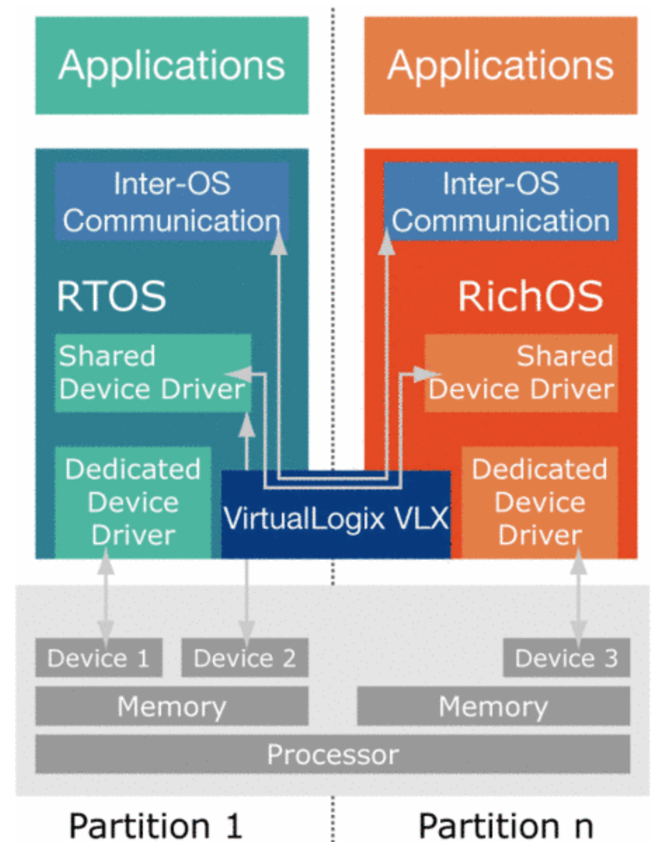

Figure 9: An example use of VLX

4.1 Overview of VLX

VLX is a real-time hypervisor that enables multiple guest OS's to run simultaneously on the same single-core or multi-core processor. A typical use of VLX is illustrated in Figure 9. With VLX, the guest OS's are independent from each other, but can cooperate via efficient communication mechanisms. VLX runs on various processors, including Intel x86, PPC, DSP's and ARM. The version used in this project was based on the ARM processor.

VLX isolates the guest OS's from the underlying hardware. It uses partitioning of resources between the guest OS's and virtualization of resources that cannot be partitioned. Typically, physical memory is partitioned between the guest OS's while the CPU, FPU, MMU and other parts of the system such as the real-time clock and interrupt controller are virtualized.

Partitioned resources such as memory, which will only be used by a single given guest OS, are exclusively owned by that guest OS. Thus, each OS may use its own native mechanisms and policies, such as memory management, without interfering with other guest OS's. Similarly, I/O devices used by a single guest OS are assigned to that guest OS, and thus native device drivers can be re-used without any modification.

Resources that are common to more than one guest OS

such as the CPU and real-time clock are virtualized so that they can be shared between those various guest OS's which need to access the resource.

In order to ensure efficiency, VLX employs paravirtualization techniques, meaning that some adaptation of the guest OS kernel is required. These changes are comparable in both effort and scope to porting that OS to a hardware very similar to the underlying one.

VLX virtualizes the CPU, FPU and MMU (if any). The CPU is shared by means of a scheduler which assigns the processor to the selected guest OS and guarantees that a real-time guest OS will get a higher priority. When a guest OS has been granted CPU access, it uses its own native scheduling policies for its applications.

If present, the MMU is virtualized so that each guest OS may use it for its own purposes. Use of the MMU by one guest OS is independent from the use of the MMU by another guest OS.

VLX provides each guest OS with synchronization (cross-interrupt) mechanisms, shared access to devices such as disk controllers, network interfaces, serial lines and inter-OS communication mechanisms through virtual devices.

Devices such as an Ethernet controller or a serial line may need to be accessed by more than one guest OS. For such I/O devices, “back-end” device drivers manage the physical hardware devices, virtualize it, and export a virtual view of that device to other guest OS. This approach provides these guest OS's with access to features of each device without actual access to the device.

Communication between the different guest OS's is provided by virtual communication devices. Different types of such devices can be configured depending upon the needs of the communicating applications. For example, a system might use a virtual Ethernet to implement a local private network that is located wholly internally to the machine, and/or it might use a virtual UART device to pass AT modem commands from one guest OS to the other.

4.2 Creating semantic patches for VLX

Porting a version of Linux to VLX requires making changes to the assembly code, the C code, and the various configuration and make files. As we are considering only the ARM architecture, these changes primarily affect files in the `arch/arm` directory. Because Coccinelle only works on C code, the other changes have to be made by hand. We found that between Linux 2.6.13 and 2.6.14 there was no change in the code in the context of the assembly code that needed to be changed, and thus it was possible to simply copy the new assembly code from the 2.6.13 version and to the 2.6.14 version. Updating the configuration and make files was also reasonably straightforward.

In the C code, the main kinds of changes that are required to port a version of Linux to VLX are to add the VLX include files, to add new functions, to insert calls to these new functions, and to augment structure declarations. In particular, the changes are mostly non-intrusive; the main changes required within existing function bodies are to replace a call to a Linux function by a call to its VLX counterpart.

All of the files affected by VLX must include the VLX include files. For this, a semantic patch such as the following was used:

```
@@ @@
  #include <asm/memory.h>
+ #ifdef CONFIG_NKERNEL
+ #include <asm/nk/f_nk.h>
+ #endif
```

The new code is added within ifdefs to maintain the ability to generate both the original and ported implementations. This semantic patch relies on the presence of an include of `asm/memory.h` in the source file to determine where to perform the modification. In general, however, it may not be the case that there is some existing include file that is common to all of the files that should be modified. SmPL thus also permits only a portion of the name of an include file to be specified, for greater genericity. The following semantic patch specifies that the new include file(s) should be added after the last `asm` header file include:

```
@@ @@
  #include <asm/...>
+ #ifdef CONFIG_NKERNEL
+ #include <asm/nk/f_nk.h>
+ #endif
```

At the function level, 42 function definitions needed to be inserted and 11 deleted. These changes are again made within ifdefs. Adding a function is expressed similarly to adding an include, in terms of existing code that the function should be placed adjacent to:

```
@@ @@
+ #ifdef CONFIG_NKERNEL
+ static inline void nkidle(void) {
+   // the definition of nkidle
+ }
+ #endif
  cpu_idle(...) { ... }
```

Removing a function similarly involves adding ifndef around it, as follows:

```
@@ @@
+ #ifndef CONFIG_NKERNEL
  cpu_idle(...) { ... }
+ #endif
```

Note that in both cases, we have specified code that should be added before the function definition by indicating that it should appear directly before the function name, without mentioning the return type or other attributes that may be associated with a function declaration. Coccinelle ensures that the added code is placed before the complete function definition and its attributes, thus making the semantic patch insensitive to changes in the set of attributes that are provided.

Thirty-two changes were required within existing functions. These were primarily to replace function calls by calls to their just-added VLX counterparts. A typical semantic patch performing such a change is as follows:

```
@@
expression E1, E2;
@@

  doSomething(type parameter) {
    <...
+ #ifdef NKERNEL
+   nk_action(E1, E2);
+ #elseif
    native_action(E1, E2);
+ #endif
    ...>
}
```

This semantic patch converts calls to `native_action` to calls to `nk_action`. Metavariables are used to copy the arguments from the original call to the new one. In general, the new arguments can also be computed from the old ones in some way, *e.g.*, reversing them, dropping some, or constructing a new value based on other information matched in the semantic patch. This semantic patch only converts calls occurring within the function `doSomething`. The use of `<...` and `...>`, known as a *nest*, indicates that this transformation should be performed wherever a call to `native_action` occurs within the function body. The dots, "`...`", we have used previously would require that every control-flow path from the beginning to the end of the function contain exactly one call to `native_action`, which in the case of renaming a function is often too constraining.

In nine cases, it was necessary to add or remove a field from a structure type declaration. In this case, the semantic patch has the form of the structure declaration, some lines are specified to be removed using the `-` annotation, and others are specified to be added before or after existing fields.

Finally, in nine cases some other top-level declarations needed to be added. These were done analogously to the insertion of a new include or a top-level function definition.

4.3 Applying the semantic patches to Linux 2.6.14 and Linux 2.6.15

The semantic patches were created from the changes required to port Linux 2.6.13 to VLX. Our goal, however, is not to simply create a patchset that is applicable to a single version of Linux, but instead to create one that can be reused over multiple versions of Linux, including both subsequent versions in the Linux source tree and versions that have been customized by a client. To determine to what extent we have reached our goal, we have applied our semantic patches to Linux 2.6.14 and 2.6.15.

In the case of Linux 2.6.14, we used semantic patches derived from the port of Linux 2.6.13 to two versions of VLX: MH 3.0.1 and MH 3.1. For MH 3.0.1, only two days were necessary to obtain a functioning VLX version of Linux 2.6.14. The main difficulty encountered was to manually fix some erroneous code that resulted from weaknesses in Coccinelle's treatment of preprocessor code. These problems in Coccinelle have subsequently been fixed. For MH 3.1, four days were required. The transformation was more difficult in this case, because it touched the implementation of third-party drivers, which changed in a more drastic way between Linux 2.6.13 and Linux 2.6.14 than ARM code in the Linux kernel source tree. These changes implied that the context code in the semantic patch before or after which the VLX code was to be added did not always have the same form in Linux 2.6.13 and Linux 2.6.14. In many cases, it was possible to make this context code more generic, allowing it to match both the Linux 2.6.13 code and the Linux 2.6.14 code. As an example, when adding one function before another existing function, the existing function can be described with an explicit parameter list or with `(...)` as the parameter list. The latter encoding is less sensitive to changes in the source code.

The problems encountered in porting Linux 2.6.14 to MH 3.1 persisted and multiplied in trying to port Linux 2.6.15 to MH 3.1. In particular, the memory initialization strategy appears to have changed drastically in Linux 2.6.15, thus making the patches derived from Linux 2.6.13 inadequate to complete a port.

4.4 Assessment

The success of our experiment with VLX was only partial; we showed that semantic patch could be applied to a version of Linux other than the one for which they were designed, but we found that the semantic patches were fragile with respect to certain kinds of changes in Linux code.

Indeed, as initially noted, the task of porting a Linux version is somewhat different from performing the collateral evolutions for which Coccinelle was developed. The semantic patches for collateral evolutions described in Section 3 perform localized changes to code directly dependent on generic API functions, and rely heavily on metavariables to provide genericity and to express relationships between disjoint code fragments. The semantic patches for VLX, on the other hand, are dominated by complete functions that are added adjacent to various specific functions in the existing file. There is little use of metavariables, and most semantic patches are applicable at only a single position. The semantic patches are thus not much more abstract than standard patches, and are quite sensitive to changes in the source file across multiple versions. Furthermore, while API changes tend to have an impact that is quite similar across many files, drastic changes can occur when moving from one version of Linux to another, implying that new functionalities have to be introduced in the VLX code. This lead ultimately to the inability to apply the Linux 2.6.13 semantic patches in the Linux 2.6.15 case.

Nevertheless, our experiment with Linux 2.6.14 shows that the use of semantic patches can provide some flexibility, and suggests that the use of semantic patches could be beneficial for porting closely related Linux versions, such as one customized by a client.

5. Related Work

The program transformation system C4 [1] is also motivated by the difficulty of keeping multiple versions of systems code up to date. C4 is inspired by aspect-oriented programming [2], and thus focuses on inserting code before, after, and around standard points of modularity, typically function calls. This degree of expressiveness is sufficient for specifying the transformations that we have carried out in porting Linux to VLX. However, one could consider whether the ability of SmPL to express transformations on any kind of program construct could allow a more fine-grained specification of the transformations required to use VLX, at the statement level rather than the function level. This could reduce the number of lines of code that are affected by the transformation and that thus have to be updated by the client manually, when they affect client customizations. We have not yet explored this issue, but hope to do so in future work. C4 is not currently publicly available.

A number of other program transformation systems have been developed. Among these, JunGL [6] targets the problem of specifying refactorings. The language relies very heavily on library functions to access and manipulate code fragments. In contrast, a SmPL semantic patch is very close to C code. Another well-developed program transformation system is Stratego [7]. Transformations are expressed in terms of concrete syntax code patterns and the Stratego language. This mix of languages can make it difficult to discern the structure of the affected code. Furthermore, Stratego has no notion of the semantics of the code that is being processed, and thus is restricted to the traversal of abstract syntax trees. Coccinelle, on the other hand, follows control-flow paths, which encode the semantics of control structures, such as loops and conditionals.

6. Conclusion

Many specialized variants of Linux are becoming available, for virtualization and other purposes. Because Linux is an open source system, clients can adapt these versions to their needs. But there remains the difficulty of keeping such a hybrid version up to date with the many tracks of Linux development, both in the Linux source tree and in the specialized variants. In this paper, we have shown that semantic patches can help with this problem, by providing a specification of code changes that abstracts away from inessential details, and thus is applicable to multiple versions of the system. But we have also seen limitations of the approach, when the changes in the system to be patched are too drastic. We will consider how to address this issue in future work.

Acknowledgements

This work has been supported in part by the Agence Nationale de la Recherche (France) and the Danish Research Council for Technology and Production Sciences.

Availability

Coccinelle and associated documentation are available at `http://www.emn.fr/x-info/coccinelle/`

References

[1] M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Patch (1) considered harmful. In *10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, June 2005.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001.

[3] 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`.

[4] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *USENIX Winter 1994 Technical Conference*, pages 23–32, San Francisco, CA, Jan. 1994.

[5] 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. To appear.

[6] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *International Conference on Software Engineering (ICSE)*, pages 172–181, Shanghai, China, May 2006.

[7] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, 2004.