Coccinelle:
10 Years of Automated Evolution in the Linux Kernel

Julia Lawall (Inria-Whisper team, Julia.Lawall@inria.fr)
March 2, 2020
Our focus: The Linux kernel

- Open source OS kernel, developed by Linus Torvalds
- First released in 1991
- Version 1.0.0 released in 1994
- Today used in the top 500 supercomputers, billions of smartphones (Android), battleships, stock exchanges, …
Some history

First release in 1991.

- v1.0 in 1994: 121 KLOC, v2.0 in 1996: 500 KLOC

Recent evolution:
Key challenge

As software grows, how to ensure its continued maintenance?

- Updating interfaces is easy.
  Make functions and data structures:
  - More efficient
  - Easier to use correctly
  - Better adapted to their usage context

- Updating the uses of interfaces gets harder as the software grows.
  - More time consuming
  - More error prone
  - Need to communicate new coding strategies to all developers

Developers may hesitate to make needed changes.
As software grows, how to ensure its continued maintenance?

- Updating interfaces is easy.
  Make functions and data structures:
  - More efficient
  - Easier to use correctly
  - Better adapted to their usage context

- Updating the uses of interfaces gets harder as the software grows.
  - More time consuming
  - More error prone
  - Need to communicate new coding strategies to all developers
As software grows, how to ensure its continued maintenance?

- Updating interfaces is easy.
  Make functions and data structures:
  - More efficient
  - Easier to use correctly
  - Better adapted to their usage context

- Updating the uses of interfaces gets harder as the software grows.
  - More time consuming
  - More error prone
  - Need to communicate new coding strategies to all developers

Developers may hesitate to make needed changes.
Example change: init_timer → setup_timer

Initializing a timer requires:

- The callback function to run when the timer expires
- The data that should be passed to that callback function
Example change: `init_timer` → `setup_timer`

Initializing a timer requires:

- The callback function to run when the timer expires
- The data that should be passed to that callback function

Original initialization strategy (present in Linux v1.2.0):

```c
init_timer(&ns_timer);
ns_timer.data = 0UL;
ns_timer.function = ns_poll;
```
Example change: init_timer → setup_timer

Replacement initialization strategy (introduced in Linux v2.6.15, Jan. 2006):

```c
setup_timer(&ns_timer, ns_poll, 0UL);
```

Advantages:

- More concise
- More uniform
- More secure
Example change: \texttt{init\_timer} $\rightarrow$ \texttt{setup\_timer}
Device node structures are reference counted:

- `of_node_get` to access the structure.
- `of_node_put` to let go of the structure.

Iterators, e.g., `for_each_child_of_node`, put one value and get another.

- Explicit put needed on `break`, `return`, `goto` out of the loop.
- Often forgotten.
Example bug: missing of_node_puts
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - Grep insufficient to find the problem.
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - Grep insufficient to find the problem.

- Changes may be widely scattered across the code base.
  - Tedious and time-consuming to find all occurrences.
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - Grep insufficient to find the problem.

- Changes may be widely scattered across the code base.
  - Tedious and time-consuming to find all occurrences.

- Changes may come in many variants.
  - Hard to anticipate; some variants may be overlooked.
Assessment

- **Changes may involve scattered code fragments and data and control flow relationships between them.**
  - Grep insufficient to find the problem.

- **Changes may be widely scattered across the code base.**
  - Tedious and time-consuming to find all occurrences.

- **Changes may come in many variants.**
  - Hard to anticipate; some variants may be overlooked.

- **Developers are unaware of changes that affect their code.**
  - New code can be introduced using the old coding strategy.
Coccinelle to the rescue!
What is Coccinelle?

- Pattern-based tool for matching and transforming C code
- Allows code changes to be expressed using patch-like code patterns (semantic patches).

**Goal:** Automate large-scale changes in a way that fits with the habits of the Linux kernel developer.
--- a/drivers/atm/nicstar.c
+++ b/drivers/atm/nicstar.c
@@ -287,4 +287,2 @@
-    init_timer(&ns_timer);
+    setup_timer(&ns_timer, ns_poll, 0UL);
        ns_timer.expires = jiffies + NS_POLL_PERIOD;
-    ns_timer.data = 0UL;
-    ns_timer.function = ns_poll;
Semantic patches

- Like patches, but independent of irrelevant details (line numbers, spacing, variable names, etc.)
- Derived from code, with abstraction.
Example: Creating an \texttt{init\_timer} $\rightarrow$ \texttt{setup\_timer} semantic patch

A patch: derived from drivers/atm/nicstar.c

- \texttt{init\_timer(&ns\_timer);}  
+ \texttt{setup\_timer(&ns\_timer, ns\_poll, 0UL);}  
  \texttt{ns\_timer.expires = jiffies + NS\_POLL\_PERIOD;}  
- \texttt{ns\_timer.data = 0UL;}  
- \texttt{ns\_timer.function = ns\_poll;}
Example: Creating an \texttt{init\_timer} $\rightarrow$ \texttt{setup\_timer} semantic patch

Remove irrelevant code:

- \texttt{init\_timer}(&\texttt{ns\_timer});
+ \texttt{setup\_timer}(&\texttt{ns\_timer}, \texttt{ns\_poll}, 0UL);
  ...
- \texttt{ns\_timer.data} = 0UL;
- \texttt{ns\_timer.function} = \texttt{ns\_poll};
Example: Creating an `init_timer` → `setup_timer` semantic patch

Abstract over subterms:

```plaintext
@@
expression timer, fn_arg, data_arg;
@@
-     init_timer(&timer);
+     setup_timer(&timer, fn_arg, data_arg);
     ...
-     timer.data = data_arg;
-     timer.function = fn_arg;
```
Example: Creating an `init_timer → setup_timer` semantic patch

Generalize a little more:

```c++
expression timer, fn_arg, data_arg;

- init_timer(&timer);
+ setup_timer(&timer, fn_arg, data_arg);
...
- timer.data = data_arg;
...
- timer.function = fn_arg;
```
**Dataset:** 598 Linux kernel `init_timer` files from different versions.

- 828 calls.
- Our semantic patch updates 308 of them.
Dataset: 598 Linux kernel init_timer files from different versions.

- 828 calls.
- Our semantic patch updates 308 of them.

Untreated example: drivers/tty/n_gsm.c:

```c
init_timer(&dlci->t1);
dlci->t1.function = gsm_dlci_t1;
dlci->t1.data = (unsigned long)dlci;
```
Example: Creating an `init_timer` → `setup_timer` semantic patch

Extended semantic patch:

```plaintext
@@ expression timer, fn_arg, data_arg; @@
-    init_timer(&timer);
+    setup_timer(&timer, fn_arg, data_arg);
    ...
-    timer.data = data_arg;
    ...
-    timer.function = fn_arg;
```

Covers 656/828 calls.
Example: Creating an `init_timer → setup_timer` semantic patch

Extended semantic patch:

```plaintext
@@ expression timer, fn_arg, data_arg; @@
-    init_timer(&timer);
+    setup_timer(&timer, fn_arg, data_arg);
    ...
-    timer.data = data_arg;
    ...
-    timer.function = fn_arg;

@@ expression timer, fn_arg, data_arg; @@
-    init_timer(&timer);
+    setup_timer(&timer, fn_arg, data_arg);
    ...
-    timer.function = fn_arg;
    ...
-    timer.data = data_arg;
```

Covers 656/828 calls.
Example: Creating an `init_timer → setup_timer` semantic patch

Remaining issues

- Some code initializes the function and data before calling `init_timer`.
- Some timers have no data initialization, default to 0.
- Coccinelle sometimes times out.

Complete semantic patch

- 6 rules, 68 lines of code.
- Covers 808/828 calls.
- TODO: Some timers have no local function or data initialization.
```
@@
expression root,e;
local id expression child;
iterator name for_each_child_of_node;
@@

for_each_child_of_node(root, child) {
    ...
    when ! = of_node_put(child)
        when ! = e = child
        + of_node_put(child);
    ? break;
    ...
} ...
when ! = child

Used in the big v5.4 cleanup.
```
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - ... connects related fragments over control-flow paths.
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - ... connects related fragments over control-flow paths.

- Changes may be widely scattered across the code base.
  - Coccinelle finds an updates all relevant code automatically.
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - ... connects related fragments over control-flow paths.

- Changes may be widely scattered across the code base.
  - Coccinelle finds an updates all relevant code automatically.

- Changes may come in many variants.
  - Semantic patches are easily adapted to new variants.
Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
  - ... connects related fragments over control-flow paths.

- Changes may be widely scattered across the code base.
  - Coccinelle finds an updates all relevant code automatically.

- Changes may come in many variants.
  - Semantic patches are easily adapted to new variants.

- Developers are unaware of changes that affect their code.
  - Semantic patches in commit logs document changes.
  - Semantic patches can be collected in a library and checked during continuous integration.
Impact: Patches in the Linux kernel

Over 7700 Linux kernel commits up to Linux v5.5 (Jan 2020).
Impact: Cleanup vs. bug fix changes among maintainer patches using Coccinelle
Impact: Maintainer use examples

TTY. Remove an unused function argument.

- 11 affected files.

DRM. Eliminate a redundant field in a data structure.

- 54 affected files.

Interrupts. Prepare to remove the irq argument from interrupt handlers, and then remove that argument.

- 188 affected files.
Impact: 0-day reports mentioning Coccinelle per year

![Graph showing the number of 0-day reports mentioning Coccinelle per year from 2013 to 2017. The graph is divided into two parts: one showing the number with patches and the other showing the number with message only. Each year is represented as a bar, with different colors indicating the categories such as api, free, iterators, locks, null, tests, and misc.](image-url)
Conclusion

- Coccinelle: brings **automatic matching and transformation** to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

http://coccinelle.lip6.fr/
https://github.com/coccinelle/coccinelle
https://github.com/kanghj/coccinelle/tree/java
Conclusion

- **Coccinelle**: brings **automatic matching and transformation** to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

- **Success**: Almost 8000 commits in the Linux kernel based on Coccinelle.
Conclusion

- **Coccinelle**: brings **automatic matching and transformation** to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

- **Success**: Almost 8000 commits in the Linux kernel based on Coccinelle.

- **Future work**: Automatic generation of semantic patches from examples.
Conclusion

- **Coccinelle**: brings *automatic matching and transformation* to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

- **Success**: Almost 8000 commits in the Linux kernel based on Coccinelle.

- **Future work**: Automatic generation of semantic patches from examples.

- **Beyond C**: Some support for C++, a variant for Java (Coccinelle4J)
Coccinelle: brings **automatic matching and transformation** to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

**Success:** Almost 8000 commits in the Linux kernel based on Coccinelle.

**Future work:** Automatic generation of semantic patches from examples.

**Beyond C:** Some support for C++, a variant for Java (Coccinelle4J)

Probably, everyone in this room uses some Coccinelle modified code!
Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

Success: Almost 8000 commits in the Linux kernel based on Coccinelle.

Future work: Automatic generation of semantic patches from examples.

Beyond C: Some support for C++, a variant for Java (Coccinelle4J)

Probably, everyone in this room uses some Coccinelle modified code!

http://coccinelle.lip6.fr/
https://github.com/coccinelle/coccinelle
https://github.com/kanghj/coccinelle/tree/java