Coccinelle: A case study in simplifying resource management in the Linux kernel

Julia Lawall, Inria

June 6, 2024

Coccinelle

Program matching and transformation for C code

- Changes expressed as abstracted **patches** (semantic patches).
- Fits with the experience of the Linux kernel developers.

Example:

```
@@
local idexpression ret;
constant c;
@@

if (...)
- {
-   ret = c;
-   return ret;
+   return c;
- }
```

Coccinelle

Program matching and transformation for C code

- Changes expressed as abstracted **patches** (semantic patches).
- Fits with the experience of the Linux kernel developers.

Sample output:

```
@@ -320,10 +320,7 @@ static int setup_umsch_mm_test(struct am
        test->vm_cntx_cntl = hub->vm_cntx_cntl;

        test->vm = kzalloc(sizeof(*test->vm), GFP_KERNEL);
-       if (!test->vm) {
-               r = -ENOMEM;
-               return r;
-       }
+       if (!test->vm)
+               return -ENOMEM;

        r = amdgpu_vm_init(adev, test->vm, -1);
        if (r)
```

Coccinelle

Overall goals:

- **Large-scale** modifications using **small** semantic patches.
- Semantic patches that are readable,
  to facilitate **communication** and **documentation**.

Coccinelle

Overall goals:

- **Large-scale** modifications using **small** semantic patches.
- Semantic patches that are readable,
  to facilitate **communication** and **documentation**.

Supporting the developer, with:

- Software-specific changes.
- Refinement of existing semantic patches.
- Adaptation of existing semantic patches.
- Incorporating interactivity.

Software-specific changes

Resource management in the Linux kernel

Traditionally, in C, the programmer manages the lifetime of resources:

- `kmalloc() ... kfree()`
- `mutex_lock() ... mutex_unlock()`

Resource management in the Linux kernel

Traditionally, in C, the programmer manages the lifetime of resources:

- `kmalloc() ... kfree()`
- `mutex_lock() ... mutex_unlock()`
- `of_node_get() ... of_node_put()`

Resource management in the Linux kernel

Traditionally, in C, the programmer manages the lifetime of resources:

- `kmalloc() ... kfree()`
- `mutex_lock() ... mutex_unlock()`
- `of_node_get() ... of_node_put()`

Reference count releases, such as `of_node_put()`, are often overlooked.

- A further complexity: `of_node_get()` is often hidden in a wrapper function.

Example usage of of_node_put

```
static int hdmi_probe_of(struct platform_device *pdev)
{
        struct device_node *node = pdev->dev.of_node;
        struct device_node *ep;
        int r;

        ep = omapdss_of_get_first_endpoint(node);
        if (!ep)
                return 0;
        r = hdmi_parse_lanes_of(pdev, ep, &hdmi.phy);
        if (r)
                goto err;
        of_node_put(ep);
        return 0;
err:
        of_node_put(ep);
        return r;
}
```

Example usage of of_node_put

```
static int hdmi_probe_of(struct platform_device *pdev)
{
        struct device_node *node = pdev->dev.of_node;
        struct device_node *ep;
        int r;

        ep = omapdss_of_get_first_endpoint(node);
        if (!ep)
                return 0;
        r = hdmi_parse_lanes_of(pdev, ep, &hdmi.phy);
        if (r)
                goto err;
        of_node_put(ep);
        return 0;
err:
        of_node_put(ep);
        return r;
}
```

Hiding resource management

Device-node iterators:

```
static void sysc_check_children(struct sysc *ddata)
{
        struct device_node *child;

        for_each_child_of_node(ddata->dev->of_node, child)
                sysc_check_one_child(ddata, child);
}
```

Partly hiding resource management

Device-node iterators:

```
static int find_node(struct device_node *tree, struct device_node *np)
{
        struct device_node *child;

        if (tree == np)
                return 1;
        for_each_child_of_node(tree, child) {
                if (find_node(child, np)) {
                        of_node_put(child);
                        return 1;
                }
        }
        return 0;
}
```

Partly hiding resource management

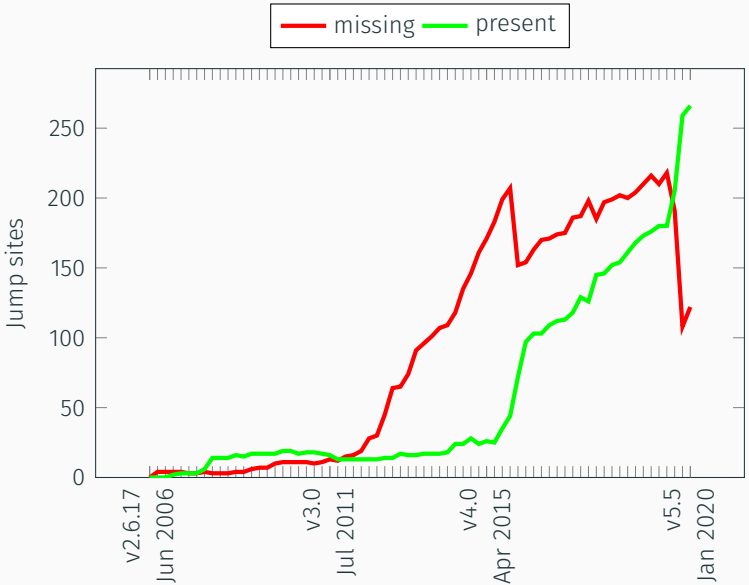
Device-node iterators:

```
static int find_node(struct device_node *tree, struct device_node *np)
{
        struct device_node *child;

        if (tree == np)
                return 1;
        for_each_child_of_node(tree, child) {
                if (find_node(child, np)) {
                        of_node_put(child);
                        return 1;
                }
        }
        return 0;
}
```

Over 400 `for_each_child_of_node` uses in Linux v6.9.

Missing of_node_puts



Scope-based resource management

`__cleanup__` annotation provides a handler to achieve **scope-based** cleanup.

- Supported by the Linux kernel since 2023.
- ```
  struct device_node *node __free(device_node) =
        rsnd_ssi_of_node(priv);
  ```
- For loops, hidden in `for_each_child_of_node_scoped`.
  ```

A semantic patch introducing `for_each_child_of_node_scoped`

```
@@
identifier np,l;
iterator name for_each_child_of_node, for_each_child_of_node_scoped;
expression node;
@@

- struct device_node *np;
  ... when != np
- for_each_child_of_node
+ for_each_child_of_node_scoped
      (node, np) {
    <... when != goto l; ... np
-   of_node_put(np);
    ...>
  }
  ... when != np
```

Sample result

```
struct dma_chan *rsnd_dma_request_channel(struct device_node *of_node, char *name,
                                          struct rsnd_mod *mod, char *x)
{
        struct rsnd_priv *priv = rsnd_mod_to_priv(mod);
        struct device *dev = rsnd_priv_to_dev(priv);
        struct dma_chan *chan = NULL;
-       struct device_node *np;
        int i = 0;

-       for_each_child_of_node(of_node, np) {
+       for_each_child_of_node_scoped(of_node, np) {
                i = rsnd_node_fixed_index(dev, np, name, i);
                if (i < 0) {
                        chan = NULL;
-                       of_node_put(np);
                        break;
                }
                if (i == rsnd_mod_id_raw(mod) && (!chan))
                        chan = of_dma_request_slave_channel(np, x);
                i++;
        }
        of_node_put(of_node);
        return chan;
}
```

Sample result

```
struct dma_chan *rsnd_dma_request_channel(struct device_node *of_node, char *name,
                                          struct rsnd_mod *mod, char *x)
{
        struct rsnd_priv *priv = rsnd_mod_to_priv(mod);
        struct device *dev = rsnd_priv_to_dev(priv);
        struct dma_chan *chan = NULL;
        struct device_node *np;
        int i = 0;

-       for_each_child_of_node(of_node, np) {
+       for_each_child_of_node_scoped(of_node, np) {
                i = rsnd_node_fixed_index(dev, np, name, i);
                if (i < 0) {
                        chan = NULL;
-                       of_node_put(np);
                        break;
                }
                if (i == rsnd_mod_id_raw(mod) && (!chan))
                        chan = of_dma_request_slave_channel(np, x);
                i++;
        }
        of_node_put(of_node);
        return chan;
}
```

Updates 236 loops.

Semantic-patch refinement

Another sample result

```
for_each_child_of_node(node, np) {
        struct rsnd_mod *mod;

        i = rsnd_node_fixed_index(dev, np, SSIU_NAME, i);
        if (i < 0) {
-               of_node_put(np);
                break;
        }
        mod = rsnd_ssiu_mod_get(priv, i);

        if (np == playback)
                rsnd_dai_connect(mod, io_p, mod->type);
        if (np == capture)
                rsnd_dai_connect(mod, io_c, mod->type);
        i++;
}
```

Creates an undesired block with only one element.

Refining the semantic patch

```
@@
identifier np;
iterator name for_each_child_of_node, for_each_child_of_node_scoped;
expression node;
statement S;
@@

- struct device_node *np;
  ... when != np
- for_each_child_of_node
+ for_each_child_of_node_scoped
      (node, np) {
    <... when != goto l; ... np
(
-   {
      of_node_put(np);
      S
-   }
|
-   of_node_put(np);
)
    ...>
  }
  ... when != np
```

Refining the semantic patch

```
@@
identifier np;
iterator name for_each_child_of_node, for_each_child_of_node_scoped;
expression node;
statement S;
@@

- struct device_node *np;
  ... when != np
- for_each_child_of_node
+ for_each_child_of_node_scoped
      (node, np) {
    <... when != goto l; ... np
(
-   {
      of_node_put(np);
      S
-   }
|
-   of_node_put(np);
)
    ...>
  }
  ... when != np
```

Updates 234 loops.

Some other issues

Gotos:

- Pull destination code up into the loop.
- Transform as previously.

Return values:

- Pulling up destination code may produce useless naming of return values.
- Reuse our initial example:

```
@@
local idexpression ret;
constant c;
@@

if (...)
- {
-   ret = c;
-   return ret;
+   return c;
- }
```

Semantic-patch adaptation

Device-node availability

A recurring pattern in the use of `for_each_child_of_node`:

```
for_each_child_of_node(node_prop, node_child) {
        if (!of_device_is_available(node_child))
                continue;

        if (of_property_read_u32(node_child, "reg", &cs)) {
                dev_err(&pdev->dev, "Couldn't get memory chip select\n");
                of_node_put(node_child);
                return -ENXIO;
        } else if (cs >= CDNS_XSPI_MAX_BANKS) {
                dev_err(&pdev->dev, "reg (cs) parameter value too large\n");
                of_node_put(node_child);
                return -ENXIO;
        }
}
```

Device-node availability

A recurring pattern in the use of `for_each_child_of_node`:

```
for_each_child_of_node(node_prop, node_child) {
        if (!of_device_is_available(node_child))
                continue;

        if (of_property_read_u32(node_child, "reg", &cs)) {
                dev_err(&pdev->dev, "Couldn't get memory chip select\n");
                of_node_put(node_child);
                return -ENXIO;
        } else if (cs >= CDNS_XSPI_MAX_BANKS) {
                dev_err(&pdev->dev, "reg (cs) parameter value too large\n");
                of_node_put(node_child);
                return -ENXIO;
        }
}
```

Use `for_each_available_child_of_node_scoped` instead.

Semantic patch for `for_each_available_child_of_node_scoped`

```
@@
identifier np;
iterator name for_each_child_of_node, for_each_available_child_of_node_scoped;
expression node; statement S;
@@

- struct device_node *np;
  ... when != np
- for_each_child_of_node
+ for_each_available_child_of_node_scoped
       (node, np) {
-   if (!of_device_is_available(np)) continue;
    <... when != goto l; ... np
(
-   {
      of_node_put(np);
      S
-   }
|
-   of_node_put(np);
)
    ...>
  }
  ... when != np
```

Semantic patch for `for_each_available_child_of_node_scoped`

```
@@
identifier np;
iterator name for_each_child_of_node, for_each_available_child_of_node_scoped;
expression node; statement S;
@@

- struct device_node *np;
  ... when != np
- for_each_child_of_node
+ for_each_available_child_of_node_scoped
       (node, np) {
-    if (!of_device_is_available(np)) continue;
     <... when != goto l; ... np
(
-    {
       of_node_put(np);
       S
-    }
|
-    of_node_put(np);
)
     ...>
  }
  ... when != np
```

Updates 6 loops.

More variants

Other device node iterators don’t yet provide scoped variants.

- `for_each_node_by_name`, `for_each_matching_node`, etc.
- When they are available, semantic patches can be written for them as well.

Interactivity

Device-node availability

Availability is not always checked in the same way:

```
for_each_child_of_node(np, child) {
        if (of_match_node(driver->subdevs, child) &&
            of_device_is_available(child)) {
                err = host1x_subdev_add(device, driver, child);
                if (err < 0) {
                        /* XXX cleanup? */
                        of_node_put(child);
                        return err;
                }
        }
}
```

Interactivity via python

```
@r exists@
identifier np;
iterator name for_each_child_of_node;
expression node;
position p;
@@

  struct device_node *np;
  ... when != np
  for_each_child_of_node(node, np) {
    <...
    of_device_is_available@p(np)
    ...>
  }
  ... when != np

@script:python@
p << r.p;
@@

print("of_device_is_available in",p[0].file,"on line",p[0].line)
```

Interactivity via python

```
@r exists@
identifier np;
iterator name for_each_child_of_node;
expression node;
position p;
@@

  struct device_node *np;
  ... when != np
  for_each_child_of_node(node, np) {
    <...
    of_device_is_available@p(np)
    ...>
  }
  ... when != np

@script:python@
p << r.p;
@@

print("of_device_is_available in",p[0].file,"on line",p[0].line)
```

Reports the positions of 25 loops for manual inspection.

Conclusion

- User-friendly tool for updating C code at large scale.
  - Over 9000 Coccinelle-based commits in the Linux kernel.
- Ease of development and evolution of transformation rules.
- Many opportunities for simplifying Linux kernel resource management.

https://coccinelle.gitlabpages.inria.fr/website/