Coccinelle: Reducing the Barriers to Modularization in a Large C Code Base

Julia Lawall
Inria/LIP6/UPMC/Sorbonne University-Regal

Modularity 2014

Modularity

Wikipedia:
Modularity is the degree to which a system's components may be separated and recombined.

- A well-designed system (likely) starts with a high degree of modularity.
- Modularity must be maintained as a system evolves.
- Evolution decisions may be determined by the impact on modularity.

Goal: Maintaining modularity should be easy as a system evolves.

Modularity and API functions

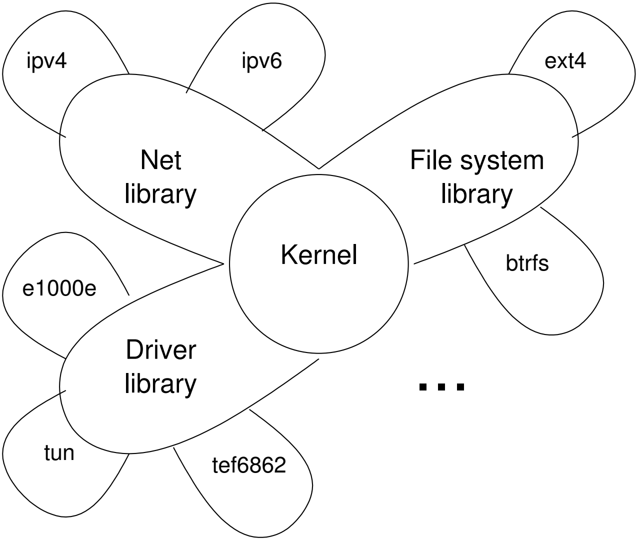
Well designed API functions can improve modularity

- Hide module-local variable names.
- Hide module-local function protocols.

Problem:

- The perfect API may not be apparent in the original design.
- The software may evolve, making new APIs needed.
- Converting to new APIs is hard.

Modularity in the Linux kernel



Case study: Memory management in Linux

Since Linux 1.0, 1994:

- `kmalloc`: allocate memory
- `memset`: clear memory
- `kfree`: free memory

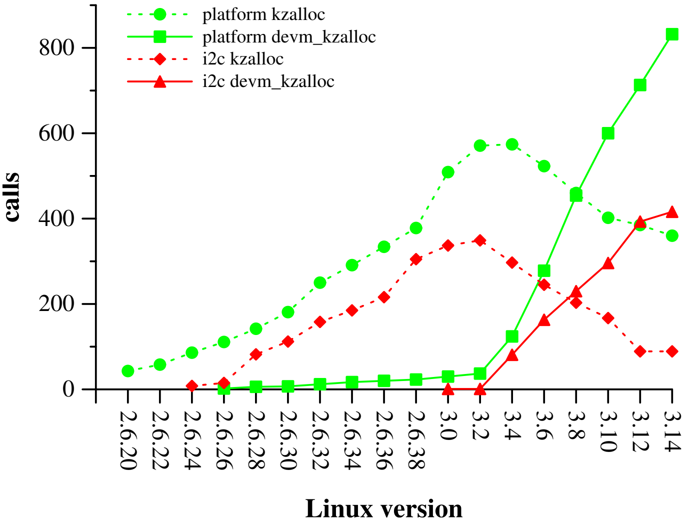
Since Linux 2.6.14, 2006:

- `kzalloc`: allocate memory
- `kfree`: free memory
- No separate clearing, but need explicit free.

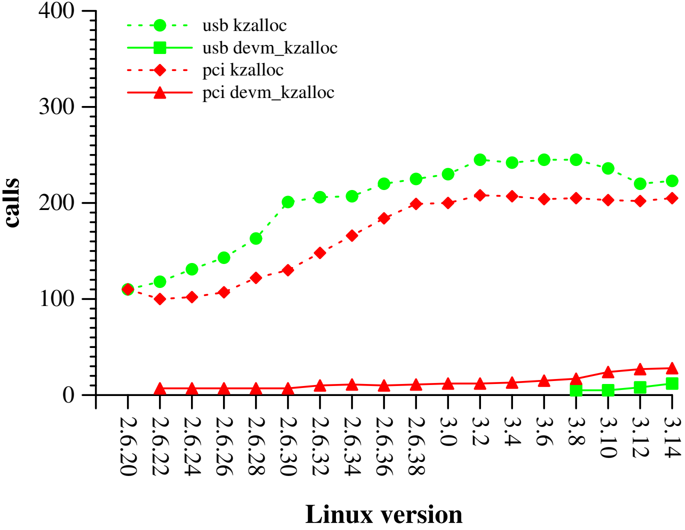
Since Linux 2.6.21, 2007:

- `devm_kzalloc`: allocate memory
- No explicit free.

API introduction in practice: devm_kzalloc



API introduction in practice: devm_kzalloc



Adoption challenges

Partial patch introducing devm_kzalloc:

```
- rfkill_data = kzalloc(sizeof(*rfkill_data), GFP_KERNEL);
+ rfkill_data = devm_kzalloc(&pdev->dev, sizeof(*rfkill_data), GFP_KERNEL);
  if (rfkill_data == NULL) {
    ret = -ENOMEM;
    goto err_data_alloc;
  }
  rf_kill = rfkill_alloc(...);
  if (rf_kill == NULL) {
    ret = -ENOMEM;
-   goto err_rfkill_alloc;
+   goto err_data_alloc;
  }
  ...
  return 0;

 err_rfkill_register: rfkill_destroy(rf_kill);
-err_rfkill_alloc:    kfree(rfkill_data);
 err_data_alloc:      regulator_put(vcc);
 out:                 return ret;
```

Summary of changes

- `devm_kzalloc` replaces `kzalloc`
- `devm_kzalloc` needs a parent argument.
  - `kzalloc(e1,e2)` becomes `devm_kzalloc(dev,e1,e2)`
- The allocated value must live from the initialization to the removal of the driver.
- `kfree`s on the allocated value should be removed.

Remaining changes

- Also have to adjust the remove function.
- `regulator_put` also has a devm variant.
  - Should fix that too.

Issues

- The API is not sufficiently well known.
- The conditions required for introducing the API are complex.
- The changes required are tedious and error prone.
- Relevance to different kinds of actors:
  - For the developer:
    How to find and fix potential uses of the new API?
  - For the manager:
    How to assess the adoption of the new API?
  - For the maintainer:
    How to find and fix faults in the use of the new API?
- All need to know precisely how the API should be used.

Coccinelle to the rescue

- Matching and transformation for unpreprocessed C code.
- Developer-friendly scripting, based on patch notation
  - semantic patches.
- Applicable to large code bases.
  - The Linux kernel (12 MLOC).
- Available in major Linux distributions.

http://coccinelle.lip6.fr/

http://coccinellery.org/

For the developer: Issues to address

Pb1. `devm_kzalloc` replaces `kzalloc`

Pb2. `devm_kzalloc` needs a parent argument.

- `kzalloc(e1,e2)` becomes `devm_kzalloc(dev,e1,e2)`

Pb3. The allocated value must live from the initialization to the removal of the driver.

Pb4. `kfrees` on the allocated value should be removed.

Pb1. devm_kzalloc replaces kzalloc

```
@@
expression e, e1, e2;
@@

- e = kzalloc(e1, e2)
+ e = devm_kzalloc(dev, e1, e2)
```

Pb1. devm_kzalloc replaces kzalloc

```
@@
expression e, e1, e2;
@@

- e = kzalloc(e1, e2)
+ e = devm_kzalloc(dev, e1, e2)
```

Where does dev comes from?

Pb2. Obtaining a `dev` value

`devm_kzalloc` can only be used with drivers that build on libraries that manage memory.

- Examples: platform driver, i2c driver, usb driver, pci driver.

These libraries pass to the driver probe function a dev value.

Pb2. Obtaining a dev value

```
@@
identifier probefn, pdev;
expression e, e1, e2;
@@
probefn(struct platform_device *pdev, ...) {
  <+...
- e = kzalloc(e1, e2)
+ e = devm_kzalloc(&pdev->dev, e1, e2)
  ...+>
}
```

Pb2. Obtaining a dev value

```
@@
identifier probefn, pdev;
expression e, e1, e2;
@@
probefn(struct platform_device *pdev, ...) {
  <+...
- e = kzalloc(e1, e2)
+ e = devm_kzalloc(&pdev->dev, e1, e2)
  ...+>
}
```

How to be sure that probefn is a probe function?

Pb2. Obtaining a dev value

```
@platform@
identifier s, probefn;
@@
struct platform_driver s = {
  .probe = probefn,
};

@@
identifier platform.probefn, pdev;
expression e, e1, e2;
@@
probefn(struct platform_device *pdev, ...) {
  <+...
- e = kzalloc(e1, e2)
+ e = devm_kzalloc(&pdev->dev, e1, e2)
  ...+>
}
```

Pb3. Lifetime of the allocated value

Issues:

- Using devm functions, allocated values are live until after the driver remove function.
- To preserve the same behavior, have to check all the other functions for kfrees.
- Simplifying assumption: `kzalloced` data in the probe function is live until the remove function.
  - This assumption can be removed using a more complex Coccinelle rule.

Pb4. Removing `kfrees`

Where are they?

- Failure of probe function.
- Success of remove function.

Which ones to remove?

- Simplifying assumption: An allocated value is always referenced in the same way.
- This assumption can be partially removed using a more complex Coccinelle rule.

Pb4. Removing `kfrees`: Find the remove function

```
@platform@
identifier s, probefn, removefn;
@@
struct platform_driver s = {
  .probe = probefn,
  .remove = removefn,
};
```

Pb4. Remove kfrees from probe

```
@platform@
identifier s, probefn, removefn;
@@
struct platform_driver s = {
  .probe = probefn,
  .remove = removefn,
};

@prb@
identifier platform.probefn, pdev; expression e, e1, e2;
@@
probefn(struct platform_device *pdev, ...) {
  <+...
- e = kzalloc(e1, e2)
+ e = devm_kzalloc(&pdev->dev, e1, e2)
  ...
?-kfree(e);
  ...+>
}
```

Pb4. Remove kfrees from remove

```
@platform@ identifier s, probefn, removefn; @@
struct platform_driver s = { .probe = probefn, .remove = removefn, };

@prb@ identifier platform.probefn, pdev; expression e, e1, e2; @@
probefn(struct platform_device *pdev, ...) {
  <+...
- e = kzalloc(e1, e2)
+ e = devm_kzalloc(&pdev->dev, e1, e2)
  ...
?-kfree(e);
  ...+>
}

@rem depends on prb@ identifier platform.removefn; expression e; @@
removefn(...) {
  <...
- kfree(e);
  ...>
}
```

Proposes updates to 261 platform drivers

For the Manager: How to assess adoption of the new API?

Coccinelle supports not only transformation, but also other program matching tasks.

Idea:

- Search for the pattern as for transformation.
- Record the position of relevant information.
- Use python or ocaml scripting to process the recorded information.
  - Make charts and graphs.
  - Update a database.
  - Send reminder letters, etc.

For the Manager: How to assess adoption of the new API?

```
@initialize:python@ @@
count = 0

@platform@ identifier s, probefn; @@
struct platform_driver s = { .probe = probefn, };

@prb@
identifier platform.probefn, pdev; expression e, e1, e2; position p;
@@
probefn@p(struct platform_device *pdev, ...) {
  <+...
  e = kzalloc(e1, e2)
  ...+>
}

@script:python@ p << platform.p; @@
count = count + 1

@finalize:python@ @@
print count
```

For the maintainer: Finding faults in API usage

- `devm_kzalloc` + `kfree` is forbidden.
- `devm_kzalloc` + `devm_kfree` should be unnecessary.
- Both may result from a misunderstanding of how `devm_kzalloc` works.

Differentiated finding fault, part 1

```
@r exists@
expression e,e1;
position p;
@@
e = devm_kzalloc(...)
... when != e = e1
( kfree@p | devm_kfree@p ) (e)

@script:ocaml@
p << r.p;
@@
let p = List.hd p in
Printf.printf "Very suspicious free: line %d of file %s"
  p.line p.file
```

Differentiated finding fault, part 2

```
@s exists@
expression r.e;
position p != r.p;
@@
... when != e = kmalloc(...)
    when != e = kzalloc(...)
( kfree@p | devm_kfree@p ) (e)

@script:ocaml@
p << s.p;
@@
let p = List.hd p in
Printf.printf "Possibly suspicious free: line %d of file %s"
  p.line p.file
```

5 “possibly” reports, 3 are probable bugs.

Conclusion

- Declarative matching and transformation language.
- Mostly C-like. No large reference manual.
- Reduces the barrier to improvements that require repetitive changes.
- Versatile: developers, managers, maintainers.
  - Possibility to reuse specifications for multiple roles.
- Accessible to ordinary developers.
  - Almost 2000 patches in the Linux kernel motivated by Coccinelle, including patches by around 90 developers from outside our research group.