Restructuring Scientific Software using Semantic Patching with Coccinelle

Michele MARTONE

Leibniz Supercomputing Centre
Garching bei München, Germany

Potsdam, de-RSE Conference
June 4, 2019

de-RSE@Potsdam, 04–06.06.2019 workshop abstract

Restructuring Scientific Software using Semantic Patching with Coccinelle

2019-06-04, 18:00–19:15, A31 West

Maintenance of a large HPC software in C/C++ can be demanding. Factors like evolving 3rd-party APIs and hardware require significant efforts to project sustainability. Failure in coping with these challenges can lead to obsolescence, performance loss, vendor lock-in, bugs.

This workshop introduces the 'Coccinelle' tool for semantics-aware matching and patching of C code. While initially conceived for automatically keeping up-to-date Linux kernel drivers, Coccinelle has been underexplored in other contexts. Here, emphasis will be given on code restructuring for High Performance Computing (HPC) codes in support to domain scientists. Coccinelle can also be a powerful testing tool. Discussion and experience exchange is welcome.

https://derse19.uni-jena.de/derse19/talk/URQ7X3/

A word of caution

Code optimization and Coccinelle

- Code optimization is tricky. Coccinelle can be tricky, too.
- This is NOT a talk to teach you optimization or Coccinelle.
- This is a talk about how optimizations might be implemented by means of Coccinelle's *rewrite rules*.

For code optimization courses, take a look elsewhere, e.g.

- `https://www.lrz.de/services/compute/courses`
- `http://www.prace-ri.eu/ptcs`

For Coccinelle, one-day training: 08.10.2019 at LRZ:

- `https://www.lrz.de/services/compute/courses/2019-10-08_hspc1w19/`

what is this all about ?

- *automating* (oh well: *scripting*) code restructuring
- ...for HPC
- (but also for anything else)

STOP!

Why automate that ?

Let's see...

which *sequential* access is faster ?

```
struct ptcl_t {
 double X, Y;
 double P;
};
  ...
struct ptcl_t aos[N];

   ...
 for(i=0;i<N;++i)
   aos[i].P =
     f(aos[i+1].X +
       aos[i-1].X +
          ... );
```

```
struct ptcla_t {
 double X[N],Y[N];
 double P[N];
};
  ...
struct ptcla_t soa;

   ...
 for(i=0;i<N;++i)
   soa.P[i] =
     f(soa.X[i+1] +
       soa.X[i-1] +
          ...);
```

Array of Structures?

Structure of Arrays?

which *sequential* access is faster ?

```
struct ptcl_t {
 double X, Y;
 double P;
};
  ...
struct ptcl_t aos[N];

   ...
 for(i=0;i<N;++i)
   aos[i].P =
     f(aos[i+1].X +
       aos[i-1].X +
          ... );
```

```
struct ptcla_t {
 double X[N],Y[N];
 double P[N];
};
  ...
struct ptcla_t soa;

   ...
 for(i=0;i<N;++i)
   soa.P[i] =
     f(soa.X[i+1] +
       soa.X[i-1] +
          ...);
```

Not AoS...

...SoA vectorizes better!

A relevant motivating problem: GADGET simulation code

- Cosmological large-scale structure formation (galaxies and clusters)
- Highly scalable ($O(100k)$ Xeon cores on SuperMUC@LRZ)
- Several teams and versions (≥100 kLoC each)

A relevant motivating problem: GADGET simulation code

- Cosmological large-scale structure formation (galaxies and clusters)
- Highly scalable ($O(100k)$ Xeon cores on SuperMUC@LRZ)
- Several teams and versions (>100 kLoC each)

Refactoring for node-level performance

```
struct particle {
  double Mass, Hsml, ...;
};

...
// Array of Structures
struct particle *P;

...
// may not vectorize
P[i].Mass + P[i]...
```

A relevant motivating problem: GADGET simulation code

- Cosmological large-scale structure formation (galaxies and clusters)
- Highly scalable ($O(100k)$ Xeon cores on SuperMUC@LRZ)
- Several teams and versions (≥100 kLoC each)

Refactoring for node-level performance

```c
struct particle {
  double Mass, Hsml, ...;
};

...
// Array of Structures
struct particle *P;

...
// may not vectorize
P[i].Mass + P[i]...
```

$\Longrightarrow$

```c
struct particle_soa_t {
  double *Mass, *Hsml, ...;
};

...
// Structure of Arrays
struct particle_soa_t P_SoA;

...
// vectorizes better
P_SoA.Mass[i] + P_SoA...
```

A relevant motivating problem: GADGET simulation code

- Cosmological large-scale structure formation (galaxies and clusters)
- Highly scalable ($O(100k)$ Xeon cores on SuperMUC@LRZ)
- Several teams and versions (≥100 kLoC each)

Refactoring for node-level performance

```c
struct particle {
  double Mass, Hsml, ...;
};

...
// Array of Structures
struct particle *P;

...
// may not vectorize
P[i].Mass + P[i]...
```

⇒

```c
struct particle_soa_t {
  double *Mass, *Hsml, ...;
};

...
// Structure of Arrays
struct particle_soa_t P_SoA;

...
// vectorizes better
P_SoA.Mass[i] + P_SoA...
```

How do you do this cleanly ?

Coccinelle (http://coccinelle.lip6.fr)

Coccinelle *"...a program matching and transformation engine ... for specifying desired matches and transformations in C code"*

source to source translation

- arbitrary transformations of C code

refactoring

- making program structure easier to understand

spotting bugs

- detect bad code patterns (e.g. spot missing `free()`)

…*semantic patching* with Coccinelle!

"…engine for specifying desired **matches** *and* **transformations** *in C code"*

...*semantic patching* with Coccinelle!

"...engine for specifying desired **matches** *and* **transformations** *in C code"*

Example AoS ⇒ SoA *conversion rules*

```
@@
identifier id,I;
type T;
@@
struct id { ...
- T  I;
+ T *I;
  ...
};
```

```
@@
expression E;
identifier AoS,J;
fresh identifier SoA=AoS##"_SoA";
@@
- AoS[E].J
+ SoA.J[E]
```

...*semantic patching* with Coccinelle!

"...engine for specifying desired **matches** *and* **transformations** *in C code"*

Example AoS ⇒ SoA *conversion rules*

```
@@
identifier id,I;
type T;
@@
struct id { ...
- T  I;
+ T *I;
  ...
};
```

```
@@
expression E;
identifier AoS,J;
fresh identifier SoA=AoS##"_SoA";
@@
- AoS[E].J
+ SoA.J[E]
```

Strengths

- **Generality**: multiple code forks, if semantic structures match
- **Flexibility**: conversion can be partial
- **Consistency**: patch only if semantic model satisfied

Contents overview

Description
Intro
Invocation
SmPL crash course
Example use cases
Outro
Reminder: LRZ Coccinelle Training

Story of Coccinelle: a bugs' story

- a project from INRIA (France)
- appeared in 2006
- originally for
  - *collateral evolutions* in Linux kernel drivers[1]
  - **smashing bugs** (hence the name)[2]

[1] https://git.kernel.org/pub/scm/linux/kernel/git/backports/backports.git/tree/patches

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/coccinelle

Another word of caution

Limitations

Coccinelle was born to serve the Linux kernel community.
It was not thought to cover **all** *the possible C modification needs.*

But

...is **incredibly versatile**, and in active development!

Version used here:

1.0.7-00151-ga48bc27d compiled with OCaml version 4.06.0

Coccinelle for HPC ?

- C to C code translation!
- might assist when several forks exist:
  - HPC expert gets a code branch / snapshot
  - develops a series of semantic patches
  - consults with code authors / community
  - *backports* (brings back to the original) at the very end of the optimization activity time frame

Possible *collateral evolutions* in HPC

- API change and necessary update
- introducing specific `pragma` directives
- Keyword add
- Keyword remove
- introducing *intrinsics*
- simplifying expressions
- AoS $\Rightarrow$ SoA
- SoA $\Rightarrow$ AoS
- parallelization: serial to OpenMP-parallel
- parallelization: serial to MPI-parallel
- serialization: removing OpenMP
- serialization: removing MPI

Further possible applications in HPC

- produce statistics and reports, analysis
  - e.g. of API misuse (bugs)
  - detecting notoriously inefficient code patterns
- C $\Rightarrow$ C++ transition (e.g. cast after `malloc,calloc`)

Semantic patching invocation

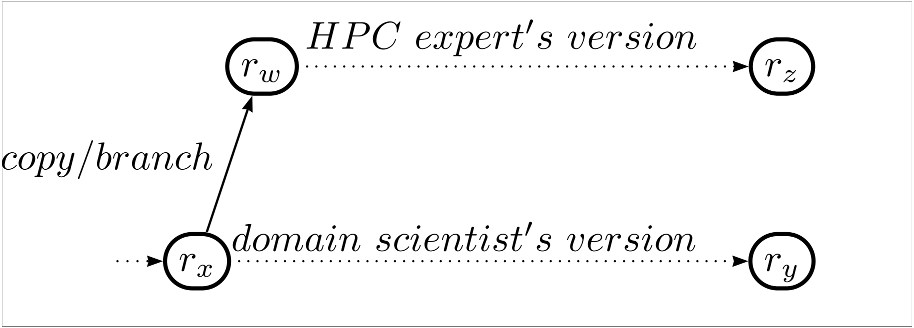
- identify a C file to be changed, say: `f.c`
- write a semantic patch representing the change:
  `$EDITOR sp.cocci`
- apply:

```
# produce patch:
spatch --sp-file sp.cocci f.c > sp.diff
# apply patch:
patch < sp.diff # this patches f.c
```

Important switches

```
spatch ...
  -j # threaded parallel

  --parse-cocci # parse rules

  --parse-c # parse C source

  --verbose

  --verbose-parsing

  --debug

  --local-includes # C headers

  --recursive-includes # C headers
```

"can you optimize my code ?"



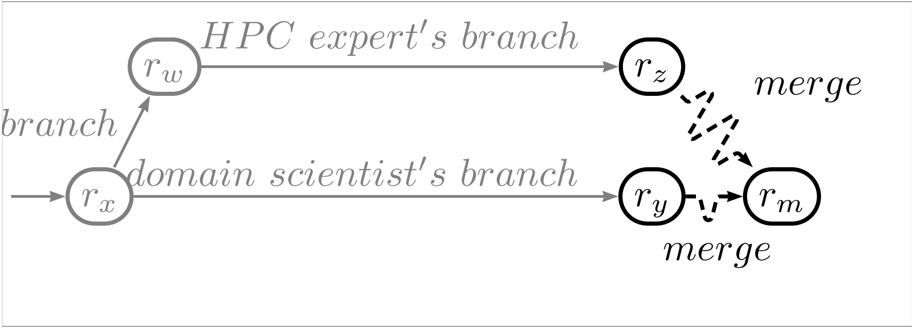
Possible workflow agreement

1. determine a "starting" relevant code snapshot

2.A. domain expert continues on usual development line

2.B. HPC expert works on another

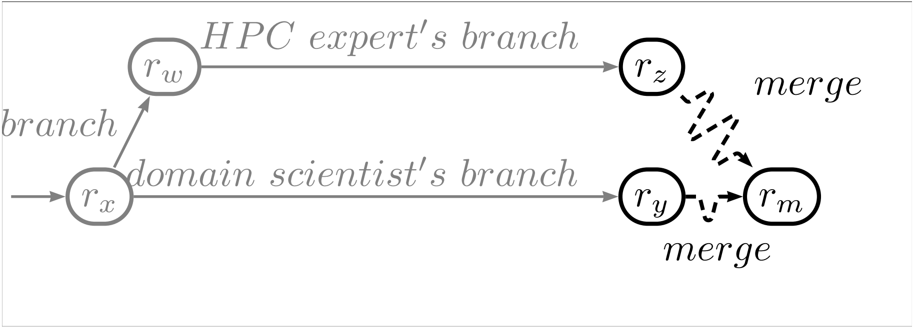
branch and *merge*



Possible workflow

- the two parties can work independently
- weeks to months pass
- at some point, performance-enhancing changes need *merge*

Backport / merge may be problematic



Merge ? OK if branches did not diverge too much

- what if say, **every second line** changed ?
- would you accept such a large “patch” to your code ?

Possible *performance patch engineering* workflow

Develop e.g. a data layout change codified in *semantic patches*.
Maintain them together with sources.

```
project/
├── f1.c
├── f2.c
├── patch.cocci
├── Makefile
└── ...
```

⇒ patch code ⇒

```
project/
├── f1.c.bak
├── f2.c.bak
├── patch.cocci
├── Makefile
├── f1.patch
├── f2.patch
├── f1.c
├── f2.c
└── ...
```

Measure new code performance.
Change original sources if really needed.

Caution: Coccinelle (as any tool) assumes decent code!

- .c files `#include` 'ing other .c files, not `#include` 'ing required headers
- non-well-behaved `#ifdef` branches leading to
  - unbalanced brackets
  - broken expressions
  - further inconsistencies
- please follow **any** convention of good coding and code structuring
  - keep functions sanely *short* (no multi-KLOC monsters!)

Caution: Coccinelle (as any tool) assumes decent code!

- .c files `#include` 'ing other .c files, not `#include` 'ing required headers
- non-well-behaved `#ifdef` branches leading to
  - unbalanced brackets
  - broken expressions
  - further inconsistencies
- please follow **any** convention of good coding and code structuring
  - keep functions sanely *short* (no multi-KLOC monsters!)

Consider example in following slides

Imagine we wish to transform all expressions as:

```
SphP[i].Metals, ...
```

into

```
SphP_soa.Metals[i], ...
```

...

Original AoS code, text editor view

```

#ifndef LT_METAL_COOLING_on_SMOOTH_Z
  Z = get_metallicity_solarunits(get_metallicity(i, Iron));
#else
  double metalmass = get_metalmass(SphP[i].Metals);
  if(metalmass > 0)
#ifdef LT_ZSMOOTH_ALLMETALS
    Z = get_metallicity_solarunits(SphP[i].Zsmooth[Iron]);
#else
    Z = get_metallicity_solarunits(SphP[i].Zsmooth * SphP[i].Metals
        [Iron] / metalmass);
#endif
  else
    Z = NO_METAL;
#endif
#endif
```

Is this what compiler sees ?

Original AoS code, compiler view

Assume both preprocessor symbols defined.

```




double metalmass = get_metalmass(SphP[i].Metals);
if(metalmass > 0)

  Z = get_metallicity_solarunits(SphP[i].Zsmooth[Iron]);



else
  Z = NO_METAL;
```

Compiler parses *preprocessed* code.

And so shall Coccinelle, right ?

SoA on *parsable* code, result view

```
#ifndef LT_METAL_COOLING_on_SMOOTH_Z
  Z = get_metallicity_solarunits(get_metallicity(i, Iron));
#else
  double metalmass = get_metalmass(SphP[i].Metals); // OK
  if(metalmass > 0)
#ifdef LT_ZSMOOTH_ALLMETALS
    Z = get_metallicity_solarunits(SphP_soa.Zsmooth[i][Iron]); // OK
#else
    Z = get_metallicity_solarunits(SphP[i].Zsmooth * SphP[i].Metals[Iron] / metalmass); // NOT OK!
#endif
  else
    Z = NO_METAL;
#endif
#endif
```

Is this what we want ?

No!

But.. can we afford defining each combination ?

Target code: Zsmooth as SoA

```
#ifndef LT_METAL_COOLING_on_SMOOTH_Z
  Z = get_metallicity_solarunits(get_metallicity(i, Iron));
#else
  double metalmass = get_metalmass(SphP[i].Metals); // OK
  if(metalmass > 0)
#ifdef LT_ZSMOOTH_ALLMETALS
    Z = get_metallicity_solarunits(SphP_soa.Zsmooth[i][Iron]); //
        OK
#else
    Z = get_metallicity_solarunits(SphP_soa.Zsmooth[i] * SphP_soa[i
        ].Metals[Iron] / metalmass); // OK
#endif
  else
    Z = NO_METAL;
#endif
#endif
```

All preprocessor combinations... but how ?

Can't we just ignore #ifdefs ?

```
Z = get_metallicity_solarunits(get_metallicity(i, Iron));

double metalmass = get_metalmass(SphP[i].Metals);
if(metalmass > 0) // NOT OK: unparsable if construct

  Z = get_metallicity_solarunits(SphP_soa.Zsmooth[i][Iron]);

  Z = get_metallicity_solarunits(SphP_soa.Zsmooth[i] * SphP_soa[i
      ].Metals[Iron] / metalmass);

else   // NOT OK: two statements before 'else'
  Z = NO_METAL;
```

No.

We want *well-behaved* `ifdef` branches!

Like here:

```
---
+++
@@ -1584,11 +1584,13 @@
 #else
   double metalmass = get_metalmass(SphP[i].Metals);
   if(metalmass > 0)
+  {
 #ifdef LT_ZSMOOTH_ALLMETALS
     Z = get_metallicity_solarunits(SphP[i].Zsmooth[Iron]);
 #else
     Z = get_metallicity_solarunits(SphP[i].Zsmooth * SphP[i].
         Metals[Iron] / metalmass);
 #endif
+  }
   else
     Z = NO_METAL;
 #endif
```

What does that mean ?

Code correctly *parsable* even if `#ifdefs` ignored

Parsable code = transformable code.

```
double metalmass = get_metalmass(SphP[i].Metals);
if(metalmass > 0)
{

  Z = get_metallicity_solarunits(SphP[i].Zsmooth[Iron]);

  Z = get_metallicity_solarunits(SphP[i].Zsmooth * SphP[i].Metals
      [Iron] / metalmass);

}
else
  Z = NO_METAL;
```

Need a bit care and coordination during programming.

(unless you want to repeat semantic patch application with each legal combination of defined preprocessor symbols – likely not).

Coccinelle *rules* and *transformations*

- `@rules@` can **match** context, **insert** code, or **delete** it
- follow respective rules of *minus* and *plus* code
- match-only code is called *context*

```
@myrule@
@@
//context:
 a=0;

-a=0; // minus code
+a=1; // plus code

// comment insertion
+// a=0;
```

Metavariables

SmPL variables to **match** and **remove** / **manipulate**:

- tokens as: `symbol`, `constant`, `identifier`, `operator`, `type`, ...
- `expressions` and `statements`
- portions of other, structured C entities as `struct`'s or `union`'s ...
- `positions` in the code, a `format` string, ...
- occurs in plus/minus code and context

```
@@
identifier I =~ "i|j";
binary operator o;
type T = {int,double};
@@
-T I;//match & remove
 ...
-I o I;
```

- instantiate when parsed C entity *matches*
- no match, no instance
- certain metavariables' values can be **whitelisted** or **blacklisted**

Metavariable for a structure's `field`

- match for *fields* in structs
- allow
  - restructure existing structs,
  - create ad-hoc ones

```
@@
field lfld;
field list [n={2}] f2fld;
@@
struct str_t {
- f2fld
  ...
- lfld
};
+ struct l_t { f2fld lfld };
```

e.g. match and move selected

- `field`s, or
- `field list`s

Inheritance

- a rule can use another, already matched rule's *bound* metavariables
- dependency across rules

```
@r1@
identifier I;
@@
 I=0;

@r2@
identifier r1.I;
@@
-I--;

@r3@
identifier r1.I;
@@
 a=b+c;
+I++;
```

- inherited `identifier`s can be matched negatively with `!=`

Scripting

- many *internals* are accessible
- via `script:python` or `script:ocaml`

```
@r@
// metadecls
@@
// normal rule ...

@script:python p@
// variables binding
I << r.I;
N; // new variables
@@
// python code using I and N

@@
identifier r.I;
identifier p.N;
@@
// normal rule ...
```

Left: stateless Python scripting usage.
Below: stateful Python scripting usage.

```
@initialize:python@
@@
// python code ...

@script:python@
I << r.I;
// ...
@@
// python code using  ...

@finalize:python@
@@
// python code ...
```

What now ?

Interesting part starts now

Real-life Coccinelle rules use all the features shown so far combined

Think of the following simplified use cases as *building blocks*

Insert statement after local variables declarations, naive

```
@@
declaration D;
statement S;
@@
 D
+printf("in %s\n",__FUNCTION__);
 S
```

```
--- cex_stmt_after_decl.c
+++ cex_stmt_after_decl.patched.c
@@ -1,12 +1,15 @@
 void v() { return; }

 int f(int i) { int j;
+ printf("in %s\n", __FUNCTION__);
  return i+j; }

 int f(int i) { int j,k;
+ printf("in %s\n", __FUNCTION__);
  return i+j+k; }

 int main() {
  int i; int j;
+ printf("in %s\n", __FUNCTION__);
  i=0; j=i; v( ); f(j);
 }
```

Insert statement after local variables declarations

```
@@
identifier F;
statement S1,S2;
@@
F(...) {
 ... when != S1
+printf("in %s\n",__FUNCTION__);
 S2
 ... when any
}
```

```
--- cex_stmt_after_decl2.c
+++ cex_stmt_after_decl2.patched.c
@@ -1,12 +1,16 @@
-void v() { return; }
+void v() { printf("in %s\n",
     __FUNCTION__);
+       return; }

 int f(int i) { int j;
+ printf("in %s\n", __FUNCTION__);
  return i+j; }

 int f(int i) { int j,k;
+ printf("in %s\n", __FUNCTION__);
  return i+j+k; }

 int main() {
  int i; int j;
+ printf("in %s\n", __FUNCTION__);
  i=0; j=i; v( ); f(j);
 }
```

example name: cex_stmt_after_decl2

Transfer function contents

```
@r1@
statement list sl;
@@
 int main() {
- sl
+ sub_main( );
}

@r2@
statement list r1.sl;
@@
int main (...) {...}
+ void sub_main( ) { sl }
```

```
--- cex_stmt_f2f.c
+++ cex_stmt_f2f.patched.c
@@ -1,4 +1,10 @@
 int main() {
+  sub_main();
+}
+
+void sub_main()
+{
   1;
-  if(2) 1;
+  if (2)
+    1;
 }
```

example name: cex_stmt_f2f

Clone specialized versions of function

AoS to SoA: variables selection

```
@@
identifier M = {X,Y};
fresh identifier G="g_"##M;
type T;
@@
struct ptcl_t { ...
-T M;
 ...
};
++T G[N];
```

```
--- cex_aos_to_soa1.c
+++ cex_aos_to_soa1.patched.c
@@ -1,11 +1,13 @@
 #define N 3
 struct ptcl_t {
   int x,y,z;
-  double X,Y,Z;
+  double Z;
 };
+double g_X[N];
+double g_Y[N];


 int main() {
   struct ptcl_t aos[N];
   // ...
 }
```

example name: cex_aos_to_soa1

First: rules to create data structure

Note: “right” variable mix is application dependent.

AoS to SoA: declarations and use

```
@r@
identifier M = {X,Y};
fresh identifier G="g_"##M;
symbol N;
type T;
@@
struct ptcl_t {
-T M;
};
++T G[N];

@@
identifier r.M,P,r.G;
typedef ptcl_t;
expression E;
constant N;
@@
struct ptcl_t P[N];
...
-P[E].M
+G[E]
```

```
--- cex_aos_to_soa2.c
+++ cex_aos_to_soa2.patched.c
@@ -1,11 +1,13 @@
 #define N 3
 struct ptcl_t {
-  double X,Y,Z;
+  double Z;
 };
+double g_X[N];
+double g_Y[N];


 int main() {
   struct ptcl_t aos[N];
-  aos[0].X = aos[0].Y
+  g_X[0] = g_Y[0]
             + aos[0].Z;
 }
```

example name: cex_aos_to_soa2

Second: update expressions accordingly

Iterative method and *recovery*

```
@@
identifier X,A,Y;
fresh identifier Z=X##"_rec";
@@
v_t X;
+v_t Z; // CG recovery vector
m_t A;
...
 X= A* X;
+//post-mult CG recovery code
...
 Y= norm(X);
+//post-norm CG recovery code
```

```
--- cex_cg1.c
+++ cex_cg1.patched.c
@@ -1,11 +1,14 @@
 // extract from a iterative method
 typedef int m_t;
 typedef int v_t;
 int norm(v_t v) { return 0; }
 int main() {
  v_t v,p;
+ v_t p_rec; // CG recovery vector
  m_t A;
  p= A*p;
+ //post-mult CG recovery code
  v= A*p;
  v=norm(p);
+ //post-norm CG recovery code
 }
```

example name: cex_cg1

Instead of comments, specific functions calls here

(see e.g. Jaulmes et al., 2015)

Detect variable use and change its type

```
@vr@
identifier V;
type NT={double};
@@
 NT *V;

@br@
identifier vr.V;
identifier I,J,N,M;
identifier ins_fun=~"insert";
@@
 ins_fun(M, N, V, I, J)

@dr depends on br@
identifier vr.V;
type vr.NT;
@@
-NT    *V;
+float *V;
```

```
--- cex_var_type_change.c
+++ cex_var_type_change.patched.c
@@ -1,16 +1,16 @@
 #include <blas_sparse.h>
 int main() { // ...
   int nnz;
   int*IA,*JA;
   float *FV;
   double*DV;
-  double*NV;
+  float *NV;
   // ...
   BLAS_uscr_insert_entries(A, nnz, FV,
     IA, JA);
   BLAS_usgt_entries       (A, nnz, DV,
     IA, JA);
   BLAS_uscr_insert_entries(A, nnz, NV,
     IA, JA);
   // ...
 }
```

example name: cex_var_type_change

Precision increase/decrease

http://www.netlib.org/blas/blast-forum/

Functions modifying variable

```
@@
identifier F;
type R,T;
parameter list p;
global idexpression T I = {
    a};
expression E;
assignment operator ao;
@@
+ // modifies a:
R F(p)
 {
 <+...
 I ao E
 ...+>
 }
```

```
--- cex_func_mod_var_1.c
+++ cex_func_mod_var_1.
    patched.c
@@ -1,7 +1,8 @@
 int a,b;
 int g() { b=a; }
+// modifies a:
 int f() { a=b; }
 int h() { f( ); g( ); }
 int l() { h( ); g( ); }
 int i() { h( ); l( ); }
 int main() { i( ); }
```

example name: cex_func_mod_var_1

Functions modifying variable, again

```
@mf@
identifier F;
type R,T;
parameter list p;
global idexpression T I = {a};
expression E;
assignment operator ao;
@@
R F(p)
 {
 <+...
 I ao E
 ...+>
 }

@@
identifier mf.F,F1;
type R;
@@
+ // calls a function modifying a:
R F1(...)
 {
 <+...
 F(...);
 ...+>
 }
```

```
--- cex_func_mod_var_2.c
+++ cex_func_mod_var_2.patched.c
@@ -1,7 +1,8 @@
 int a,b;
 int g() { b=a; }
 int f() { a=b; }
+// calls a function modifying a:
 int h() { f( ); g( ); }
 int l() { h( ); g( ); }
 int i() { h( ); l( ); }
 int main() { i( ); }
```

example name: cex_func_mod_var_2

Investigate tricky missing synchronization

Functions modifying variable, and again

```
@m0@
identifier F0;
type R,T;
parameter list p;
global idexpression T I = {a};
expression E;
assignment operator ao;
@@
R F0(p) { ...  I ao E; ...  }

@m1@
identifier m0.F0,F1;
type R;
@@
R F1(...) { ...  F0(...); ...  }

@m2@
identifier m1.F1,F2;
type R;
@@
+ // calls a function calling a function
     modifying a:
R F2(...) { ...  F1(...); ...  }
```

```
--- cex_func_mod_var_3.c
+++ cex_func_mod_var_3.patched.c
@@ -1,7 +1,9 @@
 int a,b;
 int g() { b=a; }
 int f() { a=b; }
 int h() { f( ); g( ); }
+// calls a function calling a function
     modifying a:
 int l() { h( ); g( ); }
+// calls a function calling a function
     modifying a:
 int i() { h( ); l( ); }
 int main() { i( ); }
```

example name: cex_func_mod_var_3

Investigate trickier missing synchronization

Identifying recursive functions

```
@m0@
identifier F0;
type R;
parameter list p;
@@
+ // a recursive function:
 R F0(p) { ...  F0(...)
     ...  }
```

```
--- cex_func_recursive_1.c
+++ cex_func_recursive_1.
    patched.c
@@ -1,6 +1,8 @@
+// a recursive function:
 int f(int i) { f(i-1); }
 int h(int i);
 int g(int i) { h(i-1); }
 int h(int i) { return g(i
     -1); }
+// a recursive function:
 int l(int i) { return l(i
     -1); }
 int main() { f(1); g(1); h
     (1); }
```

example name: cex_func_recursive_1

Spot tricky interactions

Identifying mutually recursive functions

```
@ar@
identifier F0;
type R;
@@
 R F0(...) { ...  }

@rf@
identifier ar.F0;
type ar.R;
@@
 R F0(...) { ...  F0(...)  ...  }

@nr depends on !rf@
identifier F1;
identifier ar.F0;
type ar.R;
@@
 R F0(...) { ...  F1(...)  ...  }

@@
identifier ar.F0,nr.F1;
type S;
@@
+ // mutual recursion detected:
 S F1(...) { ...  F0(...)  ...  }
```

```
--- cex_func_recursive_4.c
+++ cex_func_recursive_4.patched.c
@@ -1,6 +1,9 @@
 int f(int i) { f(i-1); }
+// mutual recursion detected:
 int h(int i);
+// mutual recursion detected:
 int g(int i) { h(i-1); }
+// mutual recursion detected:
 int h(int i) { return g(i-1); }
 int l(int i) { return l(i-1); }
 int main() { f(1); g(1); h(1); }
```

example name: cex_func_recursive_4

Spot trickier interactions

Array of Arrays of Arrays ⇒ Array

```
@@ @@
 double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X)*(M*N)+(Y)*(N)+(
      M))

@@ @@
-a3 = calloc (...);
+a1 = calloc (L*M*N,sizeof(*a1));

@@
expression E1,E2,E3;
@@
-a3[E1][E2][E3]
+a1[A3D(E1,E2,E3)]
```

```
--- cex_arrays3Dto1D_1.c
+++ cex_arrays3Dto1D_1.patched.c
@@ -1,18 +1,20 @@
 #include <stdlib.h>
 double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X) * (M * N) + (Y)
      * (N) + (M))
 int main() {
   int i,j,k;
   const int L=2,M=3,N=4;

-  a3 = calloc(L,sizeof(*a3));
+  a1 = calloc(L * M * N, sizeof(*a1));
   for (i=0;i<L;++i)
   {
    a3[i]= calloc(M,sizeof(**a3));
    for (j=0;j<M;++j)
      a3[i][j]= calloc(N,sizeof(***a3));
   }
   for (i=0;i<L;++i)
    for (j=0;j<M;++j)
     for (k=0;k<N;++k)
-      a3[i][j][k]=i+j+k;
+      a1[A3D(i, j, k)]=i+j+k;
 }
```

example name: cex_arrays3Dto1D_1

How to restructure code full of indirect accesses ?

Thanks to Dr. Andre Kurzmann (LRZ) for suggesting this problem!

Array of Arrays of Arrays ⇒ Array (refinements)

```
@@ @@
-double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X)*(M*N)+(Y)*(N)+(M))

@@ @@
-a3 = calloc (...);
+a1 = calloc (L*M*N,sizeof(*a1));

@@
expression E1,E2,E3;
@@
-a3[E1][E2][E3]
+a1[A3D(E1,E2,E3)]

@@
statement S;
@@
(
- a3@S = calloc ( ... );
|
- a3[...]@S = calloc ( ... );
|
- a3[...][...]@S = calloc( ... );
)
```

```
--- cex_arrays3Dto1D_2.c
+++ cex_arrays3Dto1D_2.patched.c
@@ -1,18 +1,18 @@
 #include <stdlib.h>
-double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X) * (M * N) + (Y) * (N) + (M))
 int main() {
   int i,j,k;
   const int L=2,M=3,N=4;

-  a3 = calloc(L,sizeof(*a3));
+  a1 = calloc(L * M * N, sizeof(*a1));
   for (i=0;i<L;++i)
   {
-   a3[i]= calloc(M,sizeof(**a3));
    for (j=0;j<M;++j)
-     a3[i][j]= calloc(N,sizeof(***a3));
+     {}
   }
   for (i=0;i<L;++i)
    for (j=0;j<M;++j)
     for (k=0;k<N;++k)
-     a3[i][j][k]=i+j+k;
+     a1[A3D(i, j, k)]=i+j+k;
 }
```

example name: cex_arrays3Dto1D_2

Array of Arrays of Arrays ⇒ Array (refinements)

```
@@ @@
-double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X)*(M*N)+(Y)*(N)+(
     M))

@@ @@
-a3 = calloc (...);
+a1 = calloc (L*M*N,sizeof(*a1));

@@
expression E1,E2,E3;
@@
-a3[E1][E2][E3]
+a1[A3D(E1,E2,E3)]

@@
statement S;
@@
(
- a3@S = calloc ( ... );
|
- a3[...]@S = calloc ( ... );
|
- a3[...][...]@S = calloc( ... );
)

@@ @@
- for(...;...;...) { }
@@ @@
- for(...;...;...) { }
```

```
--- cex_arrays3Dto1D_3.c
+++ cex_arrays3Dto1D_3.patched.c
@@ -1,18 +1,13 @@
 #include <stdlib.h>
-double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X) * (M * N) + (Y)
      * (N) + (M))
 int main() {
   int i,j,k;
   const int L=2,M=3,N=4;

-  a3 = calloc(L,sizeof(*a3));
-  for (i=0;i<L;++i)
-  {
-   a3[i]= calloc(M,sizeof(**a3));
-   for (j=0;j<M;++j)
-     a3[i][j]= calloc(N,sizeof(***a3));
-  }
+  a1 = calloc(L * M * N, sizeof(*a1));
   for (i=0;i<L;++i)
    for (j=0;j<M;++j)
     for (k=0;k<N;++k)
-     a3[i][j][k]=i+j+k;
+     a1[A3D(i, j, k)]=i+j+k;
 }
```

example name: cex_arrays3Dto1D_3

Array of Arrays of Arrays ⇒ Array (refinements)

```
@@ @@
-double ***a3;
+double *a1;
+#define A3D(X,Y,Z) ((X)*(M*N)+(Y)*(N)+(
      M))

@@ @@
-a3 = calloc (...);
+a1 = calloc (L*M*N,sizeof(*a1));

@@
expression E1,E2,E3;
@@
-a3[E1][E2][E3]
+a1[A3D(E1,E2,E3)]

@@
statement S;
@@
(
- a3@S = calloc ( ... );
|
- a3[...]@S = calloc ( ... );
|
- a3[...][...]@S = calloc( ... );
)

@@ @@
- for(...;...;...) { }
@@ @@
- for(...;...;...) { }
@ identifier@ @@
-a1
+a3
```

```
--- cex_arrays3Dto1D_4.c
+++ cex_arrays3Dto1D_4.patched.c
@@ -1,18 +1,13 @@
 #include <stdlib.h>
-double ***a3;
+double *a3;
+#define A3D(X,Y,Z) ((X) * (M * N) + (Y)
      * (N) + (M))
 int main() {
   int i,j,k;
   const int L=2,M=3,N=4;

-   a3 = calloc(L,sizeof(*a3));
-   for (i=0;i<L;++i)
-   {
-    a3[i]= calloc(M,sizeof(**a3));
-    for (j=0;j<M;++j)
-      a3[i][j]= calloc(N,sizeof(***a3));
-   }
+   a3 = calloc(L * M * N, sizeof(*a3));
    for (i=0;i<L;++i)
     for (j=0;j<M;++j)
      for (k=0;k<N;++k)
-       a3[i][j][k]=i+j+k;
+       a3[A3D(i, j, k)]=i+j+k;
 }
```

`#pragma omp parallel` insertion

```
@sr@
identifier A={A};
statement S;
@@
\( S \& A \)

@fr@
identifier I;
statement sr.S;
position P;
@@
for( I=0; I<n; ++I) S@P

@ depends on fr@
statement sr.S;
position fr.P;
@@
+#pragma omp parallel
for( ...; ...; ...) S@P
```

```
--- cex_wishlist_insert_omp_1.c
+++ cex_wishlist_insert_omp_1.patched.c
@@ -1,10 +1,11 @@
 int main() {
  const n=10;
  double A[n];
  double B[3];
  int i;
+ #pragma omp parallel
  for(i=0;i<n;++i) A[i]++;
  for(i=0;i<3;++i) A[i]++;
  for(i=0;i<3;++i) B[i]++;
  for(i=0;i<3;++i) A[i]--;
 }
```

example name: cex_wishlist_insert_omp_1

Apply to *selected* loops

`#pragma` removal

No `#pragma` matching right now.

```
@@
@@
-#pragma GCC ivdep
```

```
int main() {
 const n=10;
 double A[n];
 int i;
#pragma GCC ivdep
 for(i=0;i<n;++i) A[i
    ]++;
}
```

```
int main() {
 const n=10;
 double A[n];
 int i;
#pragma GCC ivdep
 for(i=0;i<n;++i) A[i
    ]++;
}
```

example name: cex_wishlist_del_pragma1

`#pragma` removal

No `#pragma` matching right now.

```
@@
identifier I;
@@
-#pragma I
```

```
int main() {
 const n=10;
 double A[n];
 int i;
#pragma GCC
 for(i=0;i<n;++i) A[i
     ]++;
}
```

```
int main() {
 const n=10;
 double A[n];
 int i;
#pragma GCC
 for(i=0;i<n;++i) A[i
     ]++;
}
```

example name: cex_wishlist_del_pragma2

Scripting for custom comments insertion

```
@nr exists@
identifier CALLED;
identifier CALLER;
type R;
parameter list p;
@@
 R CALLER(p) { ... when any
   CALLED(...)
   ... when any
 }

@script:python pr@
CALLER << nr.CALLER;
CALLED << nr.CALLED;
K;
@@
coccinelle.K=cocci.make_ident("/* %s() invoked by %s() */" % (CALLED, CALLER));

@nri@
identifier pr.K;
identifier nr.CALLED;
type nr.R;
parameter list p;
@@
 R CALLED(p) {
++K;
 ...
}
```

```
--- cex_custom_comments_2.c
+++ cex_custom_comments_2.patched.c
@@ -1,8 +1,12 @@
-void f() { }
-void g() { f() ; }
-void h() { f() ; }
+void f() {
+       /* f() invoked by h() */;
+       /* f() invoked by g() */; }
+void g() {
+       /* g() invoked by i() */; f() ; }
+void h() {
+       /* h() invoked by i() */; f() ; }
 void i() { g() ; h() ; }
 int main() {
   f();
   g();
 }
```

Please note this is a dirty trick !

Call tree analysis

```
@initialize:python@
@@
KL=[]

@nr@
identifier CALLED;
identifier CALLER;
type R;
parameter list p;
@@
 R CALLER(p) { ... CALLED(...) ... }

@script:python@
CALLER << nr.CALLER;
CALLED << nr.CALLED;
@@
KL.append("%s -> %s" % (CALLER,CALLED));

@finalize:python@
@@
print "// " + str(len(KL)) + " relations
      :"
for kl in KL:
        print "//",kl
```

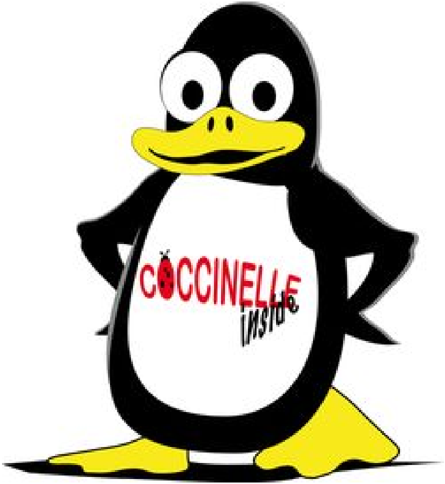
example name: cex_call_tree_1

Can arrange for other, specific analyses

Summing up

- powerful *open source* tool
- unique in its kind
- expressible almost as C itself
- **let's check it out for HPC codes restructuring!**

http://coccinelle.lip6.fr

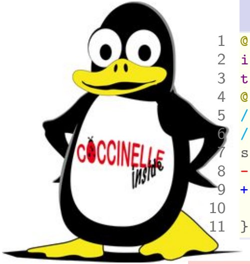


LRZ Training: Semantic Patching with Coccinelle

"...engine for specifying desired **matches** *and* **transformations** *in C code"*

API upgrade, bug hunt, HPC restructure (e.g. GPU port), analysis...

(original use: automatically keep Linux kernel driver code up-to-date)



Example: specialized data layout *conversion rules*

```
@struct_rule@
identifier id,I;
type T={float};
@@
//Match & modify
// float fields:
struct id { ...
- T  I;
+ T *I;
  ...
};
```

```
@exp_rule@
expression E;
identifier AoS,J;
fresh identifier SoA=AoS##"_soa";
@@
//Match & modify C expressions
//from array of structs to struct
//of arrays:
- AoS[E].J
+ SoA.J[E]
```

One-day training: 08.10.2019 at LRZ

Register online: https://www.lrz.de/services/compute/courses/2019-10-08_hspc1w19/