Knowledge Extraction from C-Code

Willibald Krenn and Franz Wotawa

Institute for Software Technology,
Graz University of Technology

Outline

1. Motivation: "*Why do we want to extract knowledge?*
   "*What type of knowledge do we search for?*"

2. Basic Idea and Conversion Process: "*How do we extract knowledge?*"

3. Discussion of Limitations, Outlook

4. Q/A

Motivation – Why extract knowledge?

- Conversion of existing control programs to knowledge-base based ones
  - Make knowledge explicit and easier to maintain
  - Preserve large parts of the original control program
  - Enable the system to reason about itself: Truly autonomous systems.
- Debugging Aid
  - Quickly gain overview

Motivation – What type of knowledge?

- Conditions under which a certain functions get called:

  cond_1 & cond_2 & …. & cond_n → func

- IOW: We extract rules that tell the system when some low-level function (“action”) can be called.
- The extracted rules should preserve the original program behavior as much as possible.
- The rule set should not be a 1:1 representation of the C-program.

Outline

1. Motivation: "*Why do we want to extract knowledge?*
   "*What type of knowledge do we search for?*"

2. Basic Idea and Conversion Process: "*How do we extract knowledge?*"

3. Discussion of Limitations, Outlook

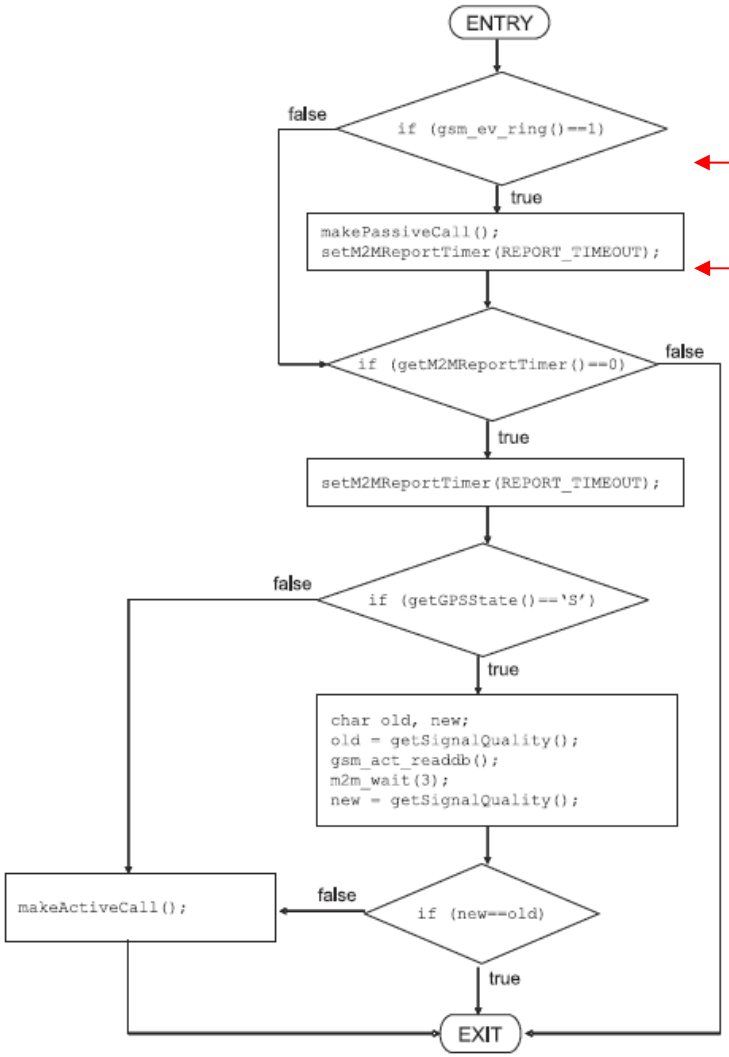
4. Q/A

Conversion Process – Control Program

```
1 void main ( void )
  {
3   /  . . . /
    while ( 1 ) {
5           runM2M( ) ;
    }
7 }
  / . . . /
9 void runM2M( void )
  {
11  if ( gsm_ev_ring ( ) == 1) {
          makePassiveCall ( );
13        setM2MReportTimer (REPORT_TIMEOUT ) ;
    }
15  if ( getM2MReportTimer ( ) == 0) {
          setM2MReportTimer (REPORT_TIMEOUT ) ;
17        if ( getGPSState ( ) == 'S' ) {
                    char old , new;
19                  old = getSignalQuality ( ) ;
                    gsm_act_readdb ( ) ;
21                  m2m_wait ( 3 );
                    new = getSignalQuality ( ) ;
23                  if ( new == old )
                              return;
25        }
          makeActiveCall ( );
27  }
   }
```

Extract knowledge about makePassiveCall and makeActiveCall

Conversion Process (2)



- Easy in the case of makePassiveCall:
  Cond1 → passiveCall()

- More difficult in case of makeActiveCall: Local Variables in condition are not allowed because their value is dependent on execution of some code.

- Solution: Encapsulate calculation and comparison inside a new function (“doCheck”).

Conversion Process (3)

```
int doCheck ( ) {
     char old , new ;
     old = getSignalQuality ( ) ;
     gsm_act_readdb ( ) ;
     m2m_wait ( 3 ) ;
     new = getSignalQuality ( ) ;
     if ( new == old )
            return 1 ;
     else
            return 0 ;
}
```

- Using this function, lines 17 to 25 can be re-written as follows:

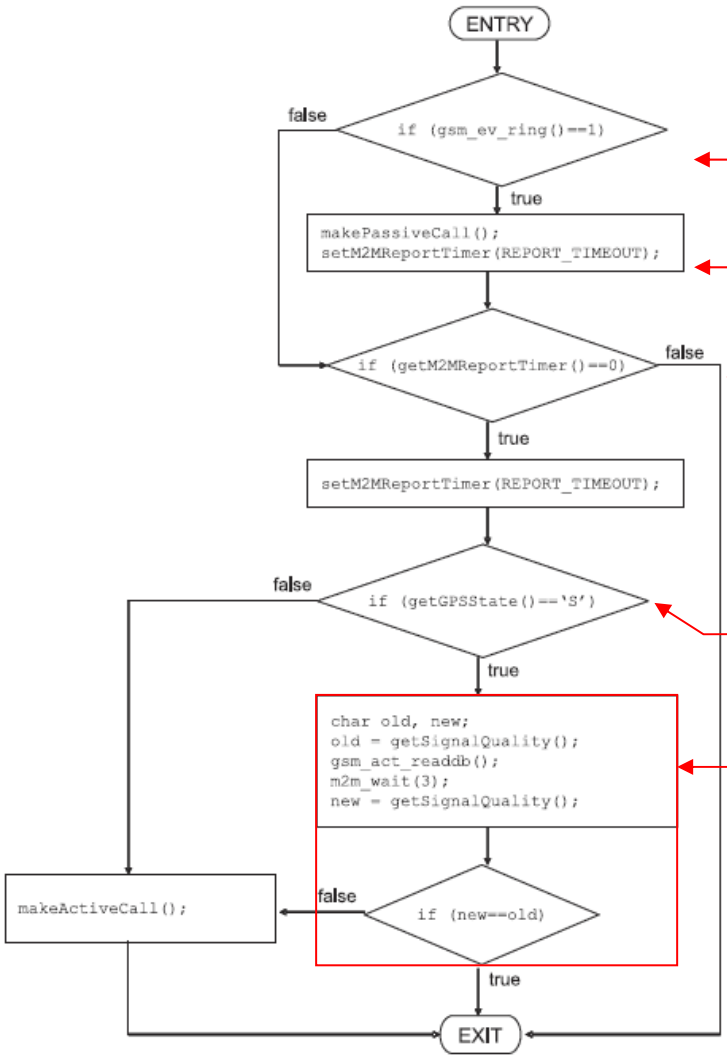
```
if ( getGPSState ( ) == 'S' )
     if ( doCheck ( ) == 1 )
            return ;
makeActiveCall ( ) ;
```

- Finally following rule can be extracted:

cond2 & (!cond3 | !cond4) → activeCall

cond2 … getM2MReportTimer() == 0
cond3 … getGPSState() == 'S'
cond4 … doCheck() == 1

Conversion Process (4)

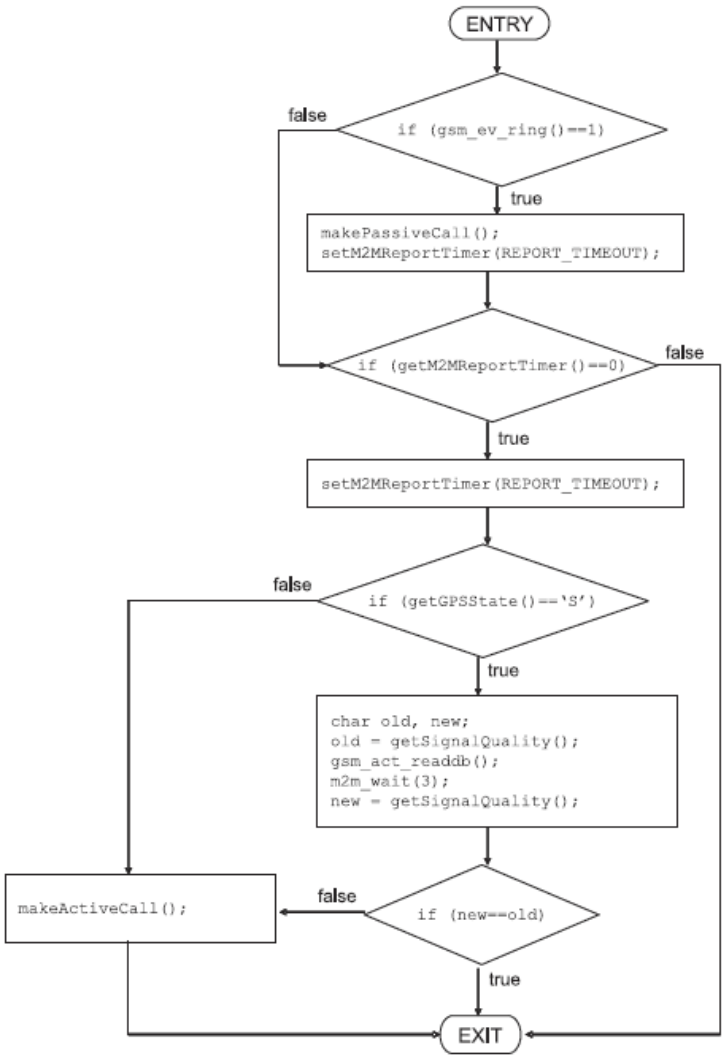


- In the case of makePassiveCall:
  Cond1 → passiveCall()
- In case of makeActiveCall:
  Cond2 & (!Cond3 | !Cond4) → activeCall

Conversion Process (5)

- After generating rules, the system minimizes them
- A search for common condition sequences also can be carried out
  - Re-introduce the notion of a state
- More detailed discussion of the algorithm contained in the paper.

Algorithm



Algorithm computeRules

Input: A program Π and a set of procedures or functions of interest F.
Output: A set of rules.

1. Let Π' be the program where all local variables used in conditional expression of Π have been eliminated by using behavior preserving transformations.
2. Construct a CFG for Π'.
3. Let R be the empty set. In R we are storing the extracted rules.
4. For all $f \in F$ do:
   (a) For all vertices v where f is called in the corresponding source code do:
      i. Extract the path(s) $(ENTRY, v_1, \ldots, v_k, v)$ from ENTRY to the vertex v.
      ii. Apply the transformation function l to the path(s) which is defined as follows:

$$l(x) = \begin{cases} \epsilon & \text{if } x = v \text{ or } x = ENTRY \text{ or } x \text{ is not a conditional} \\ x & \text{if } y \text{ is the immediate successor of } x \text{ in the path and the label of the arc } (x, y) \text{ is } \textit{true} \\ \neg x & \text{if } y \text{ is the immediate successor of } x \text{ in the path and the label of the arc } (x, y) \text{ is } \textit{false} \end{cases}$$

      Let $(l_1, \ldots, l_k)$ be the path after applying the function l.
      iii. Generate rule(s) $l_1 \wedge \ldots \wedge l_k \rightarrow f$ and add it to the set of rules R.
5. Minimize the set of rules R and return them as result.

Outline

1. Motivation: "*Why do we want to extract knowledge?*
   "*What type of knowledge do we search for?*"

2. Basic Idea and Conversion Process: "*How do we extract knowledge?*"

3. Discussion of Limitations, Outlook

4. Q/A

Limitations

- The algorithm may fail in several ways
  - Extract a wrong rule set due to e.g., hidden bugs in C-code
  - Extract rules that do not match the behavior of the C-code
- Other challenges
  - Interrupt routines (if candidates for knowledge extraction)
  - Worst case exponential
  - Lost states

Limitations - Example

```
if (i != j) {
2       if(a)
                c;
4       if (i==j)
                return;
6       if (!a)
                c;
8 }
```

- “c” is low-level (won’t be looked at)
- “c” does “i = j;”

- Extracted rules:
  - i != j & a → c
  - i != j & !a → c
- Simplifies to:
  - i != j → c;

This is NOT the intended behavior!

Outlook

- Working on a implementation in order to evaluate the usefulness.
- Compare the results with those of other approaches

Thank you for your attention