Automated Proof with Caduceus: Recent Industrial Experience

Dan Sheridan (djs@adelard.com)
About this talk

- This is a *fuzzy* talk, not a *maths* talk.
- I want to tell you about using an automated verification tool on a real piece of industrial code.
- I can't show you code or annotations because of the NDAs that make this work possible.
- Hopefully there will still be something interesting for the automated verification community!
Contents

- **Background**: the EAST and FEAST projects
- The FEAST4 manual proof
- **Applying** Caduceus
- **Experience** of automating the proof:
  - The good
  - The bad
- Some **conclusions**
EAST and FEAST

- **CINIF**, the Control and Instrumentation Nuclear Industry Forum funds applied research on behalf of the nuclear operating companies

- “Experience with Advanced Static Analysis” was funded by CINIF in 2001 to focus on analysing off-the-shelf computer systems in nuclear control applications

- It became “**Further Experience with Advanced Static Analysis**” in 2002. (After that we ran out of clever ideas for names)

- We looked at the effectiveness and cost of various techniques (from Lint up to Hoare logic-style proof) and their compatibility with safety cases
Smart sensors

- Little embedded computer systems: replacements for analogue level alarms and transmitters, with a bit more intelligence.
Why smart sensors?

- Lots of reasons for spending lots of effort on smart sensors:
  - Pure analogue sensors are **disappearing**
  - There is increasing **demand** for them from the stations
  - They are very **simple**, from an algorithmic point of view – they don't do very much, except linear arithmetic, table lookups, and I/O
  - They are typically written **without an OS**, removing much of the potential complexity
- The next step up, from a C&I point of view, is PLCs – fully programmable, real-time OS, ...
Industry situation

- The nuclear industry is a small customer, so doesn't have much leverage with the manufacturers. *If we want formality, we have to do it ourselves.*

- Surprising interest from one supplier: in FEAST we have worked with three devices (one assembler and two C) and the supplier has even taken on board our comments.
  - FEAST4, 5 and 6 focus on the latest of these devices
Contents

- **Background**: the EAST and FEAST projects
- **The FEAST4 manual proof**
- **Applying** Caduceus
- **Experience** of automating the proof:
  - The good
  - The bad
- **Some conclusions**
Rough block diagram

- handle inputs
- do some maths
- handle outputs

mainline code

interrupts

configuration menus

shared variables
The FEAST4 manual proof

- Out of 10,000 lines of code, we focused on 200 that do the main transformation of the process variable (the bulk of the code is concerned with the serial interface and the menu system).

- In FEAST5 we explored the problem of arguing that the rest of the code doesn't invalidate the proofs.
Why was it hard?

- Documentation shortcomings
  - At least we had some!
  - It didn't really agree with itself, and it wasn't written with verification in mind
- Constructing specifications
  - We had to make up a formal spec – based on the documentation, the code, the comments, and what we thought it was supposed to do
- Presenting the work
  - Enough information to be repeatable
- We estimate
  - 2 days to write the specs, 1 day of analysis, and 5 days of presentation
Contents

- **Background**: the EAST and FEAST projects
- The FEAST4 **manual proof**
- **Applying** Caduceus
- **Experience** of automating the proof:
  - The good
  - The bad
- Some **conclusions**
Mechanising the proofs

● **Why?**
  - Less (or differently) error-prone than manual proof
  - More scalable than manual proof
  - Repeatable, once annotations have been developed

● **Advantages of Caduceus/Why tools**
  - Ability to use SMT decision procedures
  - Diverse backends
  - Operate directly on C (unlike Malpas, the current tool of choice in the nuclear industry)
A process for using Caduceus

- Is this obvious? One of the nice things about Caduceus is that it enables a rapid **edit/prove/debug** cycle.
- For each function:
  - Run Caduceus on the source to find out what it's going to choke on:
    - Unions, strings, and unsafe pointer manipulation
  - Try a precondition and a postcondition for each case in the specification
    - Discover unexpected proof failures (more on this later) as well as incorrect specs
  - Put the specs back together to make a “one-click” proof
Language barriers

- Caduceus handles most of C, but leaves out some constructs (especially \texttt{union, strings, unsafe pointers}).
  - We want to avoid changing the code, though.
  - Must be careful with anything we do to it to push it through the tools – is it really still the same code?
- Interesting overlap with \textbf{Safer C} recommendations explored in FEAST4, though – we've already tried to avoid some of these issues.
Old habits die hard

#include <stdio.h>
#include <math.h>

const char *Ver = "1.1";

- We don't need complex headers, but programmers often throw them in out of habit
- The only use of strings in this signal processing software – ignore it
Memory accesses

```c
addr = (float*)(x+(int)y);
return *addr;
```

Lucky in this case – access is to a table in EEPROM. Essentially an array lookup because \( b[x+y] \) is equivalent to \( *(\&b+x+y) \)
Unions

union FloatToByte {
    float F;
    long L;
};

- Not the classic use of unions (to produce sum types); this is a type-hack. Used to check whether the EEPROM has been initialised – we want to read floats, but check whether the bit pattern is all 1s.
  - Fixed by replacing the union with a float only, and dropping the initialisation check. Unsafe?
- There are other union datatypes in the header files – but the types aren't used in the interesting code, so we commented them out.
Contents

- **Background**: the EAST and FEAST projects
- The FEAST4 **manual proof**
- **Applying** Caduceus
- **Experience** of automating the proof:
  - The good
  - The bad
- Some **conclusions**
Experience of applying Caduceus

- **Success** in that we found a mismatch between the spec and the code, missed in the manual proof – neatly illustrates the advantage of the approach
  - The mistake wasn't a big deal, though – it's to do with the value of the process variable in the divide-by-zero case, which is later discarded.

- **Cost-effective** – the distribution of time was slightly different from manual proof, but more scalable:
  - Creating spec: 2 days
  - Preparing for Caduceus: 3 days
  - Executing Caudceus: 3 days
The divide-by-zero problem

```c
float example(float w, x, y) {
    if (x-y == 0)
        /* Raise an error */
    else
        return w/(x-y);
}
```

- Clearly, no divide-by-zero can occur (although there is danger of overflow...)
- Caduceus adds precondition \( x - y \neq 0.0 \)
- but the conditional provides \( x - y \neq \text{RealOfInt}(0) \)
- and our specification initially isolated the cases with \( x \neq y \)
Divide-by-zero and solvers

- What do solvers think of $x - y \neq 0.0$, $x - y \neq RealOfInt(0)$ and $x \neq y$?
- CVC3 (in SMTLIB mode) cannot relate any of these
- Yices (SMTLIB) barfs on $w/ (x-y)$
- CVC3 (in CVC-lite mode) treats integers as a subset of the reals, so has no trouble
Caduceus: the good

- Operates on **nearly unmodified C** – it's easy to argue that the proofs carry across to the real code
- **Easy to use** – the GUI is brilliant, and allows us easy access to lots of solvers
- **Solver diversity** – brings both confidence (when the solvers agree) and strength (when some solvers succeed and some don't)
Caduceus: the bad

- The GUI could be more supportive: it's possible to figure out which path through the code is being considered by the structure of the proof; the GUI could support this to help find spec problems.
- The frontend should ignore stuff that's not used – e.g., rather than die on unions, only die when they are instantiated.
- It would be nice to be able to specify multiple pre/post-condition pairs, rather than having to do the case split manually.
- The real output of the solvers is hidden in the terminal window and the debug option. More parsing?
Contents

- **Background**: the EAST and FEAST projects
- The FEAST4 **manual proof**
- **Applying** Caduceus
- **Experience** of automating the proof:
  - The good
  - The bad
- **Some conclusions**
Conclusions: my views

- The multiple-solver situation is reminiscent of BlackBox (planning-as-SAT system). Will a *zchaff* of the SMT world emerge and make it all pointless?
- In the course of the FEAST projects, we've also looked at partial evaluation and flow graph generation (using Sparse); an all-in-one tool would be nice. *Frama-C*?
- I hate not being able to provide more detail on this work – hopefully we'll be able to eventually release something anonymised or abstracted.