Industrial Strength Exception Freedom

Rod Chapman

Praxis Critical Systems Limited

Introduction

- AdaUK 1993 – “Automatic Proof of the absence of run-time errors” paper.
- Since then…
  - SPARK language has grown…
  - SPARK Tools have improved…
  - CPU power has increased…
  - Industrial projects…
- What happened next?

Problem

- Run-time errors (or "exceptions" in Ada terminology) are a common source of error in programs.
  - The ubiquitous "buffer overflow" for instance.
- They are intolerable in many high-integrity systems.
- They often elude detection by testing…so what can we do?

Dealing with exceptions

- Dynamic handling…
  - Can become complex. Very difficult to argue correctness of handlers. Difficult in hard real-time.
- Elimination by dynamic analysis – e.g. testing.
  - Only shows presence, not absence. Exhaustive test almost always impossible. Not good enough for highest-integrity.
- Elimination by static analysis – Aha!

Static elimination of exceptions

- Static analysis can be valid for *all* input data.
- Can be used *early* (i.e. before test), so defects are prevented at source. Cheaper!
- Core issue: utility and efficiency of such analysis depends critically on the language under analysis.
- Two main approaches have been demonstrated:
  – Program Proof
  – Abstract Interpretation

Static analysis prerequisites

- *Ambiguity* is the enemy of static analysis.
- Unfortunately, every standard, unsubsetted programming language has ambiguities.
- Ideally, the language under analysis should be as unambiguous as possible.
- Analysis must not make assumptions that cannot be checked!

The SPARK Approach

- Aims for unambiguous semantics
- Therefore, no assumptions made in analysis – results are valid for all compilers on all targets.
- All prerequisite language rules are machine-checkable:
  - e.g. Data-flow analysis to prevent erroneous execution.
- A program-proof based approach to proving freedom from exceptions.
- Result: analysis which is *both* very deep and very efficient. Low “false alarm” rate.

Example

```
type T is range -128 .. 128;
procedure Inc (X : in out T)
--# derives X from X;
is
begin
   X := X + 1;
end Inc;
```

Example (2)

- The assignment statement has a check associated with it. The Examiner generates a Verification Condition (VC) for that check.
- The Simplifier tool attempts to prove the VC, leaving:

```
H1: x >= -128
H2: x <= 128
->
C1: x <= 127
```

- This VC cannot be proven, revealing the possibility of an exception.

Practical Issues (1)

- Input Data.
  - Anything coming from “the outside world” must be rigorously validated.
  - Be very careful where a type allows an object to have an invalid representation.
  - SPARK Examiner is very clever here, and models the ‘Valid attribute correctly.

Practical Issues (2)

- "Cosmic Rays" (or "SEUs")
  - Program Proof model does **not** deal with hardware failure and/or "random" failure of memory devices.
  - Programming language exception handling mechanisms are **not** the place to deal with this problem anyway.
  - Use fault-tolerant systems, error-detecting data representations, BIT etc. etc. These problems are well-known to the space community, for example.

Practical Issues (3)

- Storage_Error
  - What about running out of memory?
  - Good question!
  - SPARK can be compiled with no (implicit or explicit) use of a heap data-structure, so no problem there.
  - SPARK is non-recursive, and all constraints are static.
  - Problem reduces to a simple analysis of worst-case stack usage. Actually quite easy.

Dealing with unsimplified VCs

- The Simplifier is not an oracle – there are always a few VCs that it cannot prove.
- Unproven VCs might be OK (no exception, but Simplifier cannot prove it) or might indicate a true potential exception.
- SO…review them, or prove them formally with the Checker - an interactive theorem prover.
- You **learn** a lot by doing this!

Performance

- Times have changed…
- Theorem proving tactics have improved (so higher “hit rate” from Simplifier)
- Moore’s law marches on – significant theorem proving can now be attempted on modest PC hardware for industrial scale applications.

Performance data

Examiner 6.1, Simplifier 2.07, running on 1.3GHz Athlon, Windows 2000. All runtime-check VCs generated (including Overflow_Check).

Test Set	Examiner	SHOLIS	Project R
Executable loc	56760	16388	22968
Analysis & VCG time	4 mins 58 secs	4 mins 34 secs	2 mins 2 secs
Simp. time	5 hours 19 mins	8 hours 14 mins	1 hours 48 mins
Total RTC VCs	20833	6741	10963
RTC VCs proven by Simplifier 2.07	19127	6088	10017
Hit rate	91.8%	90.3%	91.4%

Conclusions

- It really works…
- We have several customers and projects using this technology right now.
- But…utility and efficiency critically depend on language under analysis…(don't try this on C!)
- The effort is worth-while:
  - Defects are discovered sooner rather than later.
  - You learn a lot about your program.
  - Testing (very expensive and boring!) is eased.
- A dramatic net **saving** in cost can result, especially at the highest integrity levels.