A Retrospective on Constructive Verification

Rod Chapman

Praxis High Integrity Systems

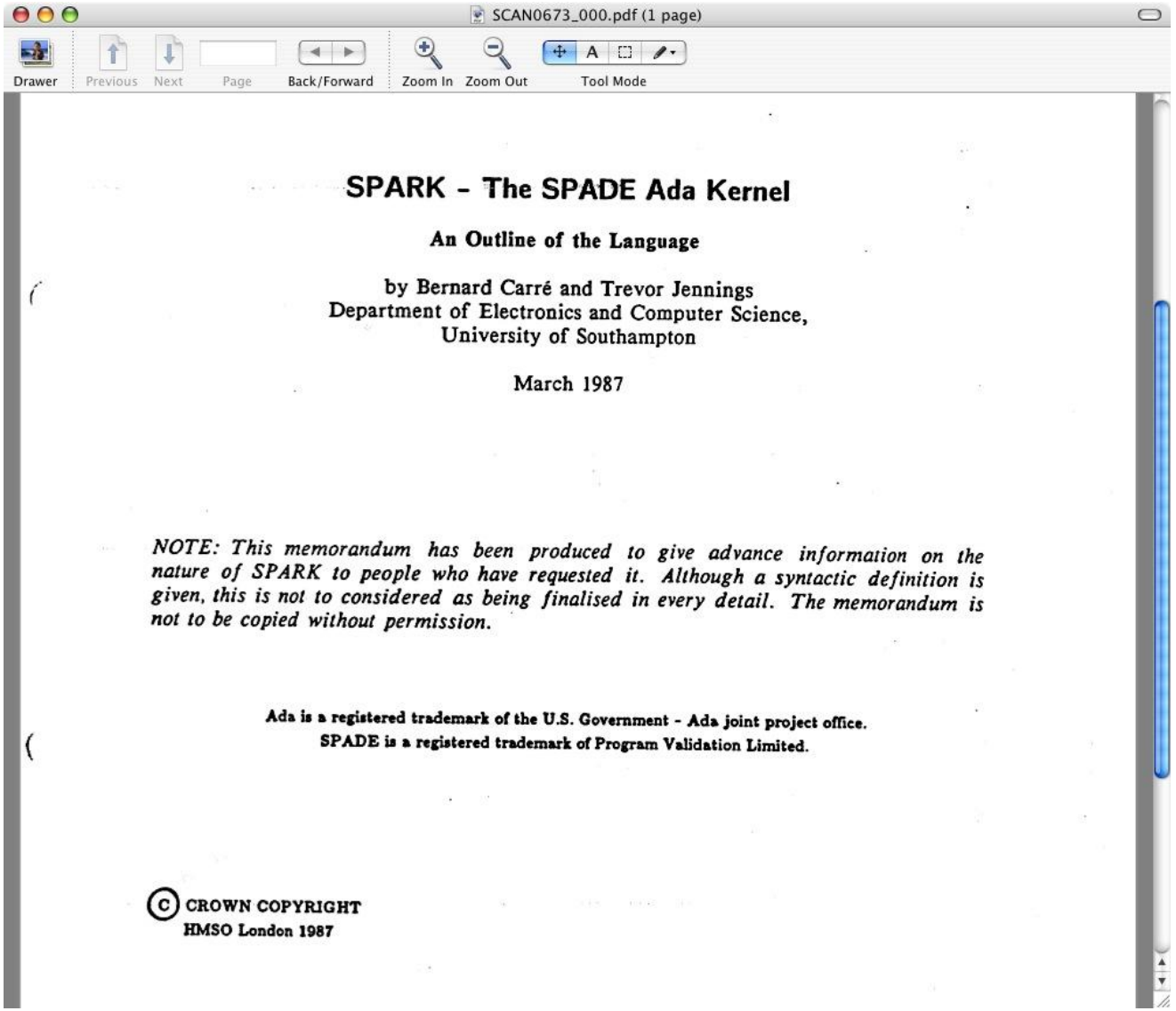
Contents

- Well…it was was 21(ish) years ago today…
- Retrospective vs Constructive verification
- Getting to 4th base…
- A future?

Contents

- Well…it was was 21(ish) years ago today…
- Retrospective vs Constructive verification
- Getting to 4th base…
- A future?

Well it was 21(ish) years ago today



Well it was 21(ish) years ago today

SPARK – The SPADE Ada Kernel

An Outline of the Language

by Bernard Carré and Trevor Jennings
Department of Electronics and Computer Science,
University of Southampton

March 1987

So what…

- PVL/Praxis/SPARK team have been
  - Designing programming languages…
  - Building static analysis tools…
  - Actually trying to use them on real projects…
  - …for what seems like a long time.
- Here are a few reflections on what we've learnt and what's going on now…

Contents

- Well…it was was 21(ish) years ago today…
- Retrospective vs Constructive verification
- Getting to 4th base…
- A future?

Early days…

- UK Military Aerospace
  - Software begins to appear in military aircraft in about 1985-1990
  - No tools exist…what can you do?
  - This led to *retrospective* style of analysis

Retrospective analysis

- Typical process
  - Procure/take delivery/pay for box containing software
  - Peer at it for a long time
  - Report bugs that you find. Hope they might be fixed.
  - Try to decide to fly aeroplane or not…
    - (In the mean time, some tools get developed – SPADE, MALPAS etc)

Retrospective analysis

- Analysis typically carried out by buyer/evaluator *after* development and "test".
- Observations:
  - Perception of limited utility: analysis is hard, slow, and subject to human frailty.
  - Little motivation for the developer to change their ways or do better

Retrospective analysis

- Key observation
  - Utility of retrospective analysis critically depends not only on quality/power of tools, but also on the quality of the software under analysis.
  - Poorly designed programs defy analysis by any method, tool or person.
    - Example: Chinook HC2 FADEC
  - Even programs which “seem to work” and “pass testing” defy analysis.

Retrospective analysis

- Is this still true?
  - Huge increase in tool power + 15 iterations of Moore's Law.
- But…
  - Massive increase in program size and complexity
  - Programming languages didn't help…they got bigger, more ambiguous, more dynamic…
- Who is winning this race?

Constructive Analysis

- The big idea:
  - Place tools in hands of developers, to be used all the time…
    - Use *discipline* to manage utility
  - Deliver system with static analysis evidence
  - Regulator and/or customer can reproduce evidence if they want.

Constructive Analysis

- Adoption is hard – requires major change of lifestyle for most developers.
- We encounter enormous resistance to the adoption of discipline.
  - Nobody likes being told what to do…

Contents

- Well…it was was 21(ish) years ago today…
- Retrospective vs Constructive verification
- Getting to 4th base…
- A future?

Getting to 4th base…

- A “playing field” for static analysis tools
  - 1st base: basic dumb mistakes – subset/coding standard etc.
  - 2nd base: absence of undefined behaviour (e.g uninitialized variables)
  - 3rd base: type safety
  - 4th base: partial corrrectness, safety and security properties, application and domain specific properties
  - 5th base+: stuff we haven’t even thought of yet…

Getting to 4th base…

- Note: at 4th base and above, desired properties are application and domain specific.
  - There is no “list of vulnerabilities” that can be enumerated or can be “built in” to a tool.
    - Overly generic description (e.g. “SQL Injection”) leads to hopeless false-positive rate from tools.
  - Many languages allow for user-defined properties, via assertions/contracts (e.g. SPARK, Eiffel) or via user-written “Rules” or “Checkers” (e.g. Coverity)

Getting to 4th base…

- SPARK gets to 4th base (just…)
- How?
  - Careful (some would say Draconian…) subset and contractualization of language.
  - Favour soundness above all other design goals.
  - Build soundness – base N+1 depends on base N analyses being OK first.

A worrying conversation

- Customer: “What list of bugs does your tools find?”
- Rod: “There’s no such list – it’s a general-purpose verification framework”
- Customer: “What list of bugs does your tools find?”
- Rod: “Anything that you can express as a predicate in first-order logic”
- Customer: “Eh?” (and leaves…)

A worrying conversation

- Where tools and languages support verification of user-defined properties:
- Perhaps we might ask:
  - "What properties can be expressed? What properties can't?"
  - "What is the soundness, completeness, and efficiency of the checking algorithm?"
- Many tool vendors don't seem to be very forthcoming with this information.

Does Soundness Matter?

- "Soundness doesn't matter"
  - Who says?
  - Well…err…All tool vendors whose tools are unsound.

- Or does it…?

Does Soundness Matter?

- In retrospective analysis mode, it appears not – finding 90% of bugs is better than none!

- But…if we are to move to constructive evidence-based assurance, soundness will matter
  - Would you present evidence to an evaluator if you *know* the tool that generated it can be unsound?
  - As an evaluator, would you accept such evidence?

Does Soundness Matter?

- A warning…
- Soundness is a one-way trip…
- Once achieved, customers will get used to it very rapidly, and come to depend on it.
  - You'll never go back…

Intermission…

- Enough moaning…
- Here's comes some code…

An example "4th Base" verification in SPARK.

- SQL Injection
  - Actually, just a special-case of input data validity.
  - It's both easy, and very hard…

SQL Injection

- Imagine a simple SPARK package that is used to query a database:

```
package DB
--# own State;
--# initializes State;
is
   procedure Query (SQL_String : in     String;
                    Result     :    out String);
   --# global in State;
   --# derives Result from State, SQL_String;
end DB;
```

SQL Injection

- Dumb implementation of user-generated query:

```
-- get input from user, whatever it is…
Read_Input (User_String);

-- construct SQL query string from user input
Form_Query (User_String, SQL_String);

-- Chuck the resulting query at the database
DB.Query (SQL_String, Result);
```

- This implementation is *weak* in that there is no checking that the user-provided string is not malicious, mal-formed, or just wrong.

SQL Injection

- A better SPARK Database Interface:

```
package DB
--# own State;
--# initializes State;
is
   function Valid_Query
      (SQL_String : in String) return Boolean;
   --# global in State;

   procedure Query (SQL_String : in     String;
                    Result     :    out String);
   --# global in State;
   --# derives Result from State, SQL_String;
   --# pre Valid_Query (SQL_String, State);
end DB;
```

SQL Injection

- Now what happens?

```
-- get input from user, whatever it is…
Read_Input (User_String);

-- construct SQL query string from user input
Form_Query (User_String, SQL_String);

-- Chuck the resulting query at the database
DB.Query (SQL_String, Result);
```

- You get an unprovable precondition VC for the call to DB.Query

SQL Injection

- The unprovable VC “reminds” you to bother to check, so I re-write the code:

```
-- get input from user, whatever it is…
Read_Input (User_String);

-- construct SQL query string from user input
Form_Query (User_String, SQL_String);

-- Check validity of generated query
if DB.Valid_Query (SQL_String) then
   -- Chuck the resulting query at the database
   DB.Query (SQL_String, Result);
else
   Error_Handler;
end if;
```

SQL Injection

- The offending precondition VC is now provable.
- Easy huh?
- Well…not quite…there's still no free lunch…

SQL Injection – The Catch…

- You have to write the bodies of DB.Valid_Query and Error_Handler
- What defines a “Valid” query anyway?
  - Look in your specification or security policy
  - You *have* got a specification, right?
- You end up *having* to specify error-handling behaviour as well…

Static analysis for engineer "behaviour modification"

- The upshot of all this:
  - A disciplined/formal/design-by-contract implementation style forces robustness.
  - This leads you to resolve issues in security policy, requirements, and specification.
  - The *behaviour* of engineers (eventually) changes to deal with these issue "up-front" rather than post-hoc.

Contents

- Well…it was was 21(ish) years ago today…
- Retrospective vs Constructive verification
- Getting to 4th base…
- A future?

A future?

- Who will win? Constructive or Retrospective tools?
  - Hopefully…*both*…
- All systems have many components – some new, some highly critical, some re-used, some COTS, some firmware, written in multiple languages.
  - There must be room for both styles of analysis.

A future?

- Why not use *architecture* to separate the really critical stuff form the rest?
  - Use sound constructive techniques where soundness and assurance really matter.
  - Use other techniques for the remainder.
    - (Assuming we can make logical arguments for separation and isolation of such components…)

Praxis High Integrity Systems

20 Manvers Street
Bath BA1 1PX
United Kingdom
Telephone: +44 (0) 1225 466991
Facsimilie: +44 (0) 1225 469006

Website: www.praxis-his.com, www.sparkada.com

Email: sparkinfo@praxis-his.com