Correctness by Construction: The Case for Constructive Static Verification

Rod Chapman

Praxis High Integrity Systems Limited

Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

So what is Correctness-by-Construction (CbyC)?

- Three central principles.
- *Prevent* defect introduction throughout the lifecycle.
- Detect and remove defects *as soon as possible* after their introduction.
- Say things only *once.*

CbyC Characteristics

- A development approach characterized by:
  - Use of static verification to prevent defects at all stages.
  - Small, verifiable design steps.
  - Appropriate use of formality.
  - "Right tools and notations for the job" approach.
  - Generation of certification/evaluation evidence as a side-effect of the development process. E.g. for a security evalution.

Contents

Testing…

- So why not just “test it to death…”?
- Program state space is *vast*. Testing only ever touches a *tiny fraction* of the paths and inputs.
- Statistics: to claim a reliability of N, how much testing to you need to do?
- Quiz: commercial aircraft aim for 1 failure in 10^9 flying hours. 10^9 hours is…?

- How much testing are you gonna do?!?
- Are you willing to stand up in court and say this?

Why Static Verification?

- Shows a program *will* work for *all* program paths, for *all* input data...good!
- Can be applied *early* - to specifications, designs, etc. as well as code.
- Generates assurance evidence as a by-product of the design process, *not* as an expensive, retrospective activity.

Catch 1

- Our ability to *statically* reason about programs, design, specifications etc. critically depends on the *language* in which these artefacts are written.
  - (Yes…languages do matter…)
- Questions such as
  - "What does my program mean?"
  - "Does my program have property X?"

should ideally have *only one* answer

Catch 1

- But most languages that we use are *ambiguous* – their meaning is not wholly defined. Oh dear!

- E.g. English
  - Time flies like an arrow (but, as everyone knows, fruit flies like a banana…)

- Ideally, we want notations that are as unambiguous as possible.

This is not a new idea...

“... one could communicate with these machines in any language provided it was an exact language ...”

“... the system should resemble normal mathematical procedure closely, but at the same time should be as unambiguous as possible.”

Who said this? When?

Languages - Definitions

- Programming languages have ambiguities (for good reasons) which are resolved by compiler writers.
  - Very few languages have ever been designed with verifiability as the primary design goal.
- A *dialect* P of a language S depends on particular choices made by a single compiler for a single target computer.
- A *pure subset* P of a language S is a sub-language of S where all P programs are legal in S and *have the same meaning*, regardless of compiler choice or target computer.

Languages ambiguities...
A sliding scale

- The ambiguous bits of programming languages differ in their ability to cause trouble.

- From better to worse...

- Implementation-defined: Compiler *is* obliged to document its behaviour and be consistent.
  - Examples:
    - range of “int” in C
    - Fiddly details of floating-point arithmetic

Languages ambiguities… A sliding scale

- Implementation-dependent or “unspecified”: behaviour is one of a small set, but unpredictable, and no obligation to document anything.
  - Examples:
    - Expressions evaluation order in C, C++, Ada
    - Parameter passing mechanism for composite types in Ada

Languages ambiguities…
A sliding scale

- Undefined or “Erroneous”: All bets are off! No guarantee of *anything at all*.
  - Worse: “program seems to work most of the time” *is* a common behaviour for undefined features. Yields a very bad false sense of security!
  - Examples:
    - Reading an uninitialized variable.
    - Unchecked buffer overflow

Contents

Goals for Constructive SV

- We want analyses which are:
  - Sound (absence of false negatives)
  - Complete (absence of false positives, aka “False alarms”)
  - Efficient – so it can be done in preference to compile/test
  - Modular – runs on incomplete programs and results are composable.
  - Deep – tells you something useful!

Goals for Constructive SV

- The five goals are in a subtle balance.
  - You can't have all of them all the time.
- Effectiveness critically depends on the language that you're analysing.
- No standard, unsubsetted language is suitable! There are just too many ambiguities and complications.

Contents

The irony of subsets, dialects, and analysis tools

- Most tools attempt analysis of the "whole language" to increase market share. They can be unsound, incomplete, too shallow, slow etc. etc…your mileage varies!
- BUT…*everyone* is using a subset or dialect!
  - Why?!?
  - Do you have a coding standard?

The irony of subsets, dialects, and analysis tools

- In reality, almost all projects end up unintentionally using a dialect.
  - Programmers "stray" into implementation-dependent areas of the language without even knowing it.
  - You end up "locked in" to your compiler and dialect
    - (compare with the "Software Crisis" of 1975…has anything changed?)

Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

Retrospective SV problems

- The effectiveness of retrospective analysis critically depends on how well the program is designed in the first place!
  - Adding retrospective SV doesn't improve this if it's too late to change the system.
- Put another way: You can't polish dirt!

Retrospective SV: Tool issues

- If a tool encounters an ambiguous language construct (e.g. evaluation order of an expression where the expression might have a side-effect), what can it do?
  - Assume left-to-right order? *Unsound* if compiler disagrees!
  - Assume right-to-left order? *Unsound* if compiler disagrees!
  - Analyse all orders? Horribly inefficient and tends to $O(2^N)$ time to analyse.

A failure of retrospective SV

- UK Military have been trying to use retrospective SV to evaluate and accept critical software since about the mid 1980s.
- Almost all the time, these efforts have been time-consuming, expensive, and produce dubious results.
- Sometimes, they just fail completely – here is one example:

Chinook HC2…

#include <tale of woe>;

Picture from www.raf.mod.uk

Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

Getting Constructive SV to work…

- The big idea: Start with (or design) an *unambiguous* language or a pure subset.
- Only 1 meaning to any program implies analysis can be deep, fast, sound and complete.
- Addition of “design by contract” annotations yields modular analysis.
- Goal: run the analysis *all the time* during development. There is NO separate “bug finding analysis” stage at all.

Turning the dials up...

- Types of static analysis

- The easy stuff:
  - Coding standards and “style” rules.
  - Simple subset checking (e.g. “no templates”)

Turning the dials up…

- Deeper…
  - Semantic analysis
    - Extended type checking
    - Absence of side-effects in expressions
    - Absence of implementation-defined and –dependent features.

Turning the dials up...

- Deeper still
  - Absence of undefined and erroneous behaviour
    - Data-flow analysis
    - Aliasing analysis
  - Information-flow analysis
    - Useful for MILS and other security properties

Turning the dials up...

- Really deep
  - Theorem proving or Abstract interpretation.
    - Absence of “run-time errors” such as buffer overflow, division by zero –Partial correctness verification
  - Safety and/or security property verification
  - Software model checking
  - Timing and memory-usage analysis

Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

Some example languages and tools

- A very brief and incomplete tour…
- <Insert your favourite here…>
- Apologies to any that I've missed.

Example languages and tools (1)

- MISRA C
  - A set of "guidelines" for the use of C developed by the automotive industry. Varied acceptance.
  - 127 rules.
  - Rules are informally defined, in "ISO English."
  - Rules basically imply: subset checking, static semantic checks, and data-flow analysis.
  - The good news:
    - Probably the best (public) guidelines for the use of ISO C ever produced.
    - Adoption by automotive industry has prompted much activity from the tool vendors to support it.
    - Now being revised to give a more formal definition of the rules.
    - Has influenced significant projects, such as JSF.

Example languages and tools (2)

- MISRA C - The bad news:
  - Informality of rules and inherent ambiguity of C90
    - "Compliance" is almost impossible to claim.
  - All tool vendors claim "100%" implementation of the rules.
    - All the tools are different!
    - Which is right?!?
  - C is very "pointer-centric" - meaning some of the rules are NP-hard or even undecideable to implement - oh dear...
  - Deep analysis is *slow*, which limits constructive use.
  - Tools suffer from high *false-alarm rate.*

Example languages and tools (3)

- ESC/Java2 and JML
  - The extended static checker for Java.
  - University research – no commercial support (yet…)
  - Data-flow analysis, theorem-proving etc. for runtime errors.
  - Great user interface – “hides the maths…”
  - Uses Java Modelling Language (JML) for design-by-contract.

Example languages and tools (4)

- Microsoft Static Driver Verifier
  - A retrospective analyser for device-driver code.
  - Assumes a small *dialect* of Microsoft C.
  - Checks code against the “how to write a device driver” rules.
  - Very advanced analysis – a hybrid of theorem proving and model checking.
  - Can be unsound and incomplete, but has still proven to be very very useful!
  - Productised now (flashy GUI etc.) and shipping on next MS DDK for users.

Example languages and tools (5)

- SPARK
  - Disclaimer: I am one of the designers!
  - Annotated (design-by-contract) *pure subset* of Ada95.
  - Designed from scratch for hard real-time and embedded, high integrity systems.
  - Tools do NOT attempt analysis of “Full Ada”, so the “whole language” problem does not appear.
  - It *does* deliver analysis which is sound, very nearly complete, deep, fast, and modular.

Example languages and tools (6)

- SPARK Analyses
  - Mandatory: Subset checking, static semantics, data-flow analysis.
  - Optional (stage 1): Information flow analysis
  - Optional (stage 2): Theorem proving for absence of runtime errors, partial correctness, safety and/or security properties.

Example languages and tools (7)

- SPARK Good news
  - Track record: industrial use since 1990. Has met or exceeded DO-178B level A, UK Def Stan 00-55 SIL4, ITSEC E6, Common Criteria EAL5+, CENELEC 50128 SIL4 etc. etc.
- Not so good news
  - It requires *discipline*!
  - It is unsuitable for retrospective analysis.
  - It's British ("Why can't we buy an American one?")
  - It's Ada…
- "Unfashionable but works!"

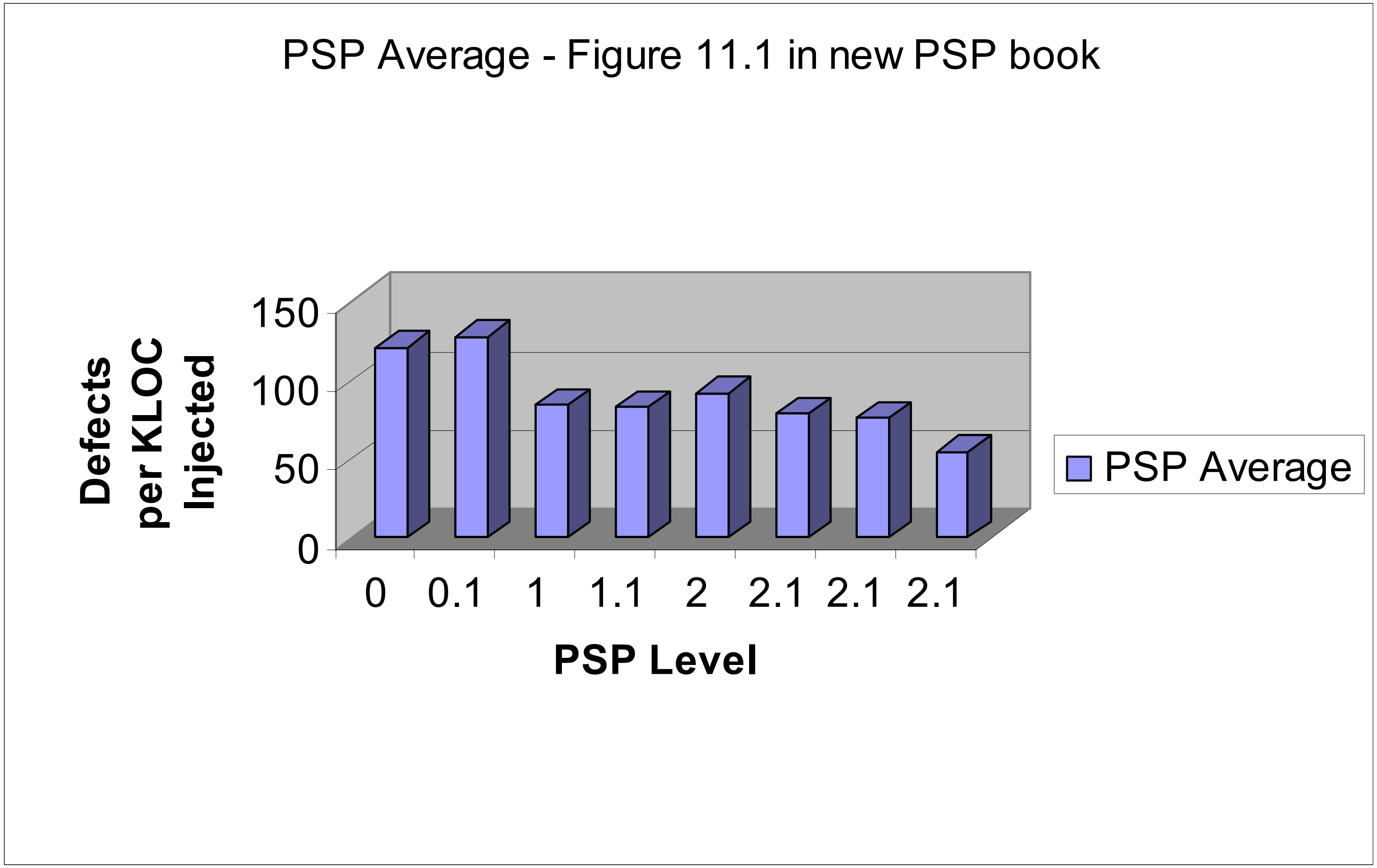
Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

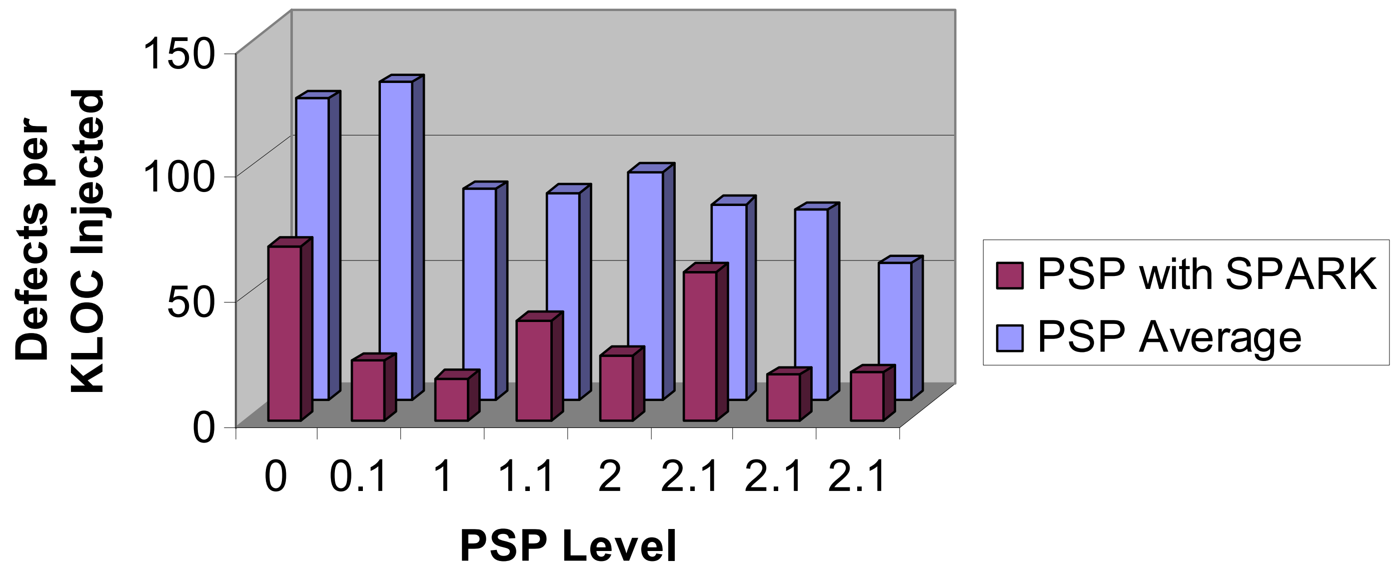
Results with Constructive SV

- Personal
  - *One* engineer has taken the SEI Personal Software Process (PSP^{SM}) course.
  - He used SPARK and constructive SV to do the PSP programming exercises.
  - Results:

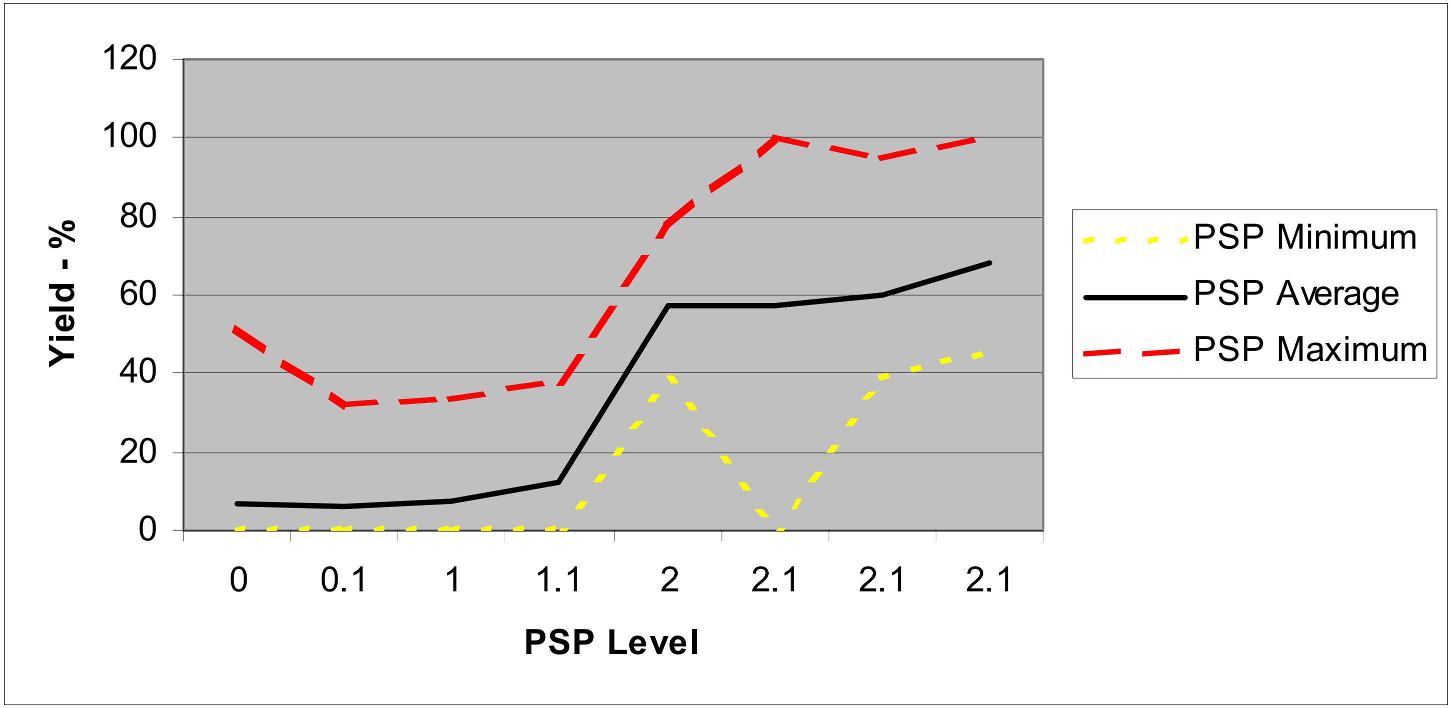
Defects injected per kloc



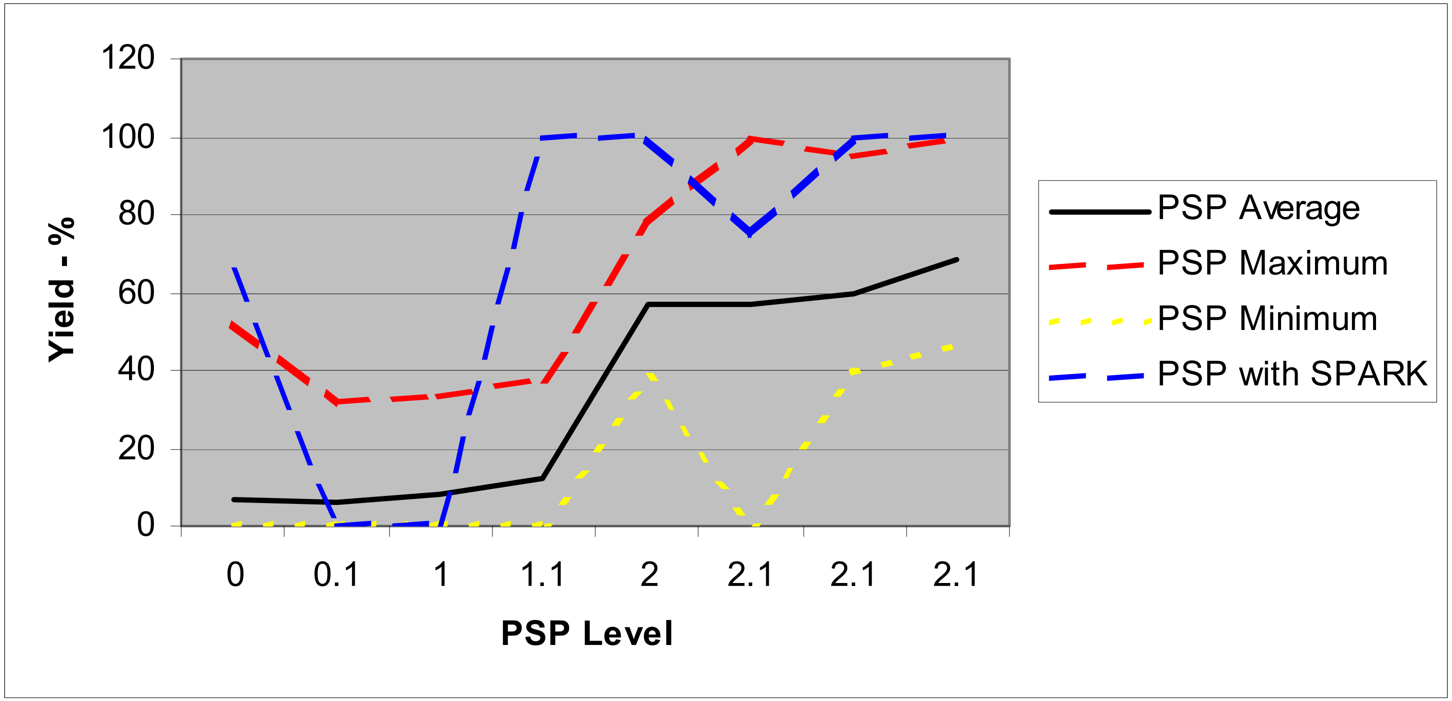
Defects injected per kloc



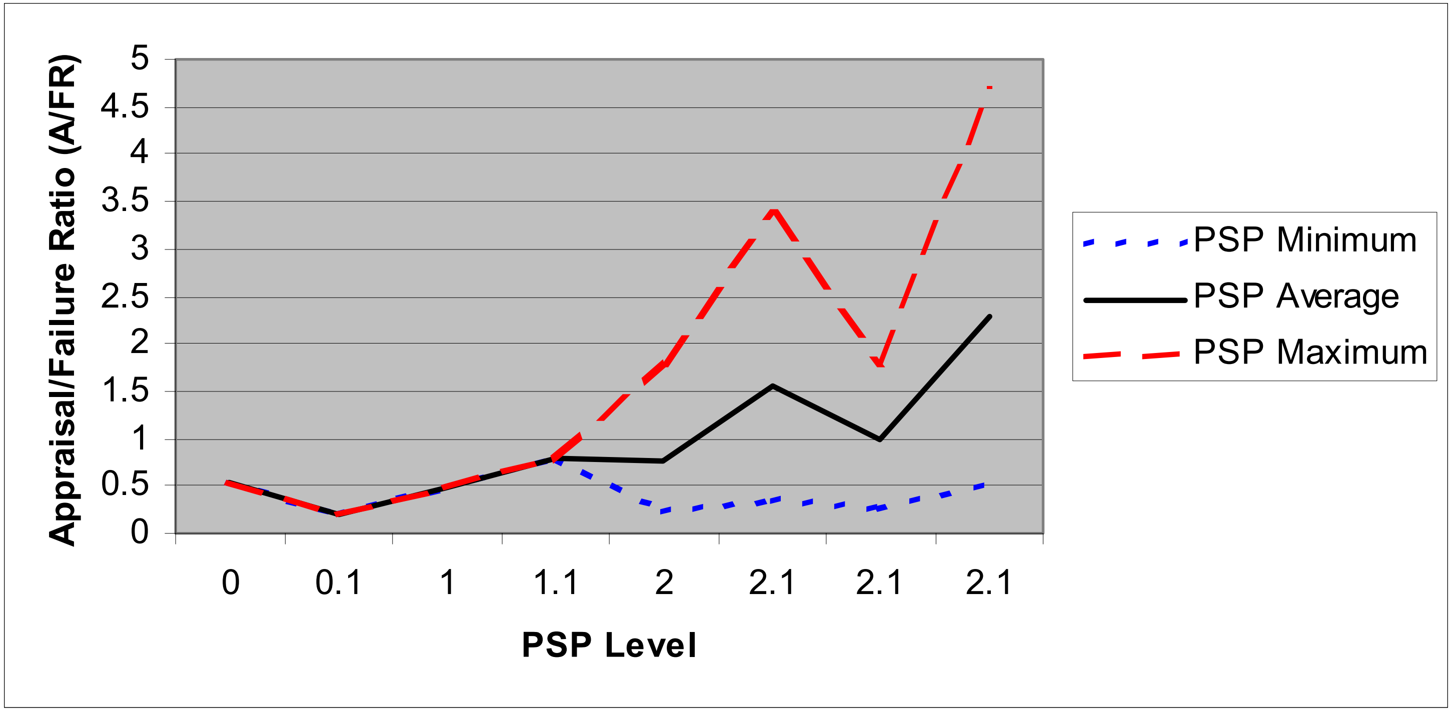
Process yield for 8 programs



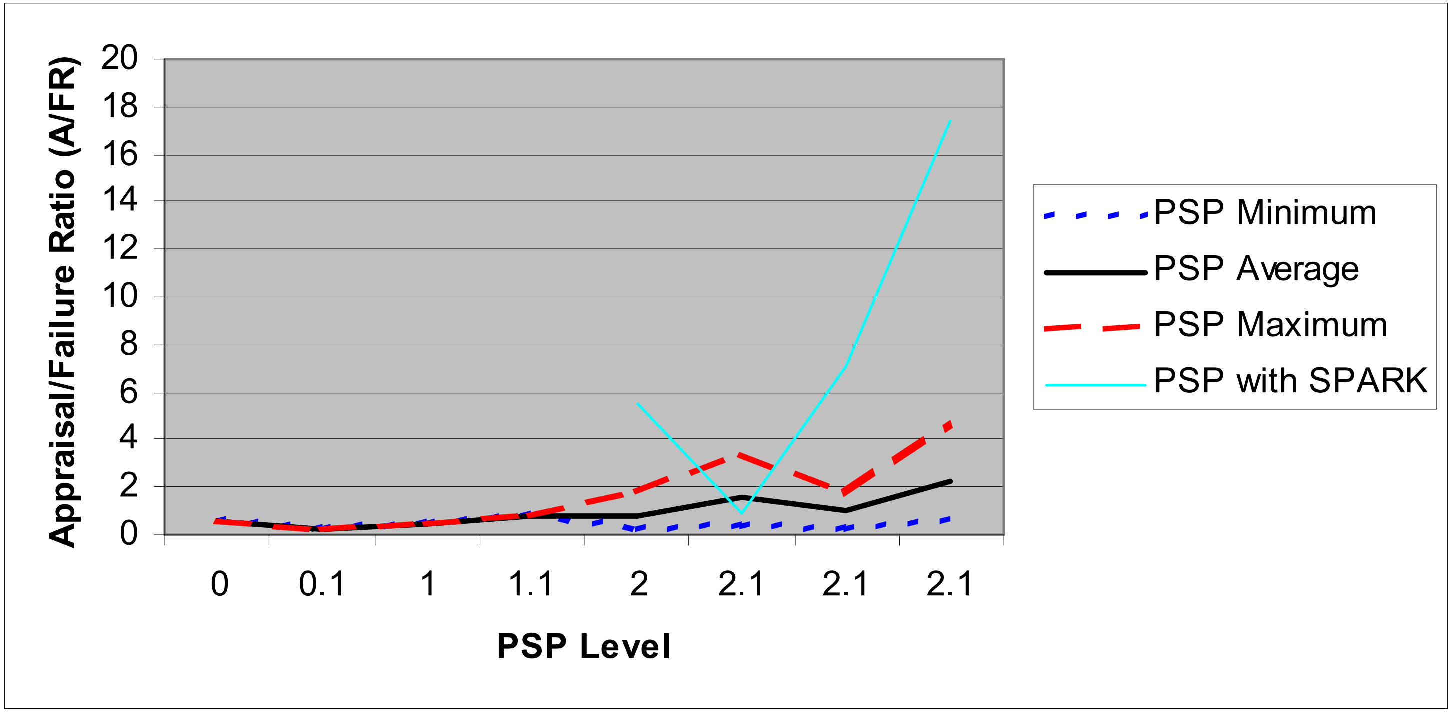
Process yield for 8 programs



A/FR Ratio for 8 programs



A/FR Ratio for 8 programs



Results with CbyC – Team and Projects

- Here are data for 5 projects using constructive static verification.
- Three are safety-critical.
- Two are security-critical.
- All used Correctness by Construction process.
- All except CDIS used strong, constructive static verification.

Results with CbyC – Team and Projects

- CDIS - Critical ATC System (London Airport)
- SHOLIS - Naval Ship/Helicopter Information System. First ever Def Stan 00-56 "SIL4" project.
- MULTOS CA - ITSEC E6 (=CC EAL7) secure certification authority.
- A - Naval stores management system.
- Tokeneer - Biometric access control system. CC EAL5 and above demonstrator project funded by a government agency.

A note on “defects”

- A "Defect" is *any* error in a design artefact once placed under change control or delivered to a client, including documents, designs, manuals etc. as well as code.
  - Expected behaviour is defined by the (formal) system specification.
- CDIS, SHOLIS and MULTOA CA were delivered with a *Warranty.*
- During the warranty period, *we* fix Defects at no charge.
- For these projects, the quoted figures are all for the whole project *after delivery*.

Results with CbyC – Team and Projects

Project	Year	Size (loc)	Productivity (loc/day)	Defects (per kloc)
CDIS	1992	197000	12.77	0.75
SHOLIS	1997	27000	7.0	0.22
MULTOS CA	1999	100000	28.0	0.04
A	2001	39000	11.0	0.05
Tokeneer	2003	10000	38.0	0.0

Contents

- Correctness by Construction
- Testing, Languages, Ambiguity, Analysis…
- Goals for Constructive SV
- The Catch…
- Why retrospective analysis doesn't work…
- Turning the dials up
- SV Languages and tools
- Results with CbyC and SV
- The future?

Future

- A few things to come:
  - We have trained the SEI in SPARK…
  - Combined PSP/TSP/SPARK/CbyC trial project soon.
  - Make SPARK subset bigger – generics, interfaces, more OO support, Ada2005 etc.

Conclusion

- It's like dieting!
  - Many "quick fixes", but to make a big difference a real change in lifestyle is needed.
  - Constructive SV offers an "alternative lifestyle" which is effective (but perhaps not for everyone.)

Resources

- www.praxis-his.com
  - Company, papers etc.
- www.sparkada.com
  - SPARK Information
  - White papers and publications

Praxis High Integrity Systems Limited

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