From Tests to Spec (and Back)

A report on work that other people
have done
but
where a lot still remains to be done.

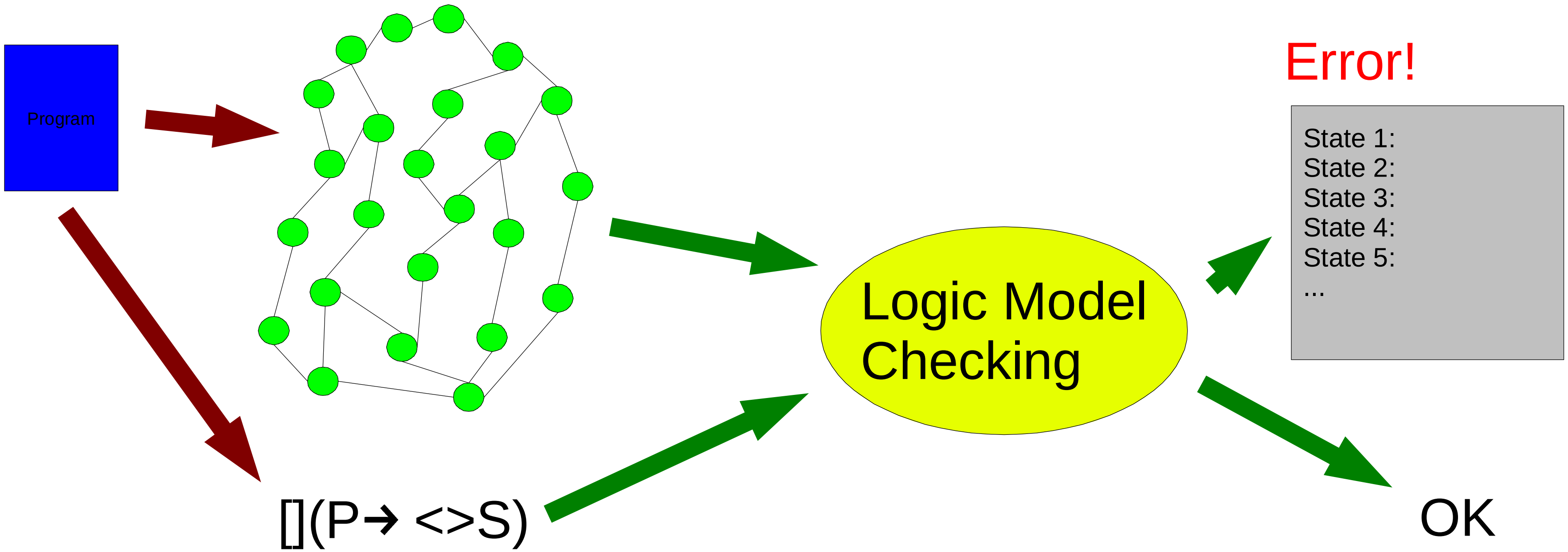
Juhan Ernits
Küberneetika Instituut

Ühel ilusal talvisel päeval Kokõl
(05.02.2005)

The Setting

- We assume, that we are interested in programs that work as intended.
- Observational estimate of the current state of matters:
  - There are lots of programs out there.
  - Most of them do not have formal specifications but some of them behave more or less as expected.
  - Some programs have test suites.
  - Fewer programs are attributed with annotations.
  - Most programs are actually checked whether they fulfil their initial intention by end users.

Verification



- Given a program and spec, build a model of the program and use logic model checking for checking the correspondence.
- The checker either returns “OK” or “Error” together with a trace to the error state

Some Issues in Verification

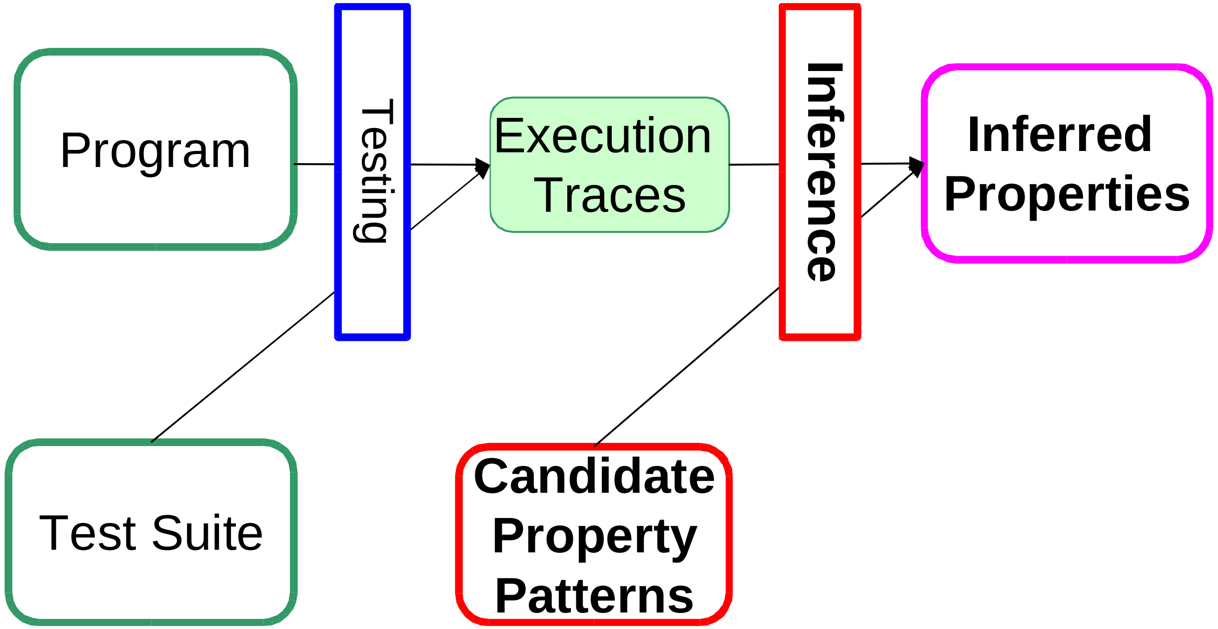
- Verification needs the desired properties to be defined in some formal way. But the properties are often
  - Unspecified;
  - Specified in prose (which is difficult to parse);
  - Specified ambiguously, e.g. “the program should work correctly”.
- There are other issues but they are not relevant here.

Test Suites

- More and more software builders pay attention to assembling test suites for automatically testing their products.
- The difference between testing and verification:
  - Testing explores some scenarios and specifies expected outcome;
  - Verification explores all scenarios regarding some specific property.

What if we say that the test suite is representative of the properties that we are interested in?

An approach by J. Yang and D. Evans



Temporal Properties

- Let us consider just one class of properties, e.g. temporal properties.
- Temporal properties are about the order of events in a system. E.g.
  - A file should be opened before it is read from.
  - When the subscriber picks up the phone, dial-tone is always generated.
- We assert that temporal properties are hard to write manually in formal ways, e.g. using temporal logic.

Temporal Logic (cont.)

- We assert that temporal properties are hard to write in formal ways, e.g. using temporal logic.

You disagree? Try writing down a formula for

"P triggers S between Q (e.g., end of system initialization) and R (start of system shutdown)"

Temporal Logic (cont.)

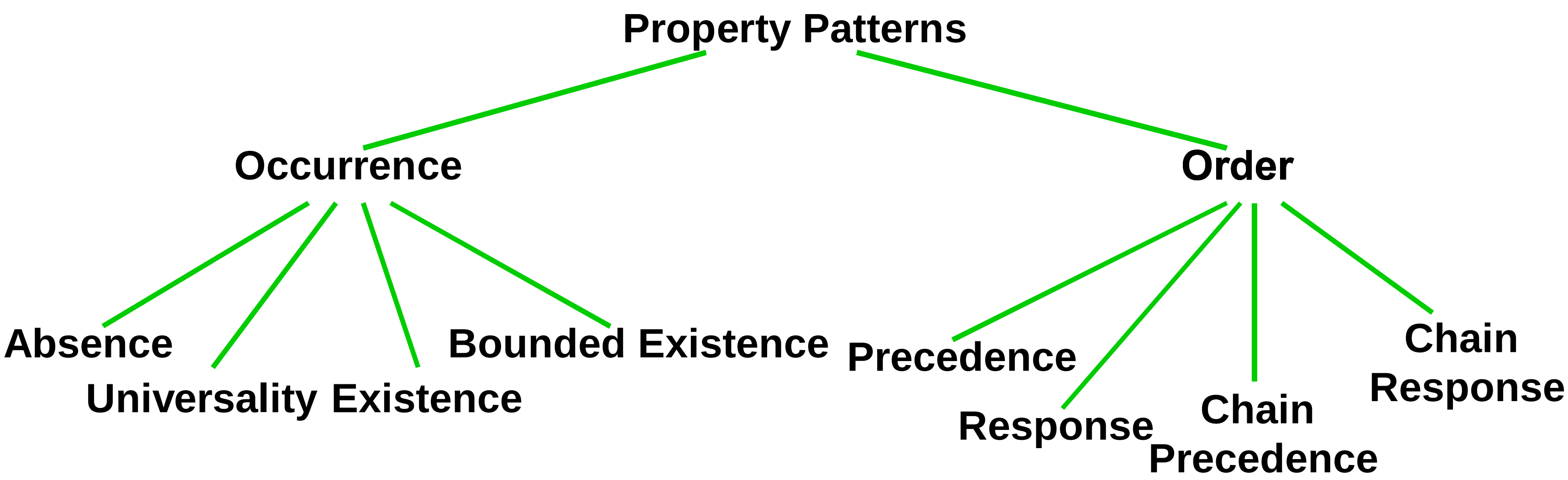
- We assert that temporal properties are hard to write in formal ways, e.g. using temporal logic.

You disagree? Try writing down a formula for

"P triggers S between Q (e.g., end of system initialization) and R (start of system shutdown)"

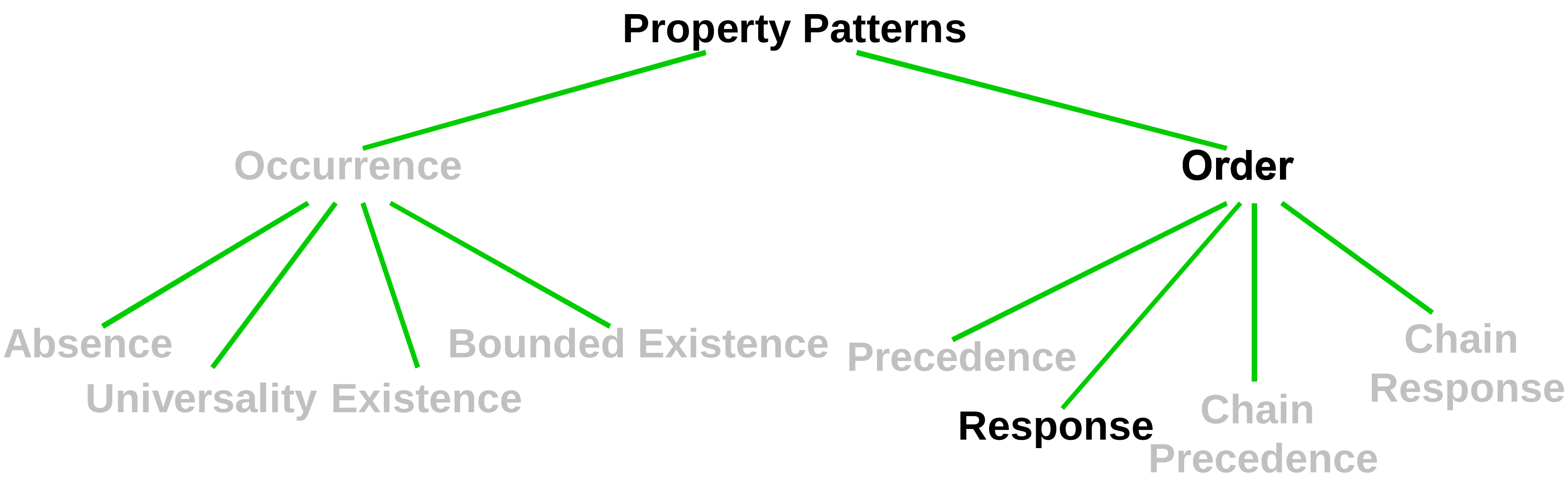
```
[]((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)
```

Property Patterns (Dwyer et al.)



This classification is a result of reading 500+ natural language specifications of real programs. The patterns are like templates (in temporal logic) where one can plug in specific P-s, Q-s, R-s and S-s, i.e. specific events.

Response Pattern



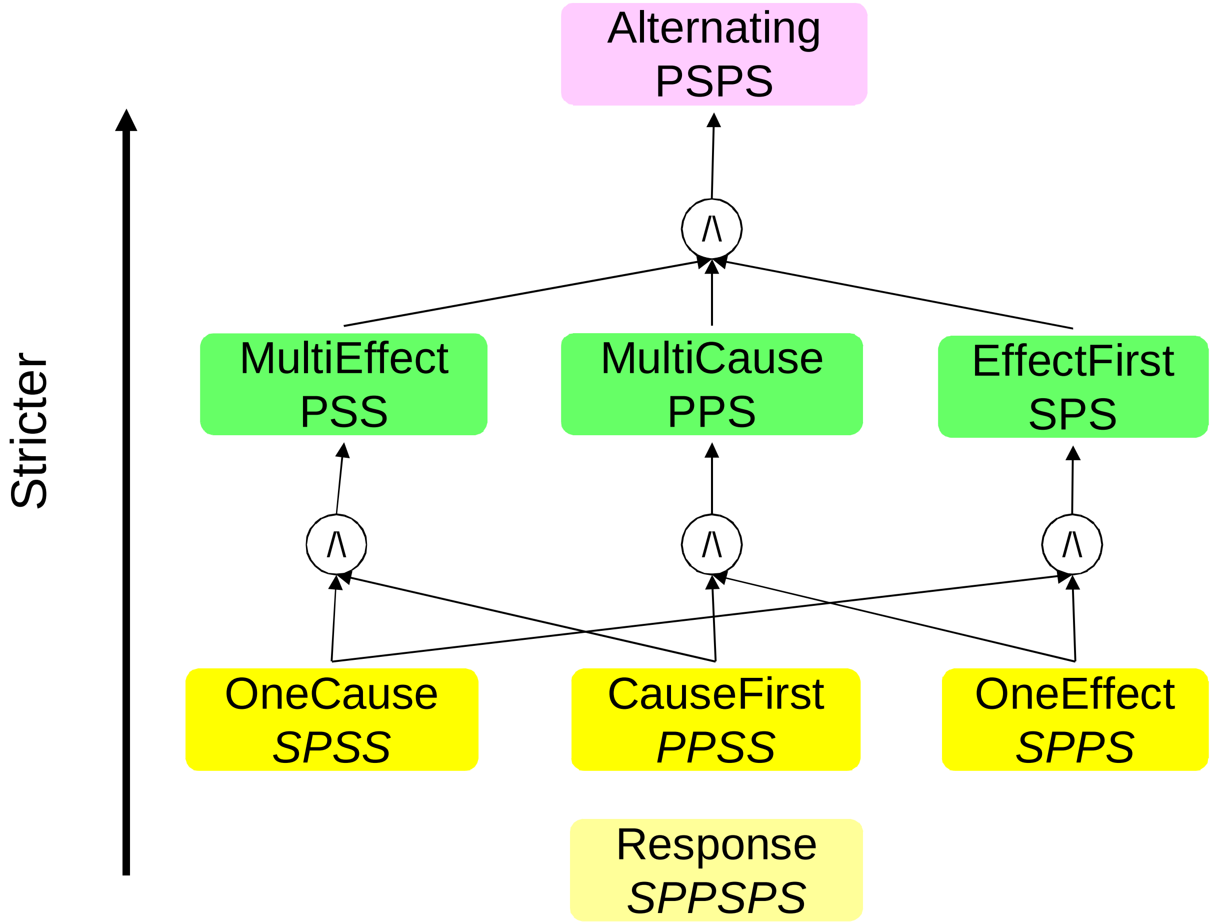
A state/event P must always be followed by a state/event Q within a scope

Or as a Quantified Regular Expression

$$[-P]^* \, (P \, [-S]^* \, S \, [-P]^*)^*$$

SPPSPS ✓ *SPSP* ✗

Refined Response Pattern (Yang et al.)



- For each combination of two events
- Decide if they satisfy *CauseFirst*, *OneCause*, or *OneEffect*
- Find the strictest pattern

Find the Strictest Pattern

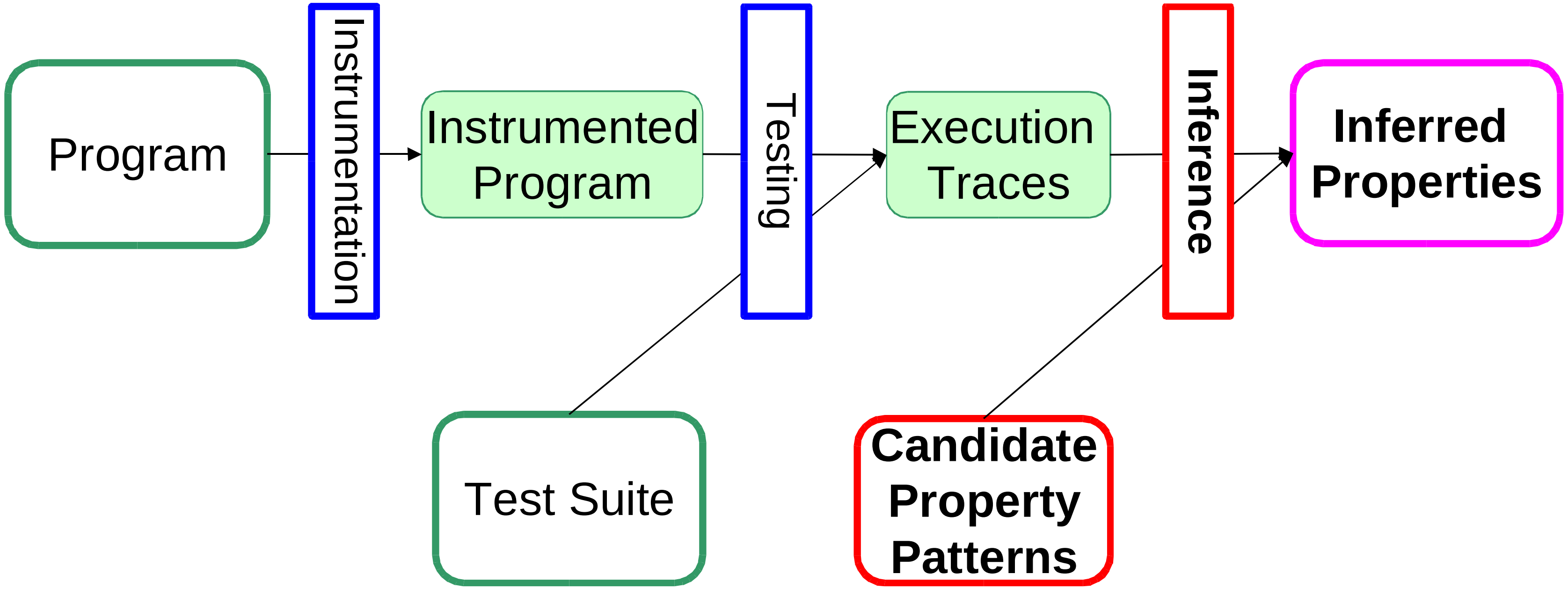
For any two events determine the strictest pattern:

	Trace 1: PSPS	Trace 2: PPS	All Traces
CauseFirst	+	+	+
OneCause	+	-	-
OneEffect	+	+	+

CauseFirst /\ OneEffect -> MultiCause

J. Yang et al.

Implementation

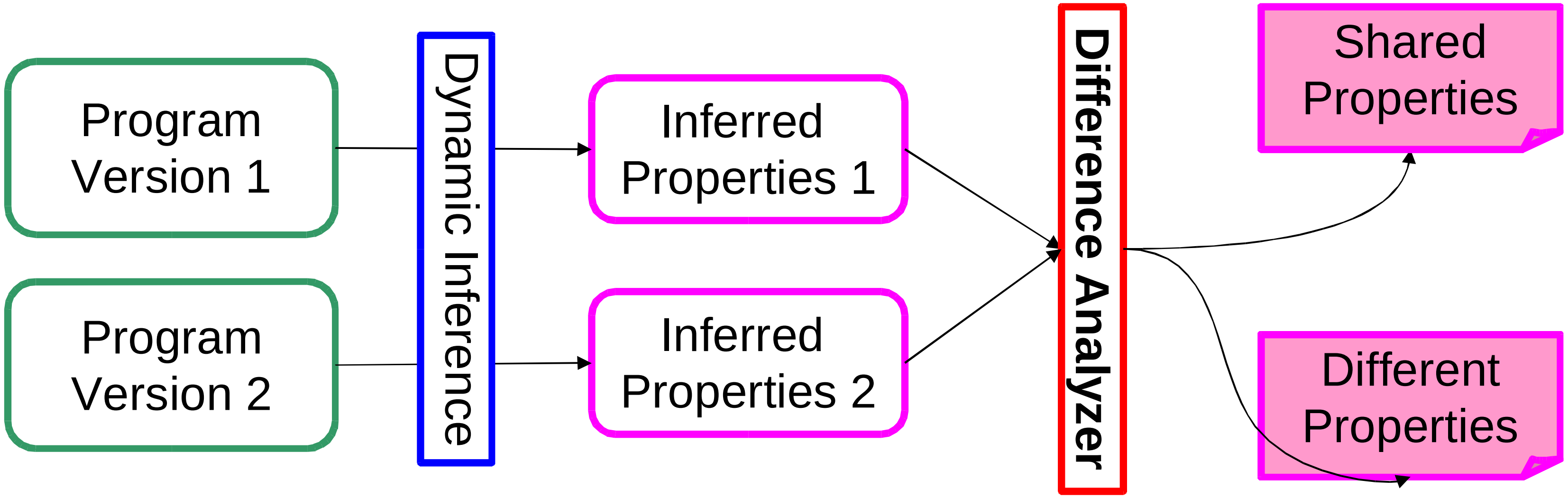


- The traces are generated by running the test suites on an instrumented program. The program is instrumented at all method entry and exit points.
- The current implementation is a 900 line Perl program.
- This approach should have alternative implementations!

Results and Perspective

- By using this approach it is possible to
  - compare the temporal behaviour of different implementations of the same specification.
  - compare different versions of some program to reveal differences in temporal behaviour.
  - use this in conjuction with verification to improve test suites.
  - automatically specify certain system call patterns of operating systems???

Compare Different Versions of a Program

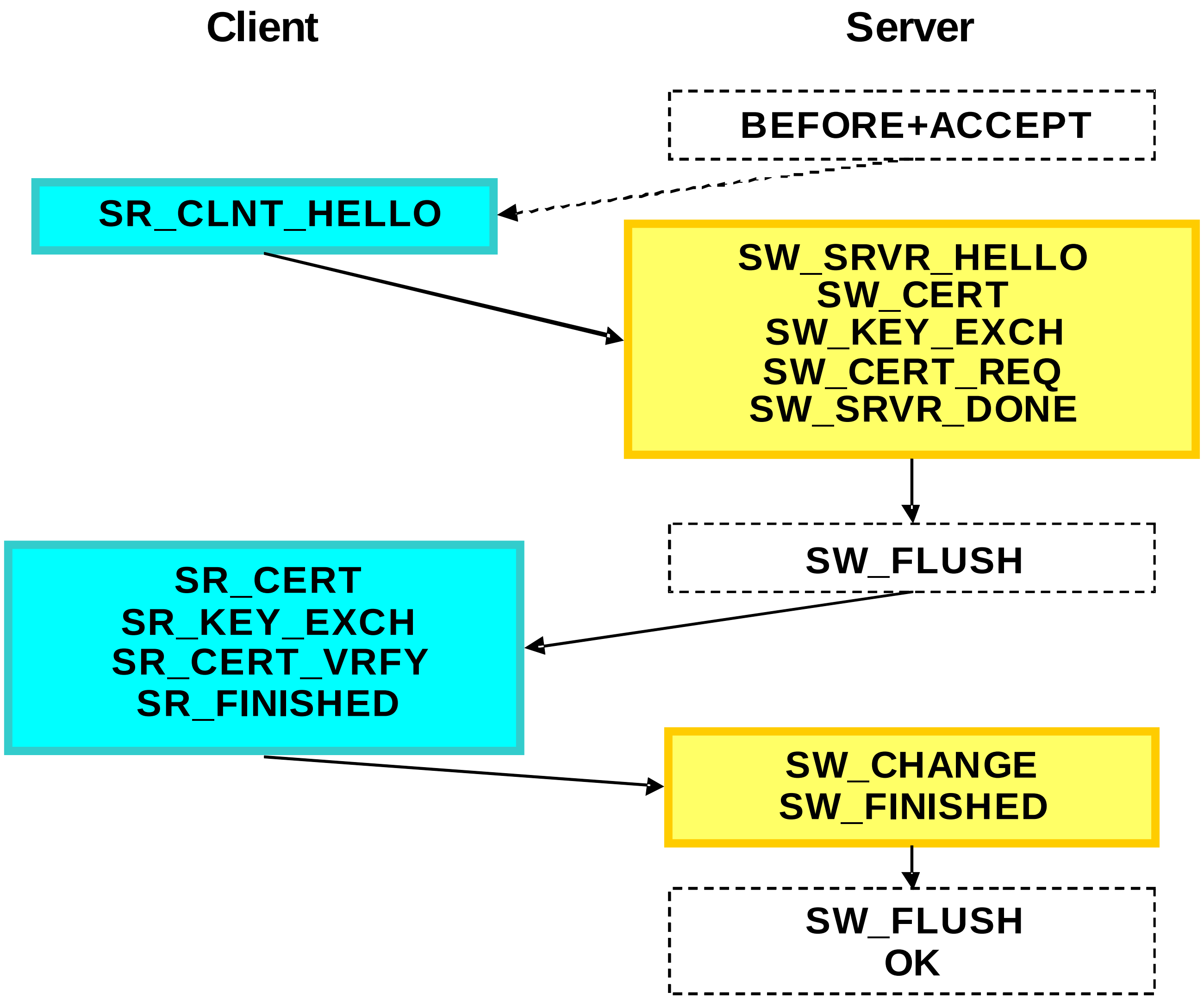


J. Yang

Example: OpenSSL

- A widely used implementation of the Secure Socket Layer protocol
- Yang et al. looked at 6 different versions:

  [0.9.6, 0.9.7, 0.9.7a-d]
- The focus is on the handshake protocol.
- Manually instrumented server
- Modified client
- Executed each version of a server with 1000 randomly generated clients.


Client
Server
BEFORE+ACCEPT
SR_CLNT_HELLO
SW_SRVR_HELLO
SW_CERT
SW_KEY_EXCH
SW_CERT_REQ
SW_SRVR_DONE
SW_FLUSH
SR_CERT
SR_KEY_EXCH
SR_CERT_VRFY
SR_FINISHED
SW_CHANGE
SW_FINISHED
SW_FLUSH
OK

J. Yang

Inferred Alternating Patterns

	0.9.6	0.9.7	0.9.7a	0.9.7b	0.9.7c	0.9.7d
SR_KEY_EXCH→ SR_CERT_VRFY	✓	✓	✓	✓		
SW_CERT→ SW_KEY_EXCH		✓	✓	✓	✓	✓
SW_SRVR_DONE→ SR_CERT		✓				

Documented improvement

Fixed bug

Race condition

7 alternating patterns same for all versions

Other Approaches to Automatic Spec Extraction

- Value relationships between variables
- Machine learning approach that discovers specifications a program must satisfy when interacting with an API
- Extraction of thread behaviour out of program code
- ...

Conclusion

- Automatically inferring temporal properties has yielded practical results.
- Even simple property patterns reveal interesting properties.
- A lot still remains to be done! Like
  - Looking at different property patterns;
  - Building a property difference analyser (for program evolution);
  - Improvement of test suites in conjunction with verification technology.

References

- Jinlin Yang and David Evans, Dynamically inferring temporal properties. Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. 2004.
- Matthew B. Dwyer, George S. Avrunin, James C. Corbett, Patterns in property specifications for finite-state verification. Proceedings of the 21st international conference on Software engineering. 1999.
- Jinlin Yang and David Evans, Automatically Inferring Temporal Properties for Program Evolution. Software Reliability Engineering, 2004. ISSRE 2004.
- Jinlin Yang's home page: http://www.cs.virginia.edu/~jy6q

Thank you!