Corporate Technology

Formal Model Verification in the Industrial Software Engineering

Erwin Reyzl, Siemens AG, Corporate Technology
Vladimir Okulevich, Siemens Russia, Corporate Technology

Software Workhop
“Efficient Development of Reliable Software and Related Methods”
Karlsruhe, Germany
24th January 2008

Dependable Software and Siemens Products

SIEMENS

Dependability of a computing system is the ability to **deliver services that can justifiably be trusted**.

Ref. : A. Avizieniz, J.-C. Laprie, B. Randell: Fundamental Concepts of Dependability

All Siemens Divisions develop and sell products that perform mission-critical operations.

Most of these products contain an ever **increasing software part**.

Dependability is decisive for our commercial success.

Dependability yields higher confidence and acceptance, and is pre-condition to market access.

Following engineering standards gives **evidence of** a product‘s quality and **trustworthiness**.

Dependability relies on an integral **management & engineering** approach.

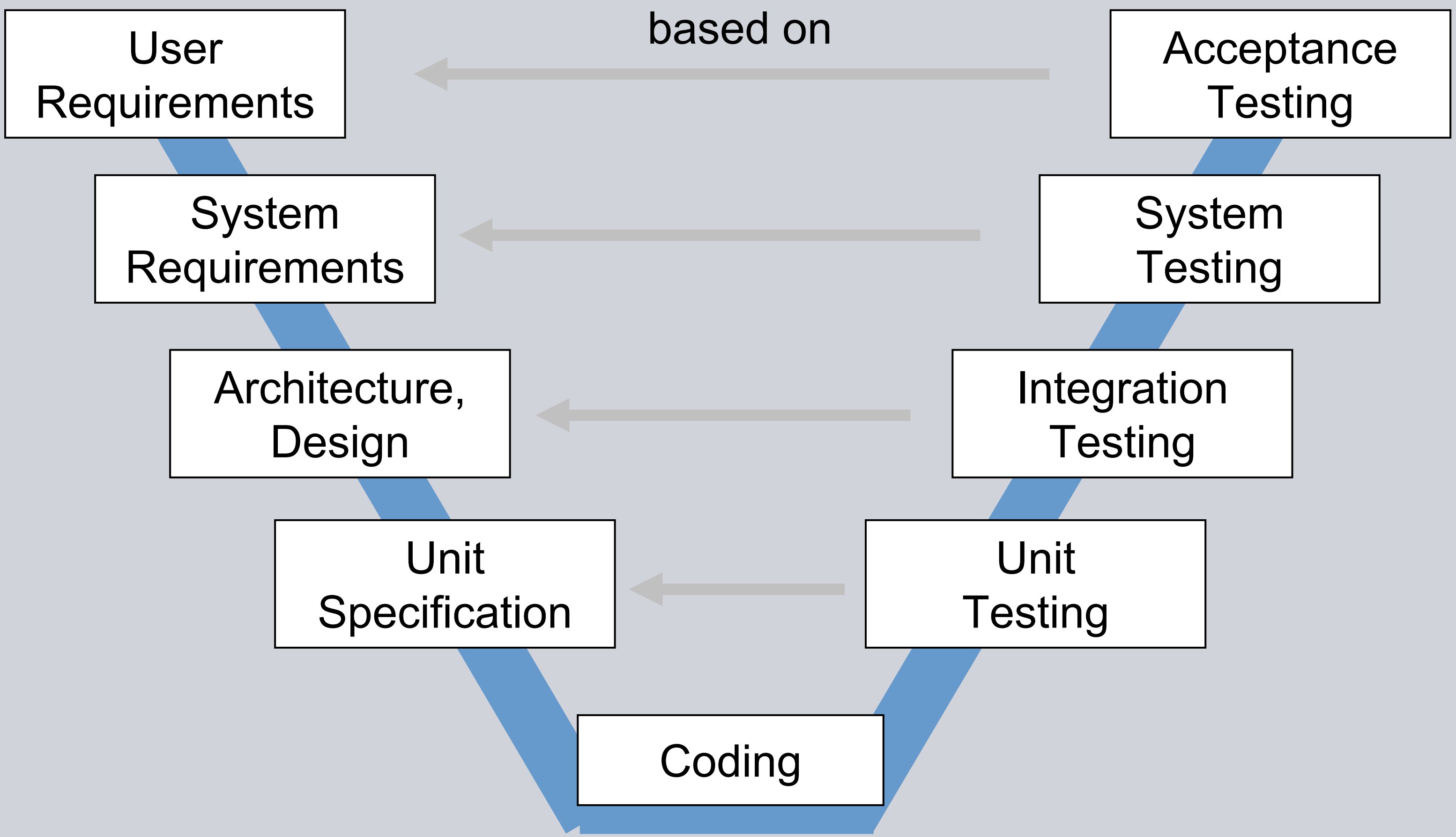
SIEMENS

Dependability Competence Team at Siemens Corporate Technology

- Safety / RAMS Engineering
- RAMS Analysis & Assessment
- Requirement Engineering
- Model Driven Development
- Code Quality Management
- Software Testing & Verification
- Validation & Certification
- Software Architectures
- Real-time Technology
- Embedded Platforms
- Fault-Tolerance
- ...

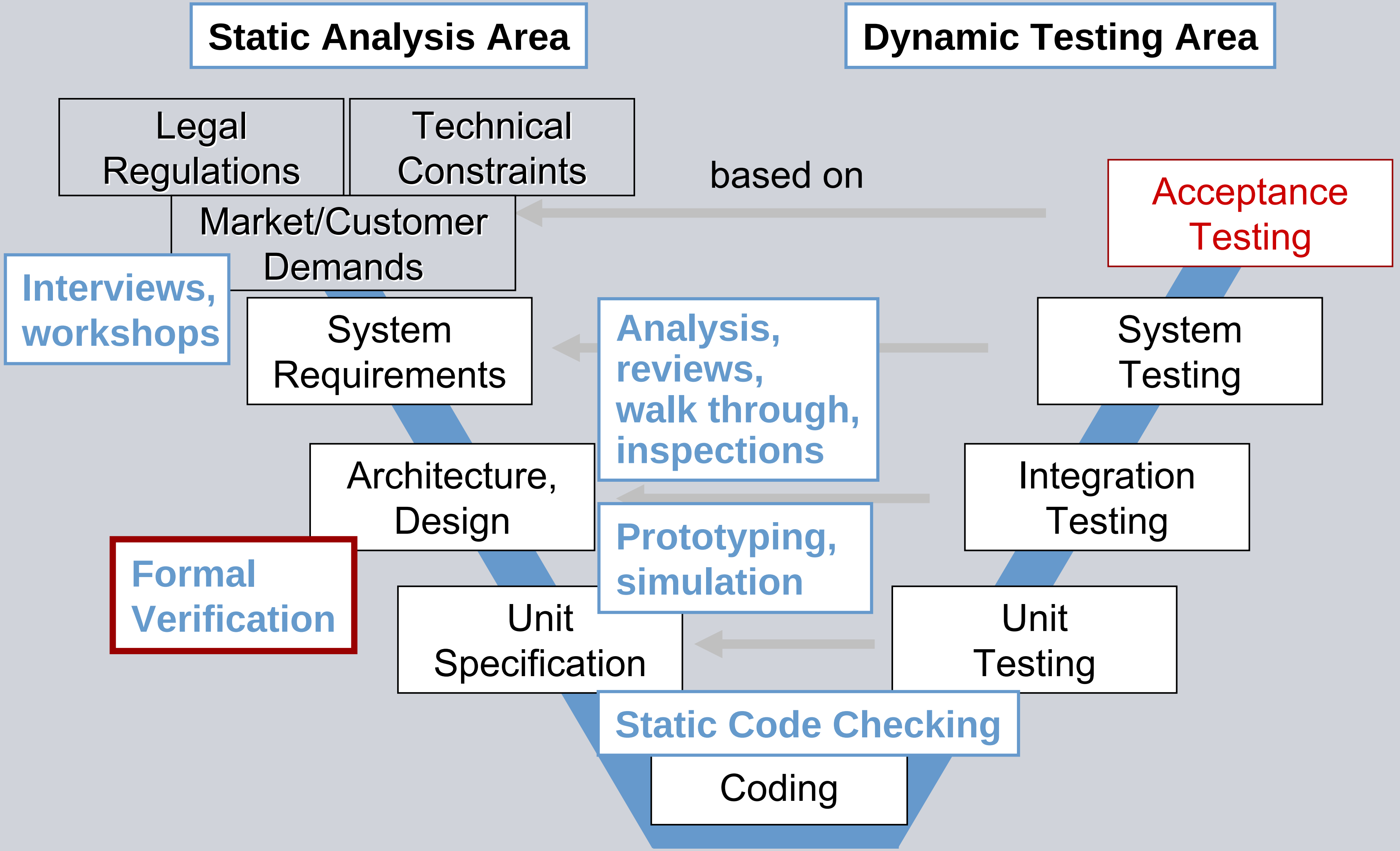
Engineering of high-quality software
Test levels – example V model

SIEMENS



SIEMENS

Engineering of high-quality software
Test levels, Verification & Validation – A closer View



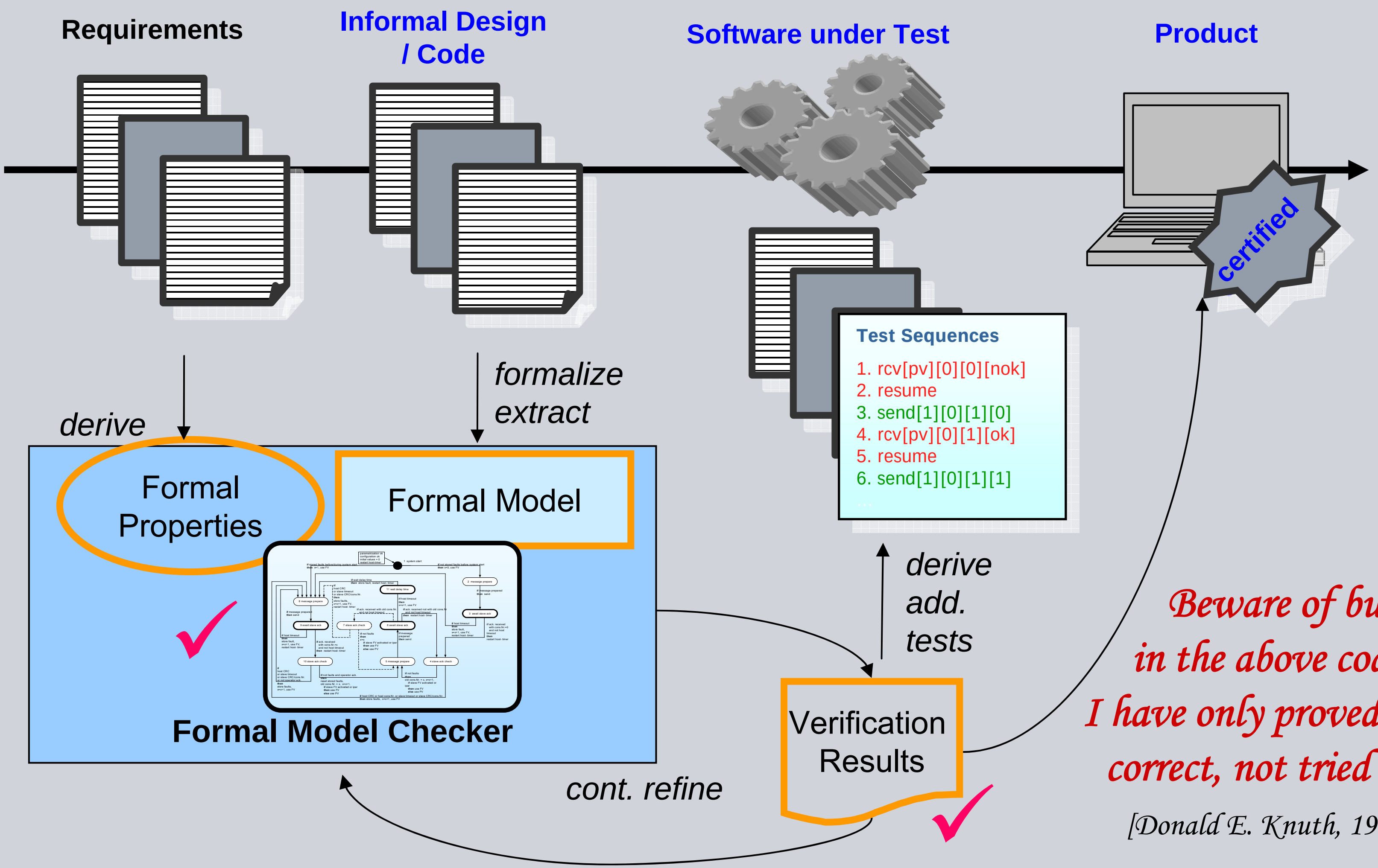
SIEMENS

Formal Verification and Related Terms

- **Software Verification, Software Safety Validation (IEC61508)**
  Verification & Validation, Static Analysis& Dynamic Testing
  - See http://en.wikipedia.org/wiki/Formal_verification :
- **Validation**: "Are we building the right product?",
  i.e., does the product do what the user/the application really requires?
- **Verification**: "Are we building the product right?",
  i.e., does the product conform to the specifications?
- **The verification process consists of static and dynamic parts.**
  E.g., for a software product one can inspect the source code (static) and run against specific test cases (dynamic). Validation usually can only be done dynamically.
- **Formal Method (IEC61508)**
  Formal Specification
- **Formal Proof (IEC61508)**
  Formal Verification, Model Checking/Theorem Proving
  - See http://en.wikipedia.org/wiki/Formal :
    formal methods in computer science, including:
    formal specification describes what a system should do
    formal verification proves correctness of a system

Embedding Formal Verification into Software Development Life-Cycle

SIEMENS



Beware of bugs in the above code; I have only proved it correct, not tried it.

[Donald E. Knuth, 1977]

Formal Model Verification – SPIN tool

SPIN (Bell Labs, G. Holzmann):

Characteristics of method:

1) exhaustive verification

2) space compression

3) probabilistic verification (hashing)

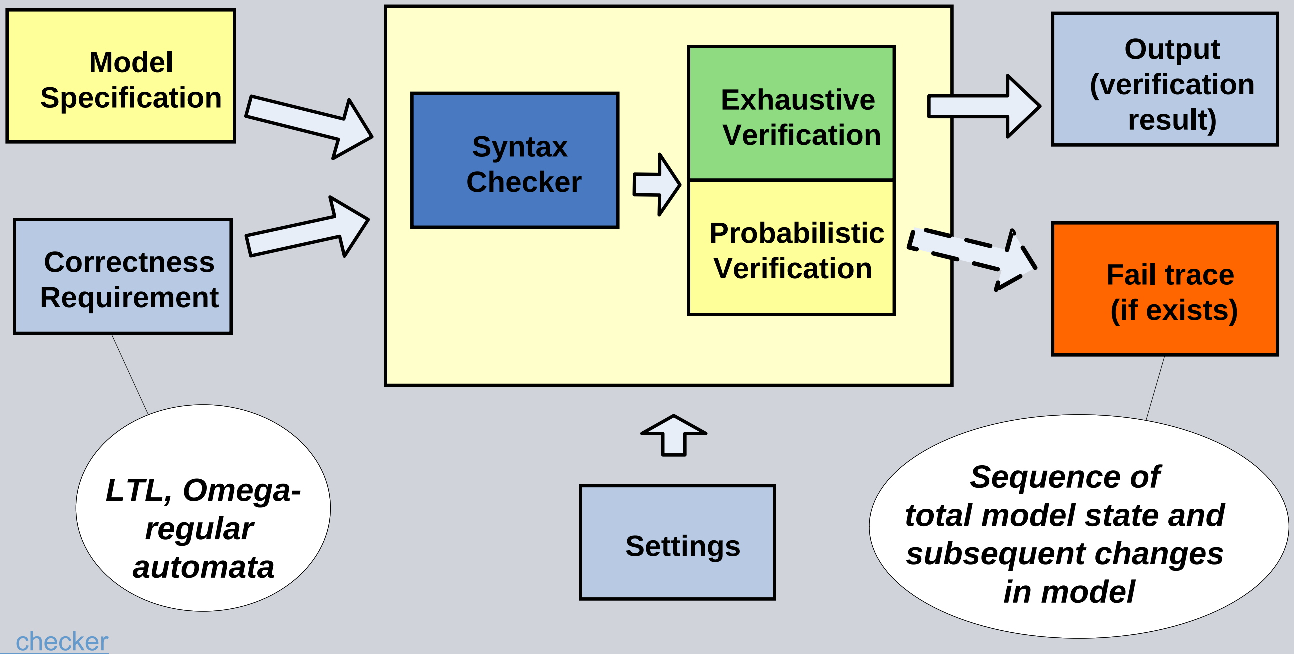
Model in PROMELA (C-like language, CSP-based) is automatically translated into the extended FSMs.

These FSMs are verified to be correct according to correctness requirements.

Correctness requirements are presented in Linear Temporal Logic (LTL)

SPIN-based technologies are used:

- **Bell-Labs (network switches, OS Plan 9)**
- **NASA (Cassini mission at Saturn, Deep Space 1)**
- **Siemens CT**

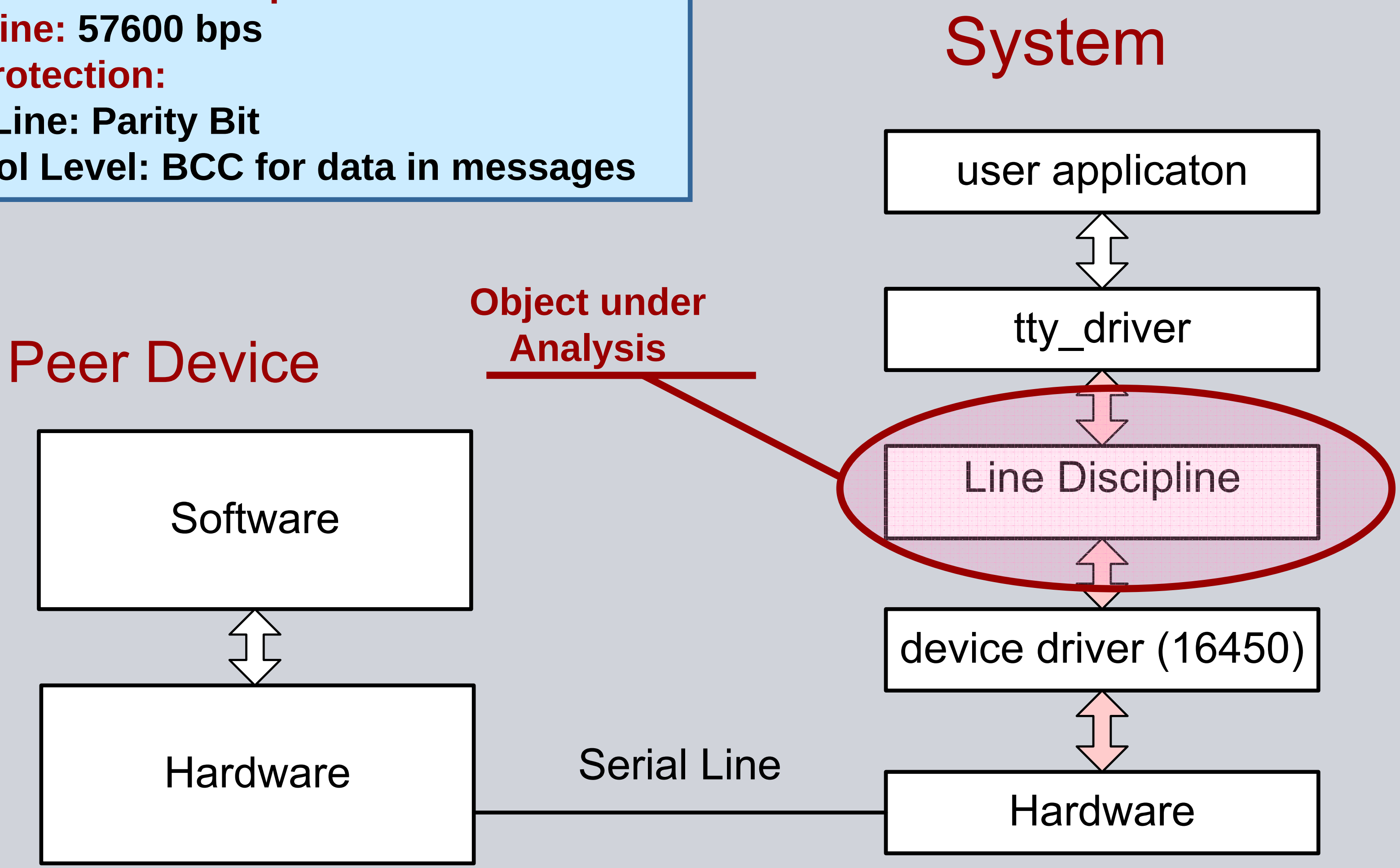


SPIN website
http://spinroot.com/spin/whatispin.html
Wikipedia
http://en.wikipedia.org/wiki/SPIN_model_checker

Example A
System Structure

SIEMENS

Customized Line Discipline
Serial Line: **57600 bps**
Error Protection:
- **Serial Line: Parity Bit**
- **Protocol Level: BCC for data in messages**

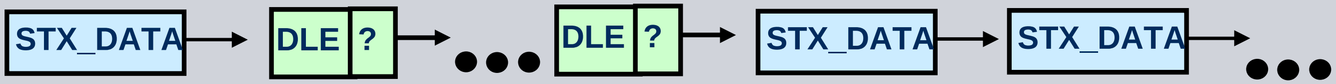


SIEMENS

Example A
Safety Property (1)

Safety Property: Any order of STX_DATA and DLE messages will be correctly received and Receiver achieves TFirstChar state after TDLE2 state.

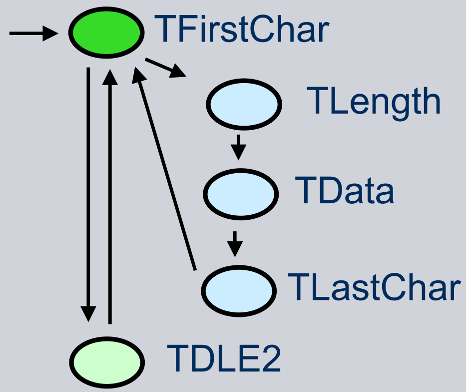
Modeling Solution: Sender generates valid messages STX_DATA and DLE in all possible sequences.



OK:
Property Valid

Property Parts:

#define a (ReceiveState==TDLE2)
#define b (ReceiveState==TFirstChar)



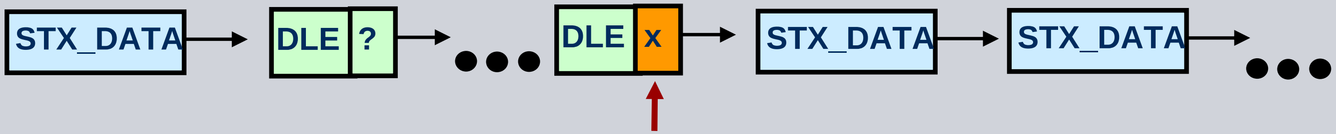
Safety Property for Model Verification in Linear Temporal Logic:

[] (<> a -> <> b)

Example A
Safety Property (2)

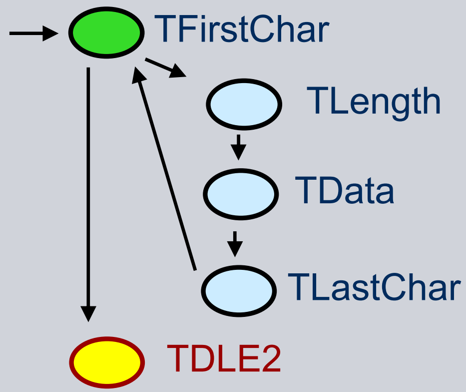
SIEMENS

Safety Property: Any order of STX_DATA and DLE messages will be correctly received and Receiver achieves TFirstChar state after TDLE2 state if the noise char comes suddenly.
Modeling Solution: Sender generates valid messages STX_DATA and DLE in all possible sequences. **We allow receiving of noisy char.**



Error:
Property Invalid

Property Parts:
#define a (ReceiveState==TDLE2)
#define b (ReceiveState==TFirstChar)

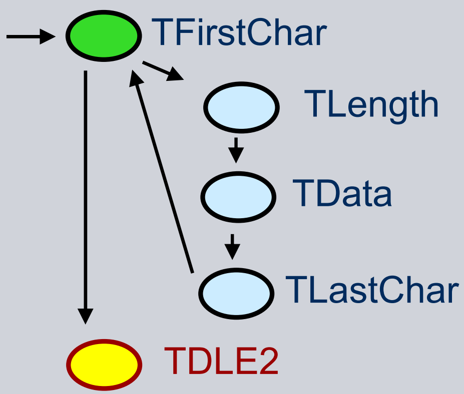


Safety Property for Model Verification in Linear Temporal Logic:
[] (<> a -> <> b)

Example A
Error Analysis for Property (2) : Source Code

SIEMENS

```
switch (theDriverData->ReceiveState) {
 case TFirstChar:
  switch (theChar) {
    (...)
    case TDRVDLE:
      theDriverData->ReceiveState = TDLE2; break;
  }
(...)
 case TDLE2:
  if(theChar == '?' || theChar == TDRVENQ)
   theEvent = TDLEReceived;
  if(atomic_read(&theDriverData->WabtCounter) == 50) {
    atomic_set(&theDriverData->WabtCounter, 0);
    theDriverData->ReceiveState = TFirstChar;
  }
break;
(...)
```



Error:
Property Invalid

Explanation:
SPIN shows that DRV could be blocked if noisy (incorrect) char came after TDRVDLE.

NOTE: the real driver will wait 50 timeouts (~5-20 sec) before it starts receiving of next telegram (e.g. STX_DATA)

Example A
FMV Results for the Driver

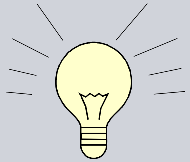
SIEMENS

Operational threats identified by SPIN:

- Long delay identified if 1 noise char comes
- Data loss within telegram receiving found

Model-building review:

- Performance bottleneck could create high interrupts latency
- Wrong API-version calls
- “Magic Constants” used in code
- “Dead Code” identified



SIEMENS

Summary (1)
Objects and Focus Setting

Objects under Analysis:
- Protocols and Interfaces (especially under construction)
- Interacting components (e.g. new architecture or critical mechanism)
- Data access and control logic in parallel and distributed system

Additional Focus set on:
- System Initialization
- Restart of Components
- Communication Delays and Faults

Fault Types:
- Deadlock/Livelock
- Endangered Safety
- Integrity violation
- Correctness violation
- Non-expected communication order
- Race Conditions
- …

Robustness Aspects:
- Standard operation
- Unpredictable rare impact
- "Aggressive/Noisy Environment"

Dos & Don'ts

- Improve verification capabilities in early phases by introducing formal techniques.
- Increase precision in specifications. Apply formal techniques at design and fight ambiguities.
- Promote formally motivated checks into standard peer reviews.
- Focus formal techniques on architectural hotspots: complex, critical, risky, central parts
- Exploit formal results for test case definition: use failure traces to focus tests on design flaws.

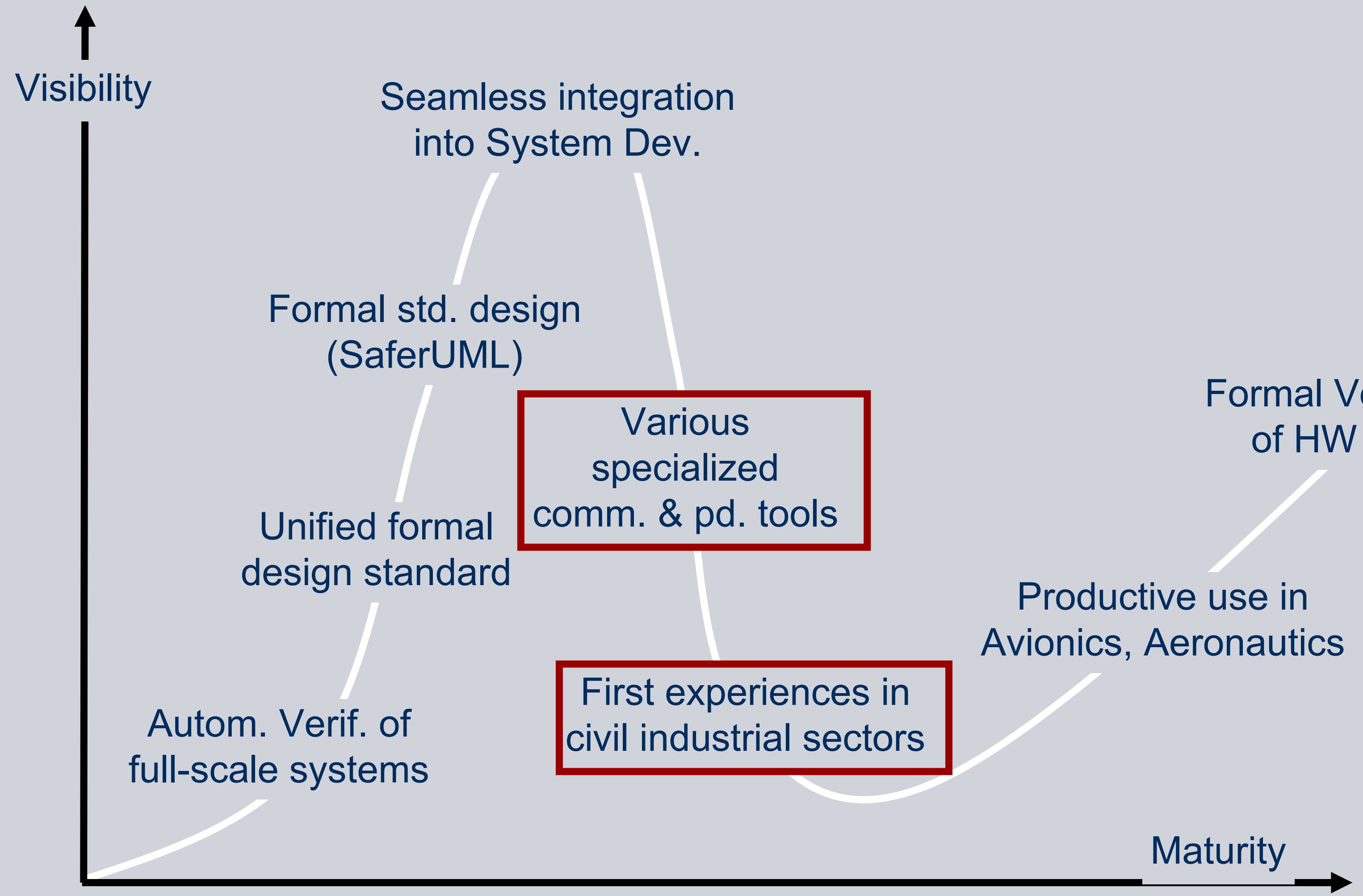
- Don't believe in wonders. Formal verification is not cheap and needs invests in early phases.
- Don't act without concept. Tools need evaluation, and competence needs to be built.
- Don't apply formal techniques as rescue belt. It's not to patch ambiguous results from less mature processes.
- Don't believe you will not need to test your software any more. Formal verification does not replace testing phases.
- Don't believe your software is completely verified. You only proved that a model fulfills certain properties. That's it – no more, no less.

Don't split theory and practice.

Closely align work of formal teams and safety/development teams.

SIEMENS

The Formal Hype Cycle



SIEMENS

Challenges for Formal Model Verification

Challenges

- Decrease modeling efforts
- Increase usability, reduce qualification degree needed
- Integration with tools for software development
- Traceability from modeling phase to testing phase
- Automated properties and models extraction
  from heterogeneous input material
- Again coverage and completeness issues
  “How much will be sufficient?”
- …

Model Checking gains importance:

“The behavior of even nonbuggy distributed applications can easily defy our human reasoning skills.”

Logic Verification of ANSI C code with SPIN
Gerard J. Holzmann

Summary (2)
Reasons for Formal Model Verification

SIEMENS

Standards recommend formal methods and proofs
e.g. IEC61508 : SIL2/3/4 - for design, verification, safety validation.

Formal methods emerge at the industry sector
easier to use tooling (Open Source, Tool Vendors), best practices (space/military, avionics, transportation/automotive, Microsoft).

Formal methods improve precision within development, capturing and ensuring functional and **non-functional properties**.

Early correctness proof of design concepts prevents design faults to propagate into development, test and operation phase.

Byzantine failures with hard to identify root-causes often are the consequence of weakly defined or misunderstood requirements.

Environmental impact and sporadic influences which are hard to test can be incorporated into formal models.

Stronger evidence of safety-related claims; amends test results and **improves acceptance by certification authorities**.

SIEMENS

Conclusions

•Increasing complexity and importance of software

More and more safety-relevant functions, which nowadays might be executed manually by human, will be realized in software and taken over automatically by the technical system.

•Traditionally software plays a subordinate role

In systems engineering and also in relevant standards the current perspective on software is that of an subordinate element. This is expected to change with the growing pervasiveness of software especially in safety-relevant development.

•Formal verification in practice applied to selected software parts

In the current practice formal verification is applied to verify selected system aspects. It already proved usefulness and applicability.

•Cost and complexity of formal techniques are further high

Up to now formal verification is not an easy-to-use technique. At this time it is not seen to enable a complete software/system verification.

•Formal Verification does not/will never replace systematic testing

Formal verification adds precision to the traditional verification process. It extends, but does not replace rigorous testing. Size limitations and abstractions of models through formal verification are to be carefully verified in reality.

SIEMENS

Thank you for your attention.

Do you have some questions ?

Corporate Technology

Backup

Fundamental Concepts of Dependability (A. Avizieniz, J.-C. Laprie, B. Randell)

SIEMENS

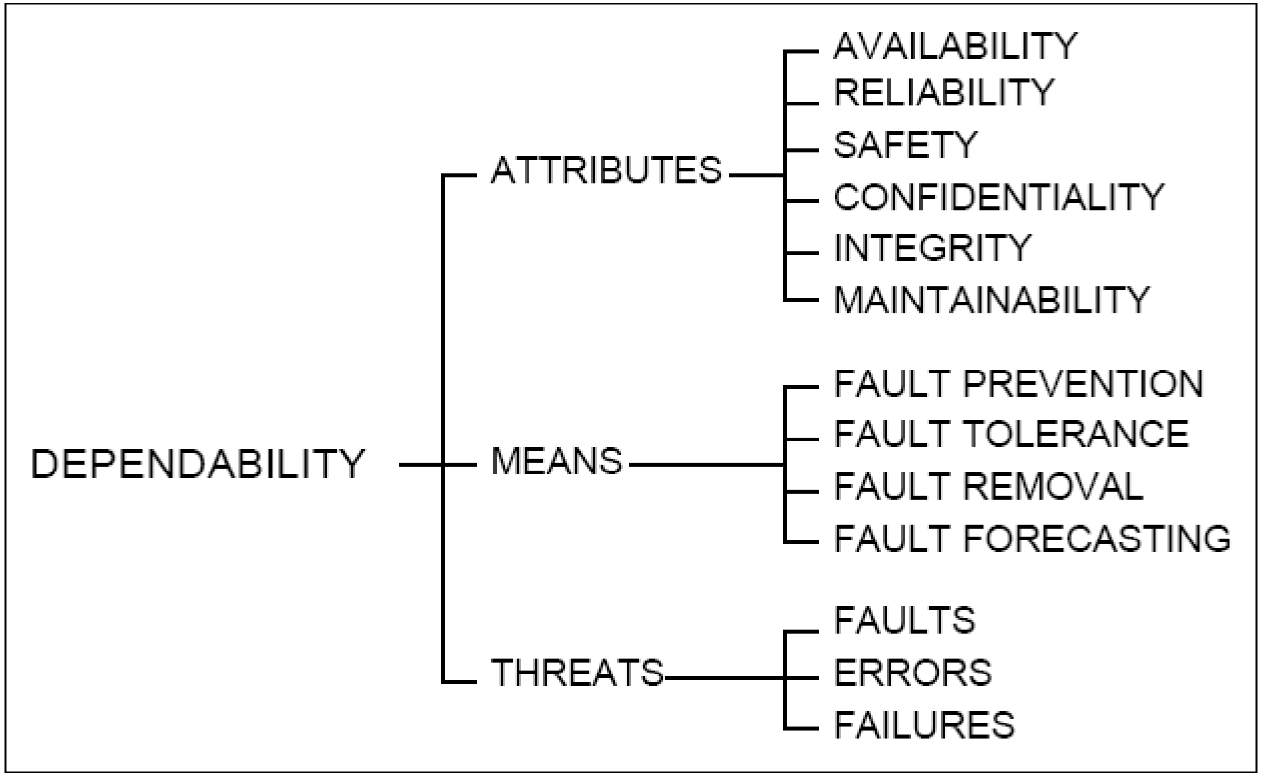


Figure 1 - The dependability tree

Computing systems are characterized by four fundamental properties: functionality, performance, cost, and dependability.

Concepts of Dependability developed by A. Avizieniz, J.-C. Laprie, B. Randell

SIEMENS

Definitions: Dependability Attributes

Dependability is an integrative concept that encompasses the following system attributes:

- **Availability:** readiness for correct service
- **Reliability:** continuity of correct service
- **Safety:** absence of catastrophic consequences on the user(s) and the environment
- **Confidentiality:** absence of unauthorized disclosure of information
- **Integrity:** absence of improper system state alterations
- **Maintainability:** ability to undergo repairs and modifications

Compound attributes:

- **Survivability:** system capability to resist a hostile environment so that it can fulfill its mission (MIL-STD-721, DOD-D-5000.3)
- **Security:** Dependability with respect to the prevention of unauthorized access and/or handling of information (Laprie, 1992)

*** RAM / RAMS:** acronyms for reliability, availability, maintainability, and safety

SIEMENS

IEC61508-3: Formal Methods

Formal methods are a specification technique.

Formal methods (see IEC61508-7, C2.4) are for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z.

Formal methods are recommended (R SIL2/3, HR SIL4) for

- 7.2/Table A.1:
  Software safety requirements specification
- 7.4.3/Table A.2:
  Software design and development: software architecture design
- 7.4.5/Table A.4:
  Software design and development: detailed design
- 7.7/Table A.7/Table B5
  Modeling in the context of software safety validation

Sometimes mixed up with semi-formal methods e.g. finite state machines (FSM)

- semi-formal methods (table B.7):
  Logic/function block diagrams, sequence diagrams, data flow diagrams, finite state machines/state transition diagrams, e.a.

IEC61508-7, C2.4: Formal Methods (ii)

•Focus: Logic/HW

- **HOL – Higher Order Logic for HW verification**
- **Temporal logic – Formal demonstration of safety and operational requirements**

•Focus: Sequential processes

- **OBJ – Algebraic specification of operations on abstract data types (ADT, similar to ADA packages).**
- **Z – Specification language notation for sequential systems**
- **VDM – Vienna Development Method (VDM++ concur. extension)**

•Focus: Communicating concurrent processes

- **LOTOS, extends CCS – Calculus of Communicating Systems**
- **CSP – Communicating Sequential Processes**

•Other semi-formal techniques (see B.2.3.2)

- **Finite state machines/state transition diagrams for control structures**
- **Petri nets (graph theory) for concurrent, asynchronous control flow; extension: time concept, data/information flow**

SIEMENS

IEC61508-3: Formal Proofs

Formal proofs are recommended (R SIL2/3, HR SIL4) for

- IEC61508-3/7.9/Table A.9: Software verification

Formal proofs are a static means for software verification

NOTE 3 – In the early phases of the software safety lifecycle verification is static, for example inspection, review, formal proof. When code is produced dynamic testing becomes possible. It is the combination of both types of information that is required for verification.

For example code verification of a software module by static means includes such techniques as software inspections, walk-throughs, static analysis, formal proof. Code verification by dynamic means includes functional testing, white-box testing, statistical testing.

It is the combination of both types of evidence that provides assurance that each software module satisfies its associated specification.

Sometimes mixed up with static analysis e.g. symbolic execution:

- Static analysis (table A.9, table B.8): e.g. Walk-through/design reviews, control flow / data flow analysis, or symbolic execution, e.a.

Outline of Formal Model Verification

SIEMENS

Main Steps:

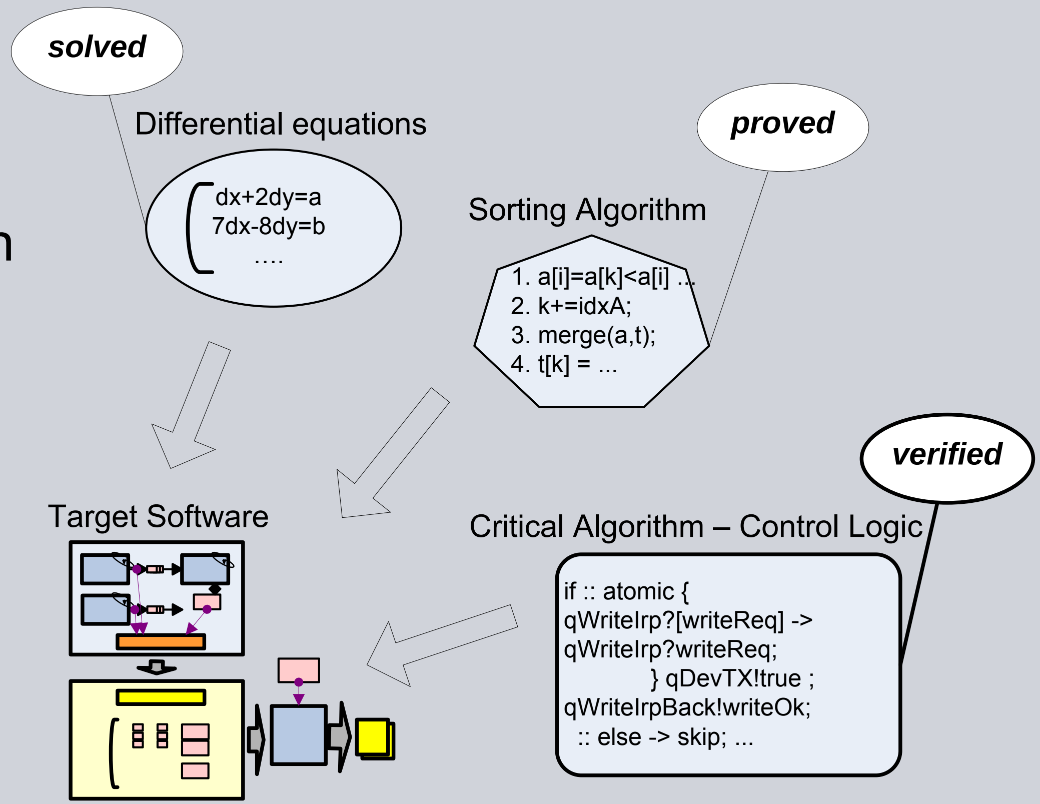
Objects under Analysis
identification in software project

Correctness properties definition
for Objects under Analysis

Creation of model for
Objects in PROMELA

Model verification by SPIN and
findings analysis

Report preparation on
verification results



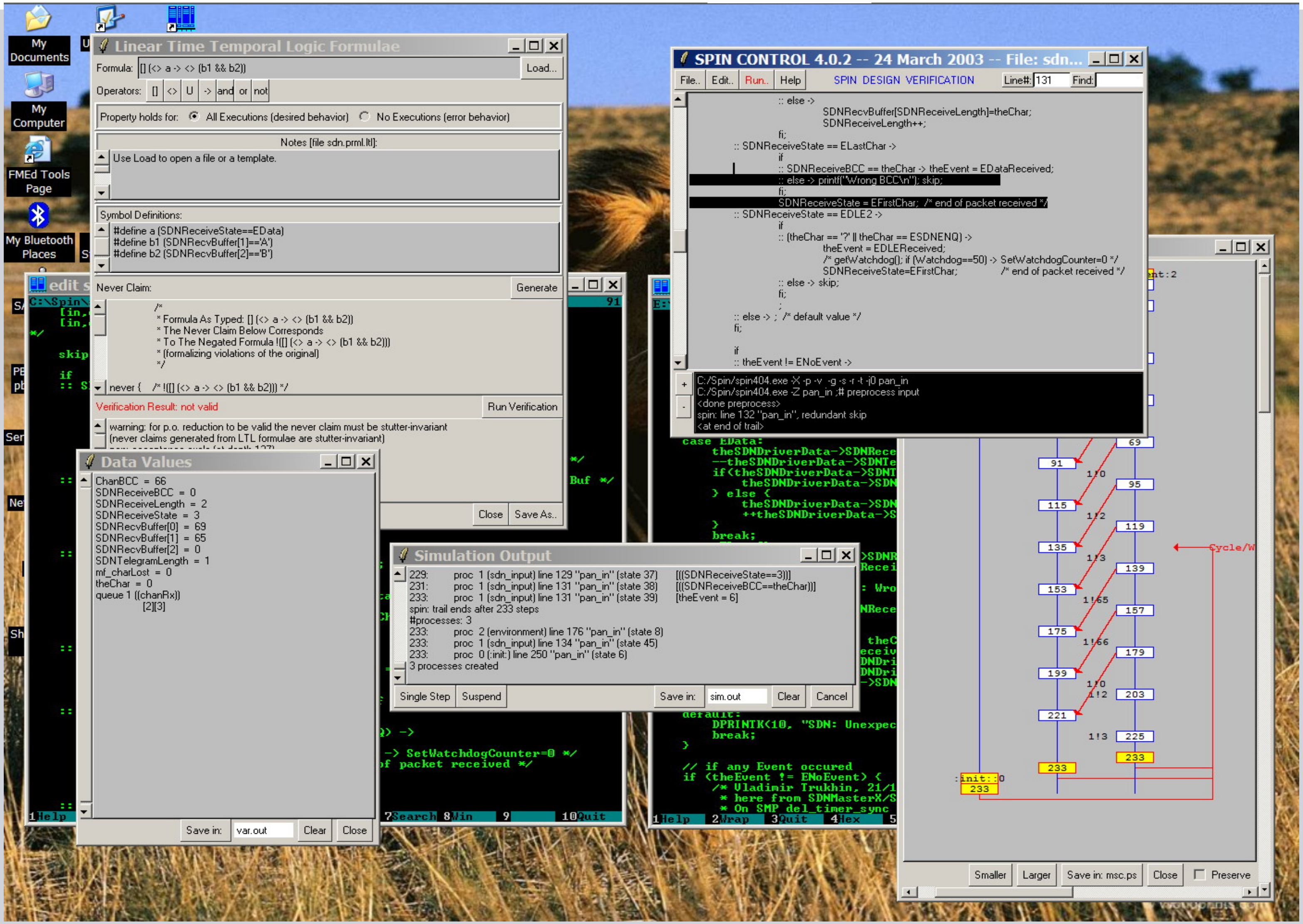
Further promising actions:

- Automate procedure of model creation from C/C++ sources

SPIN model checker (ii)

Wikipedia http://en.wikipedia.org/wiki/SPIN_model_checker
SPIN is a tool for software model checking. It was written by Gerard J. Holzmann and others, and has evolved for more than 15 years. SPIN is an automata-based model checker. Systems to be verified are described in Promela (*Pro*cess *Me*ta *La*nguage), which supports modeling of asynchronous distributed algorithms as non-deterministic automata. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Büchi automata as part of the model-checking algorithm. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. Since 1995, (approximately) annual SPIN workshops have been held for SPIN users, researchers, and those generally interested in model checking. In 2001, the Association for Computing Machinery awarded SPIN its System Software Award.
Holzmann, G. J., *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004. ISBN 0-321-22862-6.
SPIN website http://spinroot.com/spin/whatispin.html

Snapshot of SPIN Verification Screen



SPIN model checker (iii) – References

SIEMENS

Wikipedia http://en.wikipedia.org/wiki/SPIN_model_checker
SPIN Website http://spinroot.com/spin/whatispin.html
An overview paper of Spin, with verification examples, is:
The Model Checker Spin,
IEEE Trans. on Software Engineering,
Vol. 23, No. 5, May 1997, pp. 279-295.
(PDF)
The automata-theoretic foundation for Spin:
An automata-theoretic approach to automatic program verification,
by Moshe Y. Vardi, and Pierre Wolper,
Proc. First IEEE Symp. on Logic in Computer Science,
1986, pp. 322-331.
(PDF)

Linear Temporal Logic : SYNTAX

LTL - Linear Temporal Logic
Best to specify safety and correctness properties
See also http://en.wikipedia.org/wiki/Linear_Temporal_Logic

SYNTAX
Grammar: ltl ::= opd | (ltl) | ltl binop ltl | unop ltl

Unary Operators (unop):
- [] (the temporal operator *always*),
- <> (the temporal operator *eventually*),
- ! (the boolean operator for *negation*)

Binary Operators (binop):
- U (the temporal operator *strong until*)
- V (the dual of U): (p V q) == !(!p U !q)
- && (the boolean operator for *logical and*)
- || (the boolean operator for *logical or*)
- -> (the boolean operator for *logical implication*)
- <-> (the boolean operator for *logical equivalence*)

Operands (opd): Predefined: true, false

SIEMENS

Linear Temporal Logic (ii)

Extension of classical logic ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\forall$, $\exists$)

- works over an (infinite) sequence of states

New operators:

○ *next time*

○ φ φ holds at time t + 1

◊ *eventually*

◊ φ φ holds at some time t + n

□ *always*

□ φ φ holds for all future times t + n

U *until*

φ U ψ φ holds for all future times until the time where ψ holds

∩ *release*

$\varphi \cap \psi$ either φ holds forever, or until φ and ψ holds at the same time

