Marktoberdorf 2010 Overview

Martin Děcký

DEPARTMENT OF DISTRIBUTED AND DEPENDABLE SYSTEMS
http://d3s.mff.cuni.cz/

CHARLES UNIVERSITY IN PRAGUE
FACULTY OF MATHEMATICS AND PHYSICS
DISCLAIMER

The following slides are a compilation of lectures and other materials presented at the 40th International Summer School in Marktoberdorf 2010. No matter if the information is provided in verbatim or paraphrased, it should be considered as being created and owned solely by the respective authors (who are clearly denoted on the slides) and their co-workers.

By no means should be the contents of this compilation considered an original research of the author of this compilation. The author of this compilation declares that he is not taking any credit, directly nor implied, for the contents of this compilation.
Marktoberdorf 2010

• 40th International Summer School
  • August 3 – 15, 2010
  • Marktoberdorf, Bavaria, Germany
  • Presented by
    − Advanced Study Institute of the NATO Science for Peace and Security Programme (http://www.nato.int/science/)

• Supported by
  − DAAD (http://www.daad.de/)
  − Microsoft Research (http://research.microsoft.com/)
  − Town and county of Marktoberdorf (http://www.marktoberdorf.de/)
This year's general topic (orange track)
- *Software and Systems Safety: Specification and Verification*

**Academic directors**
- Manfred Broy (Technical University Munich, Germany)
- David Harel (The Weizmann Institute of Science, Israel)
- Tony Hoare (Microsoft Research, UK)

**Executive director**
- Katharina Spies (Technical University Munich, Germany)

**My participation**
- Supported by the grant SVV-2010-261312
This year's lecturers

• Manfred Broy
  • Professor at TUM
    - Chair at Software & Systems Engineering
  • Model-Driven Development of Reliable Services

• Tony Hoare
  • Microsoft Research
  • Author of Quicksort and CSP
  • Unifying Models of Data Flow
This year's lecturers (2)

- Ed Brinksma
  - Director & Chair at Embedded Systems Institute, Eindhoven
  - Author of LOTOS specification language
  - Model-Based Testing

- Carlo Ghezzi
  - Professor & Chair at Politecnico di Milano
    - Department of Electronics and Information
  - Issues of Adaptable Software for Open-World Requirements
This year's lecturers (3)

- **Susanne Graf**
  - Deputy Director of VERIMAG Laboratory in Grenoble (*Distributed and Complex Systems*)
    - Senior Researcher at CNRS
  - *Abstraction for System Verification*

- **John Harrison**
  - Intel Corporation
  - Author of *Handbook of Practical Logic and Automated Reasoning* and *HOL Light* interactive theorem prover
  - *Formal Verification*
This year's lecturers (4)

- Connie Heitmeyer
  - Head of Software Engineering Section
    - Center for High Assurance Computer Systems, U.S. Naval Research Lab
  - Requirements Models for System Safety and Security

- Holger Hermanns
  - Professor, Head & Dean at Dependable Systems Software Group
    - Saarland University, Saarbrücken
  - From Concurrency Models to Numbers: Performance, Dependability, Energy
This year's lecturers (5)

- **Kim Larsen**
  - Professor at Aalborg University
  - Author of *UPPAAL*
  - *Model-Based Verification and Analysis of Real-Time Systems*

- **Doron Peled**
  - Professor at Bar Ilan University, Israel
  - Major contributor to *SPIN*
  - Co-author of *Model Checking* (MIT Press 2000)
  - *Model Checking*
This year's lecturers (6)

• John Rushby
  • Program Director for *Formal Methods and Dependable Systems*
    - Computer Science Laboratory, SRI Int.
  • Author of SAL SMT-based theorem prover
  • *Formal Methods and Argument-Based Safety Cases*
This year's lecturers (7)
This year's participants
Intermezzo

We Won!
Intermezzo

We Won!

6 : 2
Intermezzo
Intermezzo
Intermezzo
Intermezzo
Intermezzo
Only briefly

- **Susanne Graf**
  - Mathematical basis of specification abstraction for state space reduction
    - Galois connexions

- **Carlo Ghezzi**
  - Software and services adaptation, model-checking in run-time, fault prediction and detection
    - CTL, DTMC, PRISM tool
    - See D3S seminar on October 11 2010
Only briefly (2)

• Kim Larsen
  • Checking of functional and extra-functional properties in real-time systems
    – *UPPAAL*, Timed Automata, symbolic verification
    – *Priced Timed Automata, relation to game theory*

• Holger Hermanns
  • Mathematical basis of DTMC, CTMS
    – Case studies
      • Verification of communication protocols
      • *Motor-powered bike with wireless communication between handles and brakes (Holger drove it and didn't kill himself!)*
Only briefly (3)

- Doron Peled
  - Classical model-checking, LTL, CTL, PCTL
    - SPIN
- Manfred Broy
  - Mathematical basis of compositional verification of software
  - Assumptions a guarantees
    - Mostly based on formalisms related to FOCUS component model
How to complement a Buchi automaton?

- “It’s complicated” [Facebook, 2007]
- Can ask for the negated property (the sequences that should never occur).
- Can translate from LTL formula to automaton A, and complement A. But: can translate ¬ into an automaton directly!
Specification, verification, architecture ...

Informal requirements

Requirements Engineering
Validation
Formalized system requirements in terms of service taxonomies

System delivery
System verification
\[ R \Rightarrow S \]

Integration
\[ R = R_1 \otimes R_2 \otimes R_3 \otimes R_4 \]

Component implementation verification
\[ R_1 \Rightarrow S_1 \]
\[ R_2 \Rightarrow S_2 \]
\[ R_3 \Rightarrow S_3 \]
\[ R_4 \Rightarrow S_4 \]

Architecture design
Architecture verification
\[ S \leftarrow S_1 \otimes S_2 \otimes S_3 \otimes S_4 \]
Tony Hoare: Unified Data Flow

Unifying...

- Memory
  - shared/private, weakly/strongly consistent

- Communication
  - synchronised/buffered, reliable/unreliable

- Allocation
  - dynamic/nested, disposed/collectioned

- Concurrency
  - threads/processes, coarse/fine-grained
Tony Hoare: Unified Data Flow

The sequential design pattern

Defined in relational algebra as:

\[(s \rightarrow s)^+ \delta\]
Tony Hoare: Unified Data Flow

A parameter $x$

$x := w$

$x := w$

$x := w$

$x := w$
Tony Hoare: Unified Data Flow

Assignment

\[ x := v \]

\[ x := v \]

\[ x := v \]

\[ x := v \]

\[ x := w \]
The token game (1)
Tony Hoare: Unified Data Flow

The token game (2)
Tony Hoare: Unified Data Flow

The token game (3)

\[
x := v \\
x =: v \\
x =: v \\
x =: v \\
x := v \\
x := w
\]
Tony Hoare: Unified Data Flow

The token game (4)

\[ x := v \]

\[ x := v \]

\[ x := v \]

\[ x := v \]
Tony Hoare: Unified Data Flow

The token game (5)
Tony Hoare: Unified Data Flow

Communication

```
c!3  c!7  c!9
  v    v    v
c?3  c?7  c?9
```
Tony Hoare: Unified Data Flow

Sequential outputs/inputs
Tony Hoare: Unified Data Flow

Channel

c!3 → c!7 → c!9

c?3 → c?7 → c?9

δc
Tony Hoare: Unified Data Flow

Threads

T1: fork

T2:

T3: join
Tony Hoare: Unified Data Flow

A shared variable

T1

x := 4

T2

x := 3

x := 6

x :=: 3

x :=: 4

x :=: 6
Tony Hoare: Unified Data Flow

Memory barriers

**T1**
- x := 1
- x := 2
- x := 3

**T2**
- x ::= 2
- x ::= 1
- x ::= 3

Valid
Tony Hoare: Unified Data Flow

Memory barriers

T1
x := 1 → x := 2 → B
   ^       ^       |
   |       |       |
   |       |       |
   |       |       |
   v       v       v
T2
x := 1 → x := 3 → x := 2

invalid
Tony Hoare: Unified Data Flow

Cache

T1 cache

\( x_1 := 4 \)

\( x_1 := 3 \)

\( x_1 := 6 \)
Tony Hoare: Unified Data Flow

A second cache

T1 cache
x₁ := 4 → x₁ := 3 → x₁ := 6

T2 cache
x₂ := 3 → x₂ := 4 → x₂ := 6
Tony Hoare: Unified Data Flow

Partial store ordering

T1 cache

x₁ := 4 → x₁ := 3 → x₁ := 6

T2 cache

x₂ := 3 → x₂ := 4 → x₂ := 6
Tony Hoare: Unified Data Flow

Total store ordering

T1 cache

\[ x \leftarrow 4 \]
\[ x \leftarrow 3 \]
\[ x \leftarrow 6 \]

main memory

\[ x \leftarrow 4 \]
\[ x \leftarrow 3 \]
\[ x \leftarrow 6 \]

T2 cache

\[ x_2 \leftarrow 3 \]
\[ x_2 \leftarrow 4 \]
\[ x_2 \leftarrow 6 \]

\[ x \leftarrow 3 \]
\[ x \leftarrow 4 \]
\[ x \leftarrow 6 \]
Tony Hoare: Unified Data Flow

Summary

• Data flow is a primitive concept,
  – adequate to describe the dynamic behaviour of many kinds of computing resource.

• Relational calculus,
  – illustrated by labelled graphs,
  – provides a general framework adequate for a unifying theory of data flow
Traditionally, propositional logic has been regarded as fairly boring.
- There are severe limitations to what can be said with propositional logic.
- Propositional logic is trivially decidable in theory.
- Propositional satisfiability (SAT) is the original NP-complete problem, so seems intractible in practice.

But . . .
The last decade or so has seen a remarkable upsurge of interest in propositional logic.

**Why the resurgence?**
- There are many interesting problems that can be expressed in propositional logic.
- Efficient algorithms can often decide large, interesting problems of real practical relevance.

The many applications almost turn the ‘NP-complete’ objection on its head.
John Harrison: Theorem Provers

- Logic and circuits
  - The correspondence between digital logic circuits and propositional logic has been known for a long time

<table>
<thead>
<tr>
<th>Digital design</th>
<th>Propositional Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit</td>
<td>formula</td>
</tr>
<tr>
<td>logic gate</td>
<td>propositional connective</td>
</tr>
<tr>
<td>input wire</td>
<td>atom</td>
</tr>
<tr>
<td>internal wire</td>
<td>subexpression</td>
</tr>
<tr>
<td>voltage level</td>
<td>truth value</td>
</tr>
</tbody>
</table>
Logic and circuits

The correspondence between digital logic circuits and propositional logic has been known for a long time

- Many problems in circuit design and verification can be reduced to propositional tautology or satisfiability checking (‘SAT’)
- For example optimization correctness
  - $\varphi \Leftrightarrow \varphi'$ is a tautology
John Harrison: Theorem Provers

• Combinatorial problems
  • Many other apparently difficult combinatorial problems can be encoded as Boolean satisfiability, e.g. scheduling, planning, geometric embeddibility, even factorization

\[
\neg((\text{out}_0 \iff x_0 \land y_0) \land (\text{out}_1 \iff (x_0 \land y_1 \iff \neg(x_1 \land y_0))) \land (v_2^2 \iff (x_0 \land y_1) \land x_1 \land y_0) \land (u_2^0 \iff ((x_1 \land y_1) \iff \neg v_2^2)) \land (u_2^1 \iff (x_1 \land y_1) \land v_2^2) \land (\text{out}_2 \iff u_2^0) \land (\text{out}_3 \iff u_2^1) \land \neg\text{out}_0 \land \text{out}_1 \land \text{out}_2 \land \neg\text{out}_3)
\]

• Read off the factorization 6 = 2 \times 3 from a refuting assignment
Conclusion

- A wide variety of practical problems can usefully be encoded in SAT
  - There is intense interest in efficient algorithms for SAT
  - Many of the most successful SAT-solving systems are still based on refinements of an ancient Davis-Putnam procedure
    - Non-chronological backjumping, learning conflict clauses
    - Optimization of the basic ‘constraint propagation’ rules (“watched literals” etc.)
    - Good heuristics for picking ‘split’ variables, and even restarting with different split sequence
    - Highly efficient data structures
Interactive theorem proving

In practice, most interesting theorem proving problems can’t be automated completely
- They don’t fall in a practical decidable subset
- Pure first order proof search is not a feasible approach with, e.g., set theory
- In practice, we need an interactive arrangement, where the user and machine work together
  - The user can delegate simple subtasks to pure first order proof search or one of the decidable subsets
  - However, at the high level, the user must guide the prover
John Harrison: Theorem Provers

- Interesting category of interactive provers
  - LCF (Milner et al) — Programmable proof checker for Scott’s Logic of Computable Functions written in new functional language ML
    - Implemented in a strongly-typed functional programming language
    - The \texttt{thm} (‘theorem’) is an abstract data type with only simple primitive inference rules
    - The implementation language is available for arbitrary extensions of the prover
Ed Brinksma: Testing

• Problems of testing
  • Testing is ...
    - important
    - much practiced
    - 30 % – 50 % of project effort
    - expensive
    - time critical
    - not constructive (but sadistic?)

• But also ...
  - ad-hoc, manual, error-prone
  - limited theory / research
  - little attention in curricula
  - not cool
    - “if you’re a bad programmer you might be a tester”
Ed Brinksma: Testing

- Attitude is changing
  - More awareness
  - More professional
  - Improvements possible with formal methods!

- But isn't the formal verification of the specification a better way?
  - Incomparable!
  - Only testing deals with the real implementation
  - Only testing deals also with the real environment
Ed Brinksma: Testing

TYPES OF TESTING

- Level
- System
- Integration
- Unit
- Accessibility

- Robustness
- Performance
- Usability
- Reliability
- Glass box
- Black box
- Functional behaviour
- Usability

NATO Summer School, Marktoberdorf
August 2010
Ed Brinksma: Testing

- Testing with formal methods
  - Testing with respect to a formal specification
  - Precise, formal definition of correctness
    - Good and unambiguous basis for testing
  - Formal validation of tests
  - Algorithmic derivation of tests
    - Tools for automatic test generation
    - Allows to define measures expressing coverage and quality of testing
Ed Brinksma: Testing

- Challenges of testing theory
  - Infinity of testing
    - too many possible input combinations (infinite breadth)
    - too many possible input sequences (infinite depth)
    - too many invalid and unexpected inputs
  - Exhaustive testing never possible
    - when to stop testing?
    - how to invent effective and efficient test cases with high probability of detecting errors?
  - Optimization problem of testing yield vs. effort
    - usually stop when time is over
Ed Brinksma: Testing

FORMAL TESTING

specification $S$

**correctness criterion**

implementation relation imp

implementation $\hat{I}$

test generation

test suite $\mathcal{T}$

test execution

pass / fail

NATO Summer School, Marktoberdorf
August 2010

Martin Děcký
D3S Seminar
19th October 2010
Some test generation tools (for I/O systems)

- TVEDA (CNET – France Telecom)
  - derives TTCN tests from single process SDL specification
  - developed from practical experiences
- TGV (IRISA – Rennes)
  - derives tests in TTCN from LOTOS or SDL
  - uses test purposes to guide test derivation
- TestComposer (Verilog)
  - Combination of TVEDA and TGV in ObjectGeode
- TestGen (Stirling)
  - Test generation for hardware validation
- TorX (University of Twente, ESI)
John Rushby: The $10^{-9}$ req.

• Suppose 1000 airplanes of one type
  • Each flies 3000 hours per year over a lifetime of 33 years (that's $10^8$ hours)

• Suppose 10 software-based systems on board with potentially catastrophic failure conditions
  • Then budget for each is a failure rate of $10^{-9}$ per hour, sustained for 15 hours (length of flight)
  • That's where the well-known numbers come from
    - catastrophic: $10^{-9}$ per hour
    - severe major: $10^{-7}$ per hour
    - severe: $10^{-5}$ per hour
John Rushby: Achieving $10^{-9}$

- Hardware is subject to random failures at about $10^{-6}$ per hour
  - Even worse at 35000 feet (SEU due to cosmic rays)
    - Getting worse as transistors get smaller
- Demand for fault-tolerant design
  - Fault tolerance is hard
    - Increasing complexity
    - Intuitions of engineers from traditional disciplines (continuous math) are counterproductive, lead to failure-prone homespun designs
    - Most failures in flight software are due to faults in fault tolerance
John Rushby: SW Reliability

- Software contributes to system failures through faults in its requirements, design, implementation – **bugs**
  - A bug that leads to failure is **certain** to do so whenever it is encountered in similar circumstances
    - There's nothing probabilistic about it
    - Aaah, but the circumstances of the system are a **stochastic process**
      - So there is a probability of encountering the circumstances that activate the bug
      - Hence, probabilistic statements about software reliability or failure are **perfectly reasonable**
John Rushby: Assurance for 10^{-9}

• How can we demonstrate that software (or any complex discrete system) has failure rates around 10^{-9}?
  • Down to about 10^{-4}, it is feasible to measure software reliability by statistically valid random testing
    – But 10^{-9} would need 114000 years on test
  • What we actually do is a lot of Verification and Validation (V&V)
    – Good development processes, plenty of reviews, etc.
John Rushby: Assurance for 10^{-9}

- What V&V and how much is specified by standards and guidelines
  - 57 V&V “objectives” at DO-178B Level C (10^{-5})
  - 65 V&V “objectives” at DO-178B Level B (10^{-7})
  - 66 V&V “objectives” at DO-178B Level A (10^{-9})

- How does the amount of V&V (a static global concept) connect to reliability (a dynamic execution concept)?
John Rushby: Uncertainty

• **Aleatory** (irreducible) uncertainty
  • *Uncertainty in the world*
    - E. g. impossibility to predict exactly how many heads will occur in 100 trials of tossing a coin
    - Frequentist interpretation of probability needed
  
• **Epistemic** (reducible) uncertainty
  • *Uncertainty about the world*
    - E. g. If I give you the coin, you will not know the probability of tossing a head
    - But you can estimate it, try to improve your estimate by doing experiments, learning something about its manufacture, the historical record of similar coins, etc.
In much scientific modeling:

- **Aleatory** uncertainty is captured conditionally in a model with parameters.
- **Epistemic** uncertainty centers upon the values of these parameters.

**Definition: Perfect software**

- Software that will never experience a failure in operation, no matter how much operational exposure it has.
  - Correctness relative to the critical claims.
John Rushby: Uncertainty (3)

- Possibly perfect software
  - You might not believe a given piece of software is perfect
  - But you might concede it has a possibility of being perfect
    - So we can speak of a probability of perfection
      - Note: There is also a frequentist interpretation
    - Hypothesis: The more V&V it has had, the greater that possibility
• How probability of perfection relates to reliability?
  - The rule of total probability
    • \( P(\text{SW fails} \mid \text{on a randomly selected demand}) = P(\text{SW fails} \mid \text{SW is perfect}) \times P(\text{SW is perfect}) + P(\text{SW fails} \mid \text{SW is imperfect}) \times P(\text{SW is imperfect}) \)
    • The first term is zero, because the software does not fail if it is perfect
    • Hence, define
      - \( p_{np} \) .. probability the software is imperfect
      - \( p_{fnp} \) .. probability that software fails if it is imperfect
    • Then
      \[ P(\text{SW fails}) \leq p_{fnp} \times p_{np} \]
\[ P(\text{SW fails}) \leq p_{fnp} \times p_{np} \]

- To apply this result, we need to assess values for \( p_{fnp} \) and \( p_{np} \)
- These are most likely subjective probabilities (i.e., degrees of belief)
- Let's make the most conservative approximations
  - Assume software always fails if it is imperfect (i.e., \( p_{fnp} = 1 \))
  - Then, very crudely, and very conservatively
    \[ P(\text{SW fails}) \leq P(\text{SW is imperfect}) \]
John Rushby: Perfect SW (3)

• Alternatively
  - Assume software is imperfect (i.e., $p_{np} = 1$)
    • This is the conventional assumption
    • Estimate of $p_{fnp}$ is then taken as system failure rate
    • Any value $p_{np} < 1$ would improve this
      - Even the most crude V&V always improves $P$(SW fails)
  • Littlewood and Povyakalo show that if we have
    • $p_{np} < a$ with doubt $A$ (i.e., confidence $1 - A$)
    • $p_{fnp} < b$ with doubt $B$ (i.e., $P(p_{fnp} < b) > 1 - B$)
  • Then system failure rate is
    less than $a \times b$ with doubt $A + B$
Suppose our goal is $p_{np}$ of $10^{-4}$

- Bulk of this “budget” should be divided between incorrect formalization of the specification and incompleteness of the formal analysis
  - Small fraction allocated to unsoundness of verification system

- Through sufficiently careful and comprehensive formal challenges, it is plausible an assessor can assign a subjective posterior probability of imperfection on the order of $10^{-4}$ to the formal statements on which a formal verification depends
Through testing and other scrutiny, a similar figure can be assigned to the probability of imperfection due to discontinuities and incompleteness in the formal analysis.

By use of a verification system with a trusted or verified kernel, or trusted, verified, or diverse checkers, assessor can assign probability of $10^{-5}$ or smaller that the theorem prover incorrectly verified the theorems that attest to perfection.
John Rushby: Application (3)

- Probability of perfection
  - A radical and valuable idea
  - Provides the *bridge* between correctness-based verification activities and probabilistic claims needed at the system level
  - Relieves formal verification, and its tools, of the burden of absolute perfection
Connie Heitmeyer: Specification

- Of the Major Components of Systems, Software Is the Most Problematic

DOD ... spends about 40% of its Research, Development, Test, and Evaluation budget on software—$21B for fiscal year 2003 ... DOD and industry experience indicates that about $8B (40 percent) of that amount may be spent on reworking software because of quality-related issues.

Details Emerge On Army’s Failed NLOS-LS Missile
In testimony before lawmakers yesterday, David Duma... detailed failings of the Army Non-Line of Sight Launch System (NLOS-LS). During most recent tests in February, new navigation software caused six of seven total system aborts.

Defense Tech, April 16, 2010

A U.S. soldier in Afghanistan used a Precision Lightweight GPS Receiver to set coordinates for an air strike. Seeing that the “battery low” warning light was on, he changed the battery, then pressed “Fire.” The device was designed, on starting or resuming operation after a battery change, to initialize the coordinate variables to its own location...
The soldier and three comrades were killed in the incident.
Connie Heitmeyer: Specification

- Of the Major Components of Systems, Software Is the Most Problematic
  - The majority of software errors are introduced early in development phase (close to specification)
  - The later a software error is detected the more costly it is to correct the error
Connie Heitmeyer: Behavior

NAT: All possible behaviors satisfying natural laws, constraints on the system env
REQ: All acceptable system behaviors
SOFT: All acceptable software behaviors
The example control system contains the following sets:

Set of monitored variables: \{Block, Reset, WaterPres\}
Set of controlled variables: \{SafetyInjection\}
Set of terms: \{Overridden\}
Set of mode classes: \{Pressure\}

Type definitions associated with these sets are

\[
\begin{align*}
TY(\text{WaterPres}) & = \{1, 2, \ldots, 2000\} \\
TY(\text{SafetyInjection}) & = \{\text{On, Off}\} \\
TY(\text{Block}) & = TY(\text{Reset}) = \{\text{On, Off}\} \\
TY(\text{Overridden}) & = \{\text{true, false}\} \\
TY(\text{Pressure}) & = \{\text{TooLow, Permitted, High}\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>variable name</th>
<th>WaterPres</th>
<th>Block</th>
<th>Reset</th>
<th>Pressure</th>
<th>Overridden</th>
<th>SafetyInjection</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable value</td>
<td>850</td>
<td>Off</td>
<td>On</td>
<td>TooLow</td>
<td>false</td>
<td>Off</td>
</tr>
</tbody>
</table>
### Connie Heitmeyer: Tables

<table>
<thead>
<tr>
<th>Mode</th>
<th>Pressure</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>High, Permitted</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>TooLow</td>
<td>Override</td>
<td>NOT Override</td>
</tr>
<tr>
<td>SafetyInjection</td>
<td>Off</td>
<td>On</td>
</tr>
</tbody>
</table>

Based on the new state dependencies set $D_n = \{\text{Pressure, Overridden}\}$ and the above condition table, the function $F_6$ defining the value of the controlled variable $r_6 = \text{SafetyInjection}$ is defined by

\[
\text{SafetyInjection} =
\begin{cases}
\text{Off} & \text{if } \text{Pressure} = \text{High} \lor \text{Pressure} = \text{Permitted} \lor \\
\text{On} & \text{if } \text{Pressure} = \text{TooLow} \land \text{Overridden} = \text{true} \lor \\
\end{cases}
\]

The table defines \text{SafetyInjection} as a function of a single state.
Each **condition table** describes the value of a controlled variable or term $r_i$ as a relation $\rho_i$ on modes, conditions, and values:

$$\rho_i = \{(m_j, c_{j,k}, v_k) \in M_{\mu(i)} \times C_i \times TY(r_i)\}.$$  

The relation $\rho_i$ must satisfy the following properties:

1. The $m_j$ are unique; the $v_k$ are unique.
2. $\bigcup_{j=1}^{n} m_j = TY(\mu(i))$ (All modes in the associated mode class are included).
3. For all $j$: $\forall_{k=1}^{p} c_{j,k} = true$ (**Coverage**: The disjunction of the conditions in each row of the table is true).
4. For all $j, k, l$, $k \neq l$: $c_{j,k} \land c_{j,l} = false$ (**Disjointness**: The pairwise conjunction of the conditions in each row of the table is always false).

**These four properties guarantee that the function is total**
Connie Heitmeyer: Tables (3)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Pressure</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Never</td>
<td>@F(Pressure = High)</td>
</tr>
<tr>
<td>TooLow, Permitted</td>
<td>@T(Block = On)</td>
<td>@T(Pressure = High) OR</td>
</tr>
<tr>
<td></td>
<td>WHEN Reset = Off</td>
<td>@T(Reset = On)</td>
</tr>
<tr>
<td>Overridden' =</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

Based on the above event table and the new state and old state dependencies sets, \{Block, Reset, Pressure, Overridden\} and \{Block, Reset Pressure\}, the function defining the value of Overridden, denoted \(F_5\), is described by

\[
\text{Overridden}' = F_5(Pressure, Block, Reset, Overridden, Pressure', Block', Reset') =
\]

\[
\begin{cases} 
\text{true} & \text{if } (\text{Block}' = \text{On} \land \text{Block} = \text{Off} \land \text{Pressure} = \text{TooLow} \land \text{Reset} = \text{Off}) \lor (\text{Block}' = \text{On} \land \text{Block} = \text{Off} \land \text{Pressure} = \text{Permitted} \land \text{Reset} = \text{Off}) \\
\text{false} & \text{if } (\text{Reset}' = \text{On} \land \text{Reset} = \text{Off} \land \text{Pressure} = \text{TooLow}) \lor (\text{Reset}' = \text{On} \land \text{Reset} = \text{Off} \land \text{Pressure} = \text{Permitted}) \lor (\text{Pressure}' = \text{High} \land \text{Pressure} \neq \text{High}) \lor (\text{Pressure}' = \text{Permitted} \lor \text{Pressure}' = \text{TooLow}) \\
\text{no change} & \text{if } \neg(\text{Pressure} = \text{Permitted} \lor \text{Pressure} = \text{TooLow}) \\
\text{Overridden otherwise} & 
\end{cases}
\]

Defines Overridden as a function of two states
Connie Heitmeyer: Tables (4)

<table>
<thead>
<tr>
<th>Old Mode</th>
<th>Event</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TooLow</td>
<td>@T(WaterPres ≥ Low)</td>
<td>Permitted</td>
</tr>
<tr>
<td>Permitted</td>
<td>@T(WaterPres ≥ Permit)</td>
<td>High</td>
</tr>
<tr>
<td>Permitted</td>
<td>@T(WaterPres &lt; Low)</td>
<td>TooLow</td>
</tr>
<tr>
<td>High</td>
<td>@T(WaterPres &lt; Permit)</td>
<td>Permitted</td>
</tr>
</tbody>
</table>

Based on the above mode transition table and the old and new dependencies sets 
\{(WaterPres, Pressure)\} and \{WaterPres\}, the function defining the value of
Pressure, denoted \(F_4\), is described by

No transitions possible from TooLow to High and vice versa

\[
\begin{align*}
\text{TooLow} & \quad \text{if} \quad \text{Pressure} = \text{Permitted} \land \text{WaterPres'} < \text{Low} \land \\
\text{High} & \quad \text{if} \quad \text{Pressure} = \text{Permitted} \land \text{WaterPres'} \geq \text{Permit} \land \\
\text{Permitted} & \quad \text{if} \quad \begin{cases} 
\text{Pressure} = \text{TooLow} \land \text{WaterPres'} \geq \text{Low} \land \\
\text{WaterPres} \neq \text{Low} \\
\text{Pressure} = \text{High} \land \text{WaterPres'} < \text{Permit} \land \\
\text{WaterPres} \neq \text{Permit}
\end{cases} \\
\text{Pressure} & \quad \text{otherwise.}
\end{align*}
\]

\[
\text{NAT: Pressure} = \text{TooLow} \Rightarrow \text{Pressure'} \in \{\text{TooLow, Permitted}\} \land ...
\]
Advantages of a tabular notation

- **Less error-prone** than, e.g., logic notation
  - Structure provided by tables eliminates whole classes of errors
- **More scalable** than many other notations
  - For example, graphic notations, such as finite state diagrams, do not scale well to practical applications
    » The labels on the transitions are often too long
    » Not practical when the number of states is large
Connie Heitmeyer: Practicality

**SPECIFY THE SYSTEM PRECISELY**

Use a **TABULAR notation** with an explicit formal semantics to specify the required behavior.

**APPLY “CONSISTENCY CHECKING”**

Automatically check spec for syntax/type errors, missing cases, nondeterminism, circular defs, etc.

**SIMULATE THE SYSTEM BEHAVIOR**

Symbolically execute the system based on the (executable) req. specs.

**VERIFY SPECS USING**

- **MODEL CHECKING**
  - Check critical application properties

- **THEOREM PROVING**
  - Verify specs using theorem proving

- **INCREASING EFFORT, INCREASED EXPERTISE**

- **Scalable tabular notation**
- **Integrated set of software tools**
  - light-weight tools (easy to use)
  - heavy-duty tools (e.g., theorem prover)

As we move down the chain, we increase assurance in the spec.
Connie Heitmeyer: Validation

Simulator Display

Simulator Log

System State

Next Event

“Executed” Events

Monitored Vars  Dependent Vars
Connie Heitmeyer: Validation (2)