The Quest for Correctness - Beyond Verification

Turing Lecture
Embedded Systems Week 2008
Atlanta, October 20, 2008
Correctness by checking vs. Correctness by construction

Building systems which are correct with respect to given requirements is the main challenge for all engineering disciplines.

Correctness can be achieved:

- Either by checking that a system or a model of a system meets given requirements.
- Or by construction by using results such as algorithms, protocols, architectures e.g. token ring protocol, time triggered architecture.

A big difference between Computing Systems Engineering and disciplines based on Physics is the importance of a posteriori verification for achieving correctness.
- Current status
- Work directions
- Conclusion
Approaches for checking correctness

Checking correctness

Physical prototypes
  e.g. testing

Ad hoc models
  e.g. SystemC simulation

Models
  (Virtual SW Prototypes)

Formal models
  (well-defined notion of state and transition)

Verification
  - Algorithmic Verification e.g. Model checking
  - Deductive Verification
Verification: Three essential ingredients

- **Requirements**
  describing the expected behavior, usually as a set of properties

- **Models**
  describing a transition relation on the system states

- **Methods**
  for checking that the models satisfy the requirements
### Requirements specification (1/3)

**State-based**

Using a machine (monitor) to specify observable behavior

- *send*
- *receive*

Good for characterizing causal dependencies e.g. sequences of actions

**Property-based**

Using formulas, in particular *temporal logic*, to characterize a set of execution structures e.g. traces, execution trees

- `always( inev ( enable( send ) ) )`
- `always( inev ( enable( receive ) ) )`

Good for expressing global properties such as mutual exclusion, termination, fairness
About Temporal logic [Pnueli, Lamport, Clarke & Emerson]

This was a breakthrough in understanding and formalizing requirements for concurrent systems. Writing rigorous specifications in temporal logic is not trivial.

- There exist subtle differences in the formulation of common concepts such as liveness and fairness depending on the underlying time model e.g. always( inevitable( f ) )

- The declarative and dense style in the expression of property-based requirements is not always easy to master and understand. Are specifications
  - Sound: there exists a model satisfying it
  - Complete: tight characterization of system behavior

Pragmatically, we need a combination of both property-based and state-based styles, e.g. PSL
Moving towards a “less declarative” style by using notations such as MSC’s or interface automata

Much to be done for extra-functional requirements characterizing:
- security (e.g. privacy properties),
- reconfigurability (e.g. non interference of features),
- quality of service (e.g. jitter).
Models should be:

- **faithful** e.g. whatever property we verify for the model holds for the real system
- generated **automatically** from system descriptions

For hardware, it is easy to get faithful logical finite state models represented as systems of boolean equations.
For software this may be much harder ....
For mixed Software / Hardware systems:

- there are no faithful modeling techniques as we have a poor understanding of how software and the underlying platform interact
- validation by testing physical prototypes or by simulation of ad hoc models
Algorithmic Verification: Using Abstraction (1/2)

\( S_A \) satisfies \( f_A \) implies \( S \) satisfies \( f \)

where \( S_A = (Q_A, R_A) \) is an abstraction of \( S = (Q, R) \)

for formulas \( f \) involving only universal quantification over execution paths

[Cousot&Cousot 79] **Abstract interpretation**, a general framework for computing abstractions based on the use of Galois connections

\[ \alpha F \gamma \]

\( \alpha, \gamma \) are monotonic
\( \text{Id} \subseteq \gamma \alpha \)
\( \alpha \gamma \subseteq \text{Id} \)

\( \alpha F \gamma \) is the best approximation of \( F \) in the abstract state space
Initially, focused on finite state systems (hardware, control intensive reactive systems)
Later, it addressed verification of infinite state systems by using abstractions
Used to check general properties specified by temporal logics

Driven by the concern for finding adequate abstract domains for efficient verification of program properties, in particular runtime errors
Focuses on forward or backward reachability analysis for specific abstract domains

**Significant results can still be obtained by combining these two approaches**
e.g. by using libraries of abstract domains in model checking algorithms.
• Current status

• Work directions

• Conclusion
Work directions: Component-based modeling

Develop theory and methods for building faithful models for mixed SW/HW systems as the composition of heterogeneous components.

Sources of heterogeneity
- **Abstraction levels**: hardware, execution platform, application software
- **Execution**: synchronous and asynchronous components
- **Interaction**: function call, broadcast, shared memory, message passing etc.

We need to move

<table>
<thead>
<tr>
<th>from</th>
<th>low level automata-based composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>a unified composition paradigm encompassing architecture constraints such as protocols, schedulers, buses.</td>
</tr>
</tbody>
</table>
Work directions: Compositional verification

Proving properties of a composite component from properties of
- individual components
- its architecture
**Work directions: Compositional verification**

Proving properties of a composite component from properties of
- individual components
- its architecture

---

**We need to move**

<table>
<thead>
<tr>
<th>Composition operation</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>to</td>
</tr>
<tr>
<td>Automata-based</td>
<td>Safety, liveness</td>
</tr>
<tr>
<td>Component-based</td>
<td>Specific properties e.g. Deadlock-freedom, mutex</td>
</tr>
</tbody>
</table>
Develop compositionality results

- For particular
  - architectures (e.g. client-server, star-like, time triggered)
  - programming models (e.g. synchronous, data-flow)
  - execution models (e.g. event triggered preemptable tasks)

- For specific classes of properties such as deadlock-freedom, mutual exclusion, timeliness

Compositionality rules and combinations of them lead

- to “verifiability” conditions, that is conditions under which verification of a particular property becomes much easier.
- to classes of systems which are correct-by-construction
Checking **global deadlock-freedom** of a system built from deadlock-free components, by separately analyzing the components and the architecture.

Potential deadlock:

\[ D = \text{en}(p1) \land \neg \text{en}(p2) \land \text{en}(q2) \land \neg \text{en}(q1) \]

Potential deadlock:

\[ D = \text{en}(p1) \land \neg \text{en}(p2) \land \text{en}(q2) \land \neg \text{en}(q3) \land \text{en}(r3) \land \neg \text{en}(r1) \]
Work directions: Compositionality - example

Eliminate potential deadlocks $D$ by computing compositionally global invariants $I$ such that $I \land D = false$

<table>
<thead>
<tr>
<th>Example</th>
<th>Number of Comp</th>
<th>Number of Ctrl States</th>
<th>Number of Bool Variables</th>
<th>Number of Int Var</th>
<th>Number Potential Deadlocks</th>
<th>Number Remaining Deadlocks</th>
<th>Verification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature Control (2 rods)</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>3s</td>
</tr>
<tr>
<td>Temperature Control (4 rods)</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>5</td>
<td>32</td>
<td>15</td>
<td>6s</td>
</tr>
<tr>
<td>UTOPAR (40 cars, 256 CU)</td>
<td>297</td>
<td>795</td>
<td>40</td>
<td>242</td>
<td>??</td>
<td>0</td>
<td>3m46s</td>
</tr>
<tr>
<td>UTOPAR (60 cars, 625 CU)</td>
<td>686</td>
<td>1673</td>
<td>60</td>
<td>362</td>
<td>??</td>
<td>0</td>
<td>25m29s</td>
</tr>
<tr>
<td>R/W (2000 readers)</td>
<td>2002</td>
<td>4006</td>
<td>0</td>
<td>1</td>
<td>??</td>
<td>0</td>
<td>4m46s</td>
</tr>
<tr>
<td>R/W (3000 readers)</td>
<td>3002</td>
<td>6006</td>
<td>0</td>
<td>1</td>
<td>??</td>
<td>0</td>
<td>10m9s</td>
</tr>
</tbody>
</table>

Results obtained by using the D-Finder tool: http://www-verimag.imag.fr/~thnguyen/tool/
- Current status
- Work directions
- Conclusion
A posteriori verification is not the only way for guaranteeing correctness.

- In contrast to Physics, Computer Science deals with a potentially infinite number of created universes
- Limiting the focus on particular tractable universes of systems can help overcome current limitations

We should concentrate on compositional modeling and verification for sub-classes of systems and properties which are operationally relevant and technically successful

This vision can contribute to the unification of the discipline, by bridging the gap between Formal Methods and Verification, and Algorithms and Complexity.
Thank You