Formal Verification using the IITD-Concurrency WorkBench

S. Arun-Kumar
sak(a)cse.iitd.ernet.in
Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110016.
http://www.cse.iitd.ac.in/ sak

September 26, 2005

Overview

1. History & Lineage
2. Design Languages
3. Capabilities of the CWB-NC
4. Why use the IITD-CWB?
5. Behavioural Verification
6. Logical Properties
7. Conclusion

History & Lineage

Concurrency Workbenches

1. The Edinburgh Concurrency Workbench (old)
2. The Concurrency Workbench of North Carolina
3. The Concurrency Workbench of the New Century
4. The IITD Concurrency Workbench

The Edinburgh Concurrency Workbench (old)

1. Designed by Cleaveland, Parrow and Steffen
2. Implemented in Standard ML '90
3. Used CCS as the only design language
4. Implemented behavioural verification
5. Implemented Hennessy-Milner Logic

The Concurrency Workbench of North Carolina

1. Complete redesign by Rance Cleaveland
2. Used the new features of a very much changed SML
3. Expanded to include the μ-calculus as the basic underlying logic
4. Modularized to allow for different kinds of design languages
5. Used a separate back-end generator for new design languages
6. Designed a sister software PAC-NC for the purpose of back-end generation

The Process Algebra Compiler of North Carolina

1. Designed by Steve Sims under Rance Cleaveland's supervision
2. A compiler generator for the CWB-NC
3. Allowed a syntactic specification of both syntax and semantics of design languages
4. Used ML-Lex and ML-YACC to generate the back-end for the CWB-NC
5. Allowed for experimenting with new languages and verification
6. Tested out on CCS, CSP, Basic LOTOS etc.

Semantics in PAC-NC

1. Designed a syntax for formal specification of operational semantics
2. semantics specified through labelled transition systems
3. Used ML-Lex and ML-Yacc to scan and parse semantic specification
4. Generated the back-end for CWB-NC as LTS generation
5. Allowed for the design of special libraries to be included in the back-end
6. Greater flexibility and scope for customization as opposed to native languages of other model-checkers (e.g. SPIN, SMV, Nu-SMV).

The Concurrency Workbench of the New Century

1. An enhancement of the North Carolina version
2. Includes GCTL* and SCCS as new interfaces
3. Remains compatible with the PAC-NC.
4. Allows for installation on win32 platforms
5. Comes as an RPM for Red Hat Linux installations

The IITD Concurrency Workbench

1. Faithful to the CWB-New Century version 1.2
2. Includes a web-interface for all basic model-checking needs
3. Includes an ACSR interface
4. Interfaces for GCTL* not currently available
5. Interface for the simulator not available

Why use the IITD-CWB?

1. No installation required
2. Minimal learning required
3. Available for free public use by anyone who simply wants to try it out

Why use the IITD-CWB 1?

1. No installation required
   - Installation of the CWB-NC first requires installing SML version 110.?.?
2. Minimal learning required
3. Available for free public use by anyone who simply wants to try it out

Why use the IITD-CWB 2?

1. No installation required
2. Minimal learning required
   - 1 web-page of instructions as opposed to a 100-page manual
   - The GUI of the CWB-NC is somewhat unreliable and does not easily customize. The IITD-CWB simply uses the well-known features of the browser.
   - Allows occasional and professional user to by-pass the crummy command line interface of the CWB-NC
   - Easy to do cut-and-paste submission as well as uploading of large files.
3. Available for free public use by anyone who simply wants to try it out

Why use the IITD-CWB 3?

1. No installation required
2. Minimal learning required
3. Available for free public use by anyone who simply wants to try it out
    - once,
    - twice or
    - ... any number of times if you begin to like it!

Design Languages

- Most model-checkers have a native language the user must use to write system specification.
- Design languages and the user interface form the core learning process for a lay user of the system
- The currently supported design languages in the IITD-CWB
  - CCS, CSP, Basic LOTOS, SCCS, PCCS, TCCS and ACSR (not in CWB-NC)
- The currently supported logics for model-checking (also in CWB-NC)
  - HML, μ-calculus, CTL, GCTL*

Capabilities of the CWB-NC

- Model-checking: Given a design SYS written in a supported design language and a property Prop written in a supported logic, answers the question

  $$\boxed{\textit{Does } \texttt{SYS} \textit{ satisfy } \texttt{Prop}\textit{?}}$$

  Also provides a counterexample if answer is “NO!”.

- Verification: Given a specification SPEC and an implementation IMP both written in the same design language, answers the question?

  $$\boxed{\textit{Do } \texttt{SPEC} \textit{ and } \texttt{IMP} \textit{ have the same observable behaviour?}}$$

  Also provides counterexample, if the answer is “NO!”

- Simulator: Given a design SYS step through its execution interactively.

Behavioural Verification: Equivalences

Given a SPEC and an IMP

- Is SPEC strongly equivalent to IMP?
- Is SPEC observably equivalent to IMP?
- Is SPEC observably congruent to IMP?
- Is SPEC trace equivalent to IMP?
- Is SPEC testing equivalent to IMP?

Behavioural Verification: Preorders

Given a SPEC and an IMP

- Is IMP MAY contained in SPEC?
- Is IMP MUST contained in SPEC?
- Is SPEC MAY contained in IMP?
- Is SPEC MUST contained in IMP?

If the answers to all the above questions are “YES”, then they are both testing equivalent.

Equivalence checking: XOR3 Spec

Consider an XOR of 3 inputs. What should be the specification?

```
             _____________
            |             |
 a  --->o                 |
            |             |
 b  --->o     XOR_3       o----> add
            |             |
 c  --->o                 |
            |_____________|
```

Equivalence checking: XOR3 Spec

Specification in CCS.

```
proc XOR_3      = a0.(b0.(c0.'add0.nil +
                          c1.'add1.nil
                         ) +
                      b1.(c0.'add1.nil +
                          c1.'add0.nil
                         )
                     ) +
                  a1.(b0.(c0.'add1.nil +
                          c1.'add0.nil
                         ) +
                      b1.(c0.'add0.nil +
                          c1.'add1.nil
                         )
                     )
```

Equivalence checking: XOR3 Imp

Construct an XOR3 of 3 inputs with two XOR gates (of 2 inputs each).

```
       ______________________________
      |________                      |
      |        |       XOR_XOR_2     |
a --->o        |                     |
      |  XOR1  o_____ i              |
b --->o        |     |        _______|
      |________|     |       |       |
      |              |------>o       |
      |                      |  XOR2 o-----> add
c ----o--------------------->o       |
      |                      |_______|
      |______________________________|
```

Equivalence checking: XOR3 Imp

Begin with a specification of an XOR gate with 2 inputs.

```
proc XOR   = a0.(b0.'out0.nil + b1.'out1.nil) +
             a1.(b0.'out1.nil + b1.'out0.nil)
```

Get 2 copies of this gate (appropriately renamed).
Here's the first one.

```
             ____________
            |            |
    a --->o              |
            |   XOR_1    o----> i
            |            |
    b --->o              |
            |____________|
```

```
proc XOR_1 = XOR [i0/out0, i1/out1]
```

Equivalence checking: XOR3 Imp

And here's the second.

```
             ____________
            |            |
      i --->o            |
            |    XOR_2   o----> add
            |            |
      c --->o            |
            |____________|
```

```
proc XOR_2 = XOR [i0/a0, i1/a1, c0/b0, c1/b1,
                  add0/out0, add1/out1]
```

Equivalence checking: XOR3 Imp

Now connect them up by matching names

```
         ______________________________
        |________                      |
        |        |       XOR_XOR_2     |
  a --->o        |                     |
        |  XOR1  o_____ i              |
  b --->o        |     |        _______|
        |________|     |       |       |
        |              |------>o       |
        |                      |  XOR2 o-----> add
  c ----o---------------------->o       |
        |                      |_______|
        |______________________________|
```

```
proc XOR_XOR_2 = (XOR_1 | XOR_2) \ {i0, i1}
```

And package it in a box so nobody from outside can see what you have done.

Equivalence checking: XOR3 Imp

- Are `XOR_3` and `XOR_XOR_2` observably equivalent?
- Are `XOR_3` and `XOR_XOR_2` observably congruent?

Equivalence checking: Other examples

- Systolic systems: Palindromes
- Protocols: Alternating Bit

Logical Properties

- Modal μ-calculus
- Computation tree logic
- Generalized Hennessy Milner Logic with recursion (GCTL*)
  - CTL* enhanced with atomic propositions for LTSs
  - Uses **R**elease as a dual for **U**ntil
  - Works on LTSs rather than Kripke systems

Logical Properties: Examples

$\texttt{<act\_set>}\psi = \texttt{E (\{ act\_set \} /\textbackslash{} X } \psi\texttt{)}$

$\texttt{[act\_set]}\psi = \texttt{A (\{ act\_set \} -> X } \psi\texttt{)}$

```
prop can_deadlock = E F ~{- }

prop recv_guarantee = A G ({send} -> F {'receive})

prop fair_recv_guarantee =
 A ((G F {-t}) -> (G {send} -> F {'receive}))
```

NWCV-DAIICT

Home Page

Title Page

Page 28 of 30

Go Back

Full Screen

Close

Quit

Conclusions

- Can we combine model-checking with an inductive theorem-proving mechanism? Then proof of hardware designs and their extensions, e.g. 32 bit to 64 bit or bit-sliced designs could be proven more easily by a theorem-prover which which may work on a certified model-check for the basis of the induction?
- The use of behavioural relations can reduce the burden on proofs by using (pre)congruences to justify and validate designs to ameliorate the state explosion problem.
- The main problem with logical properties is that for real life designs, often the number of properties which constitute the specification can be so large that even after you have verified whatever properties you specify, you may wonder whether your design is complete.

Conclusions

- This is because modularity is not built into the logics, whereas it is built into algebraic specifications.
- This state explosion problem reduces when both specification and implementation are in the same language and one can use behavioural notions, such as equivalences and preorders (and one can take advantage of congruences and precongruences).
- Most model-checking works only on finite-state systems. But for infinite-state systems static analysis and abstraction may be required.
- But under abstraction the full abstraction property is lost.
- So a verification of all the specified properties does not guarantee that your system is correct. But any incorrectness in your abstraction also implies that your concrete system is at least equally wrong.