Exploiting Independence for Verification, Refinement, and Modularity and: On Translating Among Models

Shmuel Katz
Computer Science Department
The Technion
Haifa, Israel

Terms of Reference

- Distributed or shared memory concurrent systems
  - Processes with separate statespaces, asynchronous
  - Threads with shared memory
  - Many interleavings/possible scenarios make such systems hard to develop, verify, modify
- Independence: asynchronous local actions can be done in either order or simultaneously

a := 5 ||| b:= 3

Semantics (I)

- Process: A trace or partial order run of a system….NOT HERE
- Process: A methodology or approach to the way systems are created, as in 'a software development process'……NOT HERE
- Process: A component in the design or implementation of a system, with its own variables and locus of control, that communicates and synchronizes with others of its ilk… YES

Terms of Reference (cont.)

- Top-down methodology and language support is needed and can use independence
- Refinement is needed that loosens synchrony
- Proof method can exploit the independence
  - Infinite state theorem proving (e.g., PVS-based)
  - Model checking (e.g., SMV, Spin, Murphi)

Top-Down Development

- Unity[Chandy/Misra]:
  - Global nondeterministic view
  - Later refine (split) to processes, channels
  - Adds functionality using superposition
  - Division to processes as part of the solution
  - Difficult to optimize per architecture
  - Has systematic verification for invariants, less so for liveness properties

Another Approach

- Processes, sites, distribution are sometimes part of the problem—exist throughout
- Have processes with thick atomic interactions among many at once—at top level
- Refinements have `thinner' interactions with less synchronization among fewer processes
- Must show that loosening synchrony preserves correctness

Multiparty Interactions, Joint Actions, Reconciliations

- Many processes `come together,' synchronize, and exchange information in a single atomic step
- Will give examples of how to declare such actions and use them in high-level design (?)
- Will refine to fewer participants in interactions, and less synchrony
- Has modularity, but cuts across processes
- Still missing convenient syntax, not yet popular

Scenario-based Specification

- Use typical, easy to understand, sequences of events (= *scenarios*) to specify the system
- The scenarios are similar to transactions in databases
- A sequence of scenarios is a computation that is easy to understand
- All computations are mixtures of given scenarios
- Can we show: `every computation reduces to some sequence of scenarios'?

Reductions as Specifications

- Database serializability
  - Transaction is a series of operations
  - Every interleaving of operations is equivalent to one done transaction-by-transaction
- Cache memory consistency
  - Reads and Writes are implemented by propagating values though queues
  - Every sequence of actions is equivalent to one where each Write (and Read) is implemented all at once (and local operations maintain their ordering)

Superimpositions and Aspects

- Another type of cross-cutting concern: adding functionality with a separately designed component that adds operations to existing systems
- Classic example: add termination detection to a system that otherwise might deadlock
- More recently: add aspects that treat fault tolerance, security, debugging, monitoring
- Can we fit this into the rest of our picture?

Convenient Executions

- Two stage verification:
  - Convenient computations (scenarios) are OK
  - All computations reduce to convenient ones
- For refinement:
  - Convenient easily refine thick atomic actions
  - Looser synchronization reduces to convenient

ABC

aaaaaabbbbbbccccccc

aaabbaabbabccbcccc

What does *Reduce* Mean?

- One view: computations differ by reordering of *independent* events
- Another: computations differ by having event sequences that have the same effect
- If reduced satisfies desired property, so does the original
- Need to justify for a family of reductions

Outline

- Proof theory of convenient computations
- Tool support for convenient computations
- Link to refinement: cache consistency
- Examples of multiparty language constructs (?)
- Link to refinement: loosening constructs (?)
- Integrating superimposition: a challenge (?)
- On translating among formal models

Proof Theory of Convenient Computations

- Computation = execution sequence = trace of states, each a result of an enabled transition of the system (given as textual transformations)
- Actually, sequence of pairs (state, transition)
- System semantics = collection of computations
- Fairness conditions can restrict the collection:

Enabled textual transformations are either applied or become unenabled

The Basic Theorem

- COMPS : set of all system computations
- CONV $\subseteq$ COMPS : the convenient ones
- R: a reduction relation among COMPS
- p : a predicate over COMPS

IF (p holds for all CONV) $\forall c \in$ CONV. p(c)

(R respects p) $\forall c,d \in$ COMPS.(R(c,d)$\wedge$ p(d))$\Rightarrow$p(c)

(R relates every comp. to a conv.)

$\forall c \in$ COMPS.$\exists$ d $\in$ CONV. R(c,d)

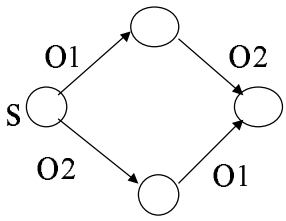
THEN (p holds for all COMPS) $\forall c \in$ COMPS. p(c)

Operations, Transitions, Events: Semantics (II)

- Operation = textual atomic transformation with an enabling condition over states
- Transition = a concrete transition taking one state to another (an arrow in transition graph), labeled by its operation name
- Event = a particular occurrence of a transition in a computation

Independence

- Operations O1 and O2 are *independent at state s (i.e., the transitions are independent)* if
  - Executing O1 from s doesn't affect O2's enabledness and vice versa
  - If both are enabled in s, they commute (*diamond* property)



Conditional Independence

- Dependence on state generalizes full independence of operations
- Full independence of two operations is easier theoretically (Mazurkiewicz trace equivalence) but less useful because it ignores too much
- Example: *put* and *get* of a queue are independent when the queue is already nonempy, but dependent for an empty queue

(so are dependent as operations)

Using Independence

- Define I $\subseteq$ Op $\times$ Op $\times$ State as a predicate defining the independence we plan to exploit
- Prove that I is conditionally independent, separately from the rest of the reasoning
- I may be easier to prove independent than the maximal relation possible, but gives less…
- Some justifications can be `structural':

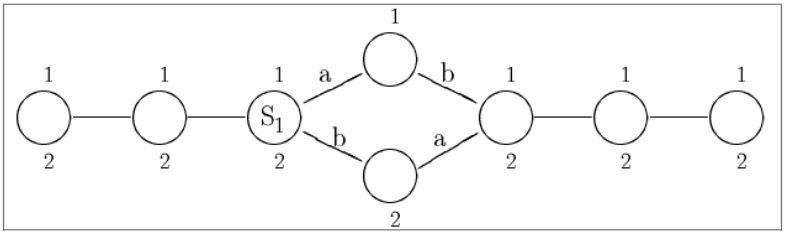
Local assignments in different processes

Queue operations on nonempty queues

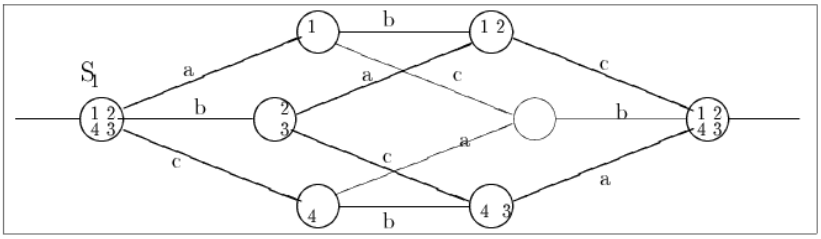
Independence as a Reduction

- Computations A and B are *one-swap equivalent* if they differ only in that one pair of adjacent independent operations appear in the opposite order
- Note: means one state can differ!
- Swap equivalent: reflexive transitive closure of one-swap equivalence (have a chain of adjacent swaps)

Examples



Computations 1,2 are one-swap-equiv. (consecutive operations a and b are independent at S_1)



Possible interleavings of operations a,b,c (independent at S_1)

How to Use the Theorem

- Show p for convenient executions

- Show that for a Family of properties F,

$\forall p \in F$, if p(d), and c is swap-equivalent to d, then p(c). (*p is respected by swap-equivalence)*

- Show that every computation is swap-equivalent to a convenient one.

All three tasks use different information and may be proven using different techniques and tools

Expressing Properties: Temporal Logic over Computations

- Gp = □p = In every future (and present) state, p
- Fp = ◊p = In some future (or present) state, p
- p U q = p is true in every future state until q
- Future fragment of linear temporal logic, over computation sequences…
- Can be encoded by referring to indices in the sequence: Fp = ∃t. p(s(t))

Properties that Are Respected By Swap-equivalence

- State properties about terminating states (where no state changes are possible), are respected by interchanging independent transitions (swap-equivalence).
  - Partial correctness, input/output relations
- For a property p known to be stable (remain true once true), Fp is respected by swap-equivalence
  - Termination, existence of deadlock

Using stability for Convenience

- If p is a stable property, then G¬p is respected by swap equivalence
- For any (nonstable) p, G ¬p may not be respected by swap equivalence:
- Can have a computation with mutual exclusion, and a swap equivalent one violating it.
- Make a temporary violation stable by recording it in the state—and then *G ¬violation* may be shown only for convenient computations, and yet is guaranteed true for all if they are swap-equivalent

More Good Properties

- For p any future path property (linear t.l. assertion with only future modalities), GFp and FGp are respected by swap-equivalence.
- Those families do not work for p with past modalities, because most states in the two computations have different histories (so a claim true in infinite states of one, might not be true in the other).

Showing Reduction to a Convenient Computation

- When a finite chain of reductions suffices, use well-founded measure function m:

If $\forall c \in (COMPS \backslash CONV).\ \exists\ d \in COMPS.$

$$R(c,d) \wedge (m(d) < m(c)),$$

Then $\forall c \in COMPS.\exists\ d \in CONV.\ R(c,d)$

(the third condition of the basic theorem)

Show each computation is convenient or reduces to one with a smaller measure…

Typical Measure Functions

- Sum of *matching intervals*: number of events between a pair of events (e.g., put and get) that are adjacent in convenient computations.
- *Layer* measure: sum of, for each event, the number of events from later layers that appear before it. (example will be seen later..)

Infinite Delays

- Method and proof environment works well for bounded-out-of-order
- Problem: sometimes can delay an action until the end of a sequence of prefixes—but in an infinite computation it is delayed forever.
- All prefixes are OK, but limit isn't…
- Need a separate justification `at the limit'

How To Prove It

- P holds for convenient: any proof will do, using strong invariants, exploiting convenience
- Reduction respects P: theorem proving for classes of properties
- All reduce to convenient:
  - Proof environment over PVS
  - Model checking revisited
- Finite prefixes are enough: bound out-of-order or prove healthiness conditions

A PVS Proof Environment (joint with Marcelo Glusman)

- Separate `the method' from any particular application; maximize infrastructure
- Provide needed definitions, proven theorems and lemmas, clear statement of proof obligations with a user `roadmap'
- Main emphasis: To show every computation reduces to a convenient one, using well-founded measure
- Hide hard parts as much as possible

Promoting Computations

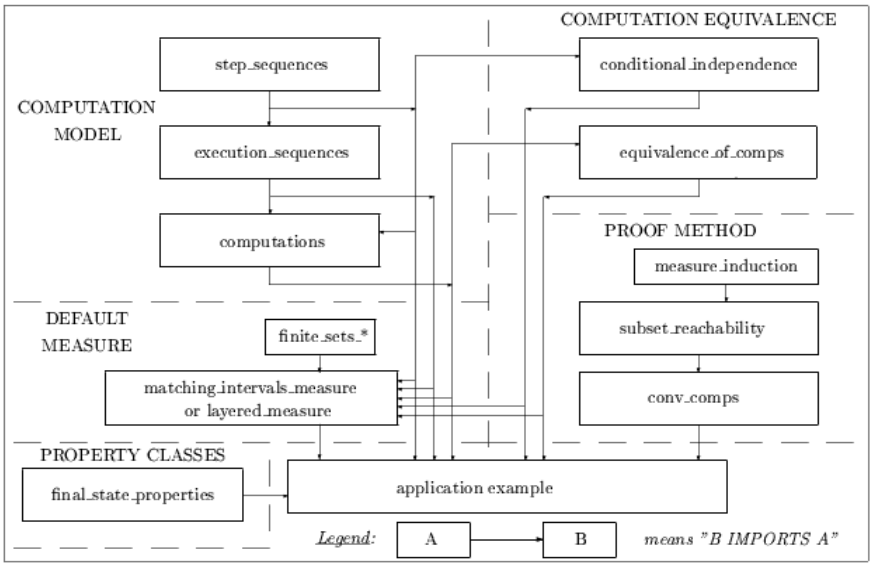
- Define generic state, computation, as first-class PVS objects over which we can quantify
- Computation = sequence of states each initial or obtained from the previous using nextstate function—actually (state, transition) pairs
- Relevant computations—include fairness or other restrictions
- Temporal logic modalities as assertions about indices in the sequences identifying key states

Pre-Proven Theorems

- Basic theorem
- Classes of properties respect swap-equivalence
- Well-founded measure theorem
- Two popular measures are appropriate
  - Matching intervals measure
  - Layers measure:

  User proves every nonconvenient has events out of layer order that can be swapped—system proves that well-founded measure theorem can be applied

Structure of the PVS Environment



The hierarchy of theories.

The User's Tasks

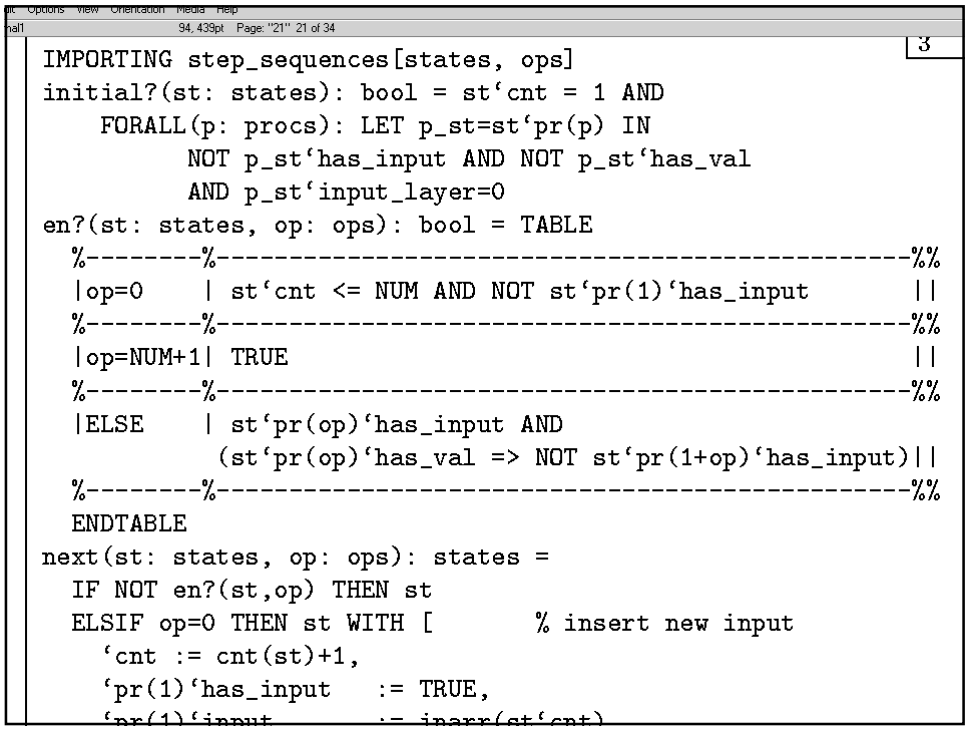
- Define the state, and transition functions
- Define additional constraints (fairness) *
- To use swap-equivalence, define I, prove it implies conditional independence *
- Define the convenient computations
- Pick a measure function and define the measure
- Prove a reduction from any nonconvenient to a computation with a smaller measure—can use sufficient conditions: any event too early by layer order can be switched to the right

Simple Example: Pipelining

- Insertion sort to a sequence of processes that each have a value, and pass on the larger of the existing value and an input:
- Typical state: (7) 2 (5) 3 ( ) 7 (9) 8
- Invariant is relatively complex
- Convenient: insert one at a time; typical state: (7) 2 3 5 7 8 9 or
  2 3 (7)5 7 8 9

```
time: THEORY
BEGIN
  time : TYPE = nat
  intervals : TYPE = [# t1: time, t2:{t:time|t>t1} #]
  distance(i: intervals): nat = i`t2 - i`t1 - 1
END time
```

```
step_sequences [states,ops: NONEMPTY_TYPE]: THEORY
BEGIN
  steps: TYPE = [# st : states, op : ops #]
IMPORTING time
  step_seqs : TYPE = [time->steps]
  initial_conditions  : TYPE = [states     -> boolean]
  enabling_conditions : TYPE = [states,ops -> bool   ]
  next_state_functions: TYPE = [states,ops -> states ]
END step_sequences
```



```
IMPORTING step_sequences[states, ops]
initial?(st: states): bool = st`cnt = 1 AND
    FORALL(p: procs): LET p_st=st`pr(p) IN
          NOT p_st`has_input AND NOT p_st`has_val
          AND p_st`input_layer=0
en?(st: states, op: ops): bool = TABLE
  %--------%-------------------------------------------------%%
  |op=0    | st`cnt <= NUM AND NOT st`pr(1)`has_input        ||
  %--------%-------------------------------------------------%%
  |op=NUM+1| TRUE                                            ||
  %--------%-------------------------------------------------%%
  |ELSE    | st`pr(op)`has_input AND
           (st`pr(op)`has_val => NOT st`pr(1+op)`has_input)||
  %--------%-------------------------------------------------%%
  ENDTABLE
next(st: states, op: ops): states =
  IF NOT en?(st,op) THEN st
  ELSIF op=0 THEN st WITH [       % insert new input
    `cnt := cnt(st)+1,
    `pr(1)`has_input   := TRUE,
    `pr(1)`input       := inarr(st`cnt)
```

4

```
ELSIF op=NUM+1 THEN st          % idling operation
ELSIF NOT st‘pr(op)‘has_val THEN st WITH [ % processor "op"
  ‘pr(op)‘has_input  := FALSE,         % stores local value
  ‘pr(op)‘has_val    := TRUE,
  ‘pr(op)‘val        := st‘pr(op)‘input ]
ELSE st WITH [
  ‘pr(op)‘has_input  := FALSE,         % store the minimum
  ‘pr(op)‘val        := IF st‘pr(op)‘input <= st‘pr(op)‘val
                        THEN st‘pr(op)‘input
                        ELSE st‘pr(op)‘val ENDIF,
  ‘pr(op+1)‘has_input := TRUE,         % forward the maximum
  ‘pr(op+1)‘input     := IF st‘pr(op)‘input <= st‘pr(op)‘val
                         THEN st‘pr(op)‘val
                         ELSE st‘pr(op)‘input ENDIF,
  ‘pr(op+1)‘input_layer := st‘pr(op)‘input_layer ]
ENDIF
```

7

```
indep?(op1,op2: ops, s: states): bool =
    op1>op2+1 OR op2>op1+1 OR op1=NUM+1 OR op2=NUM+1
```

8

```
IMPORTING conditional_independence[
            states, ops, initial?, en?, next]
JUDGEMENT indep? HAS_TYPE independence_relations
IMPORTING equivalence_of_comps[states, ops, initial?, en?,
                               next, relevant?, indep?]
```

9

```
conv?(c: comps): bool =
  FORALL(t: time): ( c(t)‘op=0 =>
    FORALL(p: procs|p <= NUM): NOT c(t)‘st‘pr(p)‘has_input )
```

Pipelining (cont.)

- Layer: a single wave inserting a new value (and passing on displaced values)
- Mixed computations have some events from lower layers after events from higher layers
- Show that whenever a transition from a higher layer is followed by one from a lower one, they can be switched—easy for pipelined computations
- `Sorted' is a termination property—so it is sufficient to prove for the convenient

Cache Memory

- Write implemented by *put*(x,c) to Outqueue,

Later, *get*(x,c) from Outqueue to n Inqueues, Propogate to n caches…

Typical state is complex

- Convenient: all steps in implementing a Write are done together—only 1 queue has an element, that is immediately passed on
- Measure: sum of distances between *put* and corresponding *get* (0 for convenient)

Sequential Consistency of Caches

- A cache that flushes values in queues only after all new write operations have ceased---seems correct for all finite sequences of writes (prefixes of computations)
- For an ongoing system, it fails to guarantee that a write is ever completed
- Need to prove that above is faulty and that in a good implementation, such things don't happen
- Might depend on fairness assumptions

Works for All Finite Comps., But…

- A disconnected memory system where writes get thrown away for some processes
- A bad infinite computation where one process writes a new value, but another reads the old value repeatedly, forever
- All finite prefixes are OK (it's as if we moved the bad writes to after all the reads)
- Conclusion: it is not always enough to prove that every finite (prefix of a) computation reduces to a convenient one

Implications

- Sequential consistency has a liveness aspect
- Reasonable to require that the effect of an action is eventually sensed
- Can also require that computations are not affected by occasionally reading and writing to new variables
- Together these are enough to allow showing that if all prefixes are sequentially consistent, so are full computations

An Eventual Influence Condition

◆Definition: $b \mid= \Gamma$ iff for every i,j,x,v:
If: - p_i writes to x infinitely many values different from v, and no other processor writes to x
- p_j reads from x infinitely often
Then: It is not the case that processor p_j will always read the value v from x.

Requires *minimal connectedness* of the system

An Auxiliary Variable Condition

- Can extend the memory space, add new (occasional) writes and reads to the new variable—without affecting the behaviors of the rest of the memory system
- Along with previous condition, will imply can't delay forever, even in infinite computations

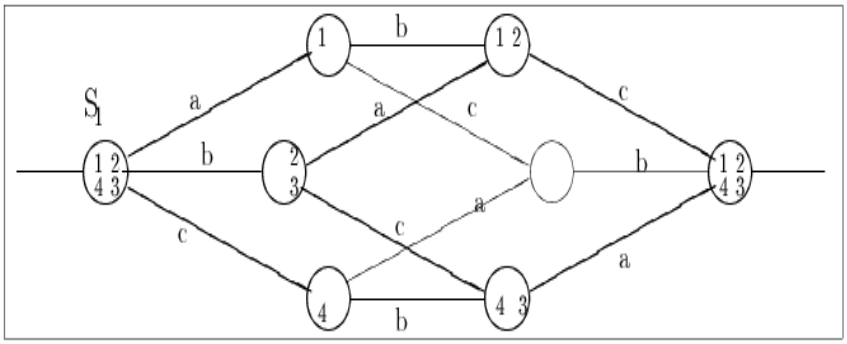
Requires *minimal independence* among variables in a memory system

What about Model Checking?

- Could (abstract and) model check that the convenient computations satisfy p.
- Could justify local subgoals from the PVS proof environment (e.g., higher layer transition followed by a lower layer transition can always be switched…a diamond at that point)
- Maybe prove all reduce to convenient?

Model Checking of Equivalence

- Directly show on state transition graph that every computation is swap equivalent to a convenient one.



Possible interleavings of operations a,b,c (independent at S_1)

Reminder:Convenience and Top-Down

- Basic idea: start with over-synchronized and over-communicating system with thick operations
- Correctness should be obvious/simple
- To refine: use two stages as seen before
  - implement using convenient
  - show loosening to general
- Problem: not supported well in design or programming languages

Summary on Convenient Comps. (I)

- Convenient Computations are another tool in our toolbox
- Sometimes appropriate (and already used in common special cases)
- Three basic tasks, with different techniques
- Invites using a combination of methods
- For theorem proving, a proof environment helps, but doesn't make it painless
- Translating among notations is needed

Summary on Convenient Comps.(II)

- Convenient computations and exploiting independence are another way to make proofs easier, for appropriate problems.
- Most closely related to partial order reductions of model checking (rather than data abstraction, symmetry, cone-of-influence, bounded model checking,…)
- For unbounded data and parametric systems, often still need some theorem proving.
- Packing a new method in a proof environment is a good idea

Observations on Translating Among Models

- Effective use of convenient approach mixes theorem proving and model checking
- Different tools express models differently wrt atomic actions, modules, synchronization,…
- A general translation framework is preferable to ad hoc translations
- Translations are not always exact, so properties may be different in source and target

Reasons for Translating (cont.)

- *From specification notation without tools*
  - Statecharts have simulation—want Alloy analysis
  - VHDL to verification tool (e.g., SMV)
  - Need `Back implication'
- *To hardware description languages*
  - Part of usual production stream
  - Translate verified design into VHDL
  - `Import' verified designs for synthesis

Decomposed Proofs

- *Abstraction*: Reduce an infinite state system to a (small) finite one
  - Justify the reduction in theorem prover
  - Check the small version in model checker
- *Proof by cases* (the PVS paradigm)
  - Some theorem proving
  - More decision procedures for fragments
  - Most model checking for finite cases

Invariant generation

- How to find relevant invariants for PVS, HOL, ACL2, or STeP induction proofs?
- Pick up invariants from Petri net versions
- Check candidates for feasibility on a Murphi explicit state version
- Generate simple invariants using SMV model checking

Also: Convenient Computations

- *Verify properties for convenient* using model checking
- *Justify reductions respect families* of properties using theorem proving
- *Show every computation reduces* to a convenient one, in a mixture of approaches
- *Justify translations* (to be explained) in theorem proving or decision procedure

Difficulties

- Different atomic actions
- Different synchronization models
- Different modularity constructs
- Different fairness /liveness assumptions
- Different control statements

Implications

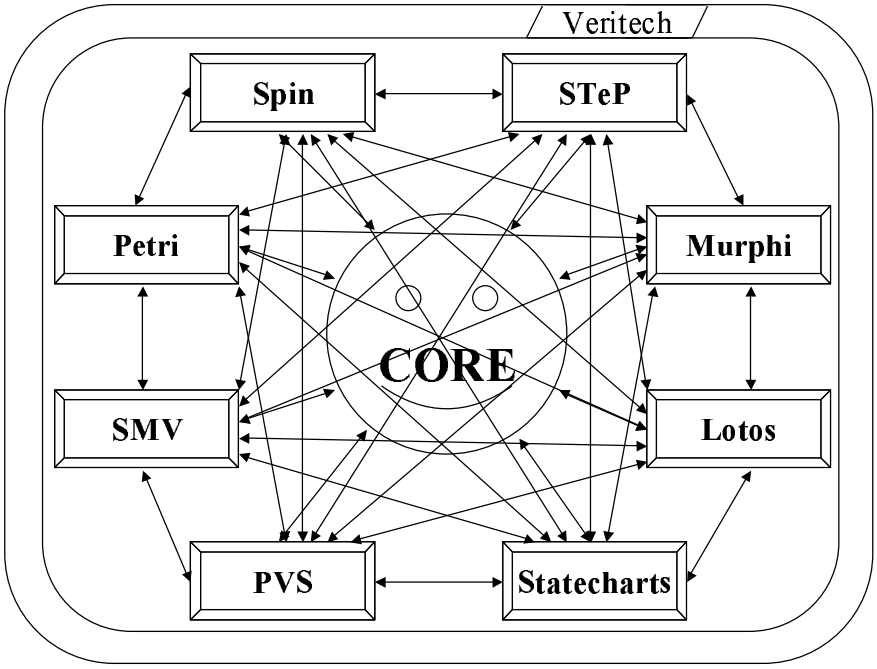
- Translations cannot be exact
- Need metrics and proofs about translations
- Different uses influence requirements of a translation
- Need to be more systematic:
  - Framework for *implementing* translations
  - Framework for *understanding* translations

The VeriTech Project

(joint work with Orna Grumberg)

- Integrates existing tools
- Permits translating from one tool to the other, going through a core notation
- adds functionality only once
- develops heuristics for choosing suitable tool for a task

http://www.cs.technion.ac.il/Labs/ssdl/veritech



The Core: Design Criteria

- Maximal expressiveness as a target
- Maximal simplicity as a source
- Ease of analysis for adding reductions
- Not needed for programming by humans

Note: the core has more than just intermediate code (kitchen sink approach….)

Core Description Language (CDL)

Structured textual transition system

- Transition:
  - name
  - enabling condition
  - relation between current and next variables
- Parameterized Module: collection of transitions
- Parallel composition:
  - asynchronous
  - synchronous
  - partially synchronous

A Sample Program: 3 modules with partial synchrony

```
MODULE SENDER (a: INT){
  VAR readys: BOOL INIT false
  TRANS produce:
        enable: !readys
        assign: readys'=true
                a'=5
  TRANS send:
        enable: readys
        assign: readys'=false}
```

```
MODULE BUFFER (c,d: INT){
  VAR cok: BOOL INIT true, dok: BOOL INIT false
  TRANS get:
        enable: cok
        assign: cok'=false
  TRANS move:
        enable: !cok /\ !dok
        assign: d'=c; cok'=true; dok'=true
  TRANS put:
        enable: dok
        assign: dok'=false}
```

```
MODULE RECEIVER (b: INT){
  VAR readyr: BOOL INIT true
      vr: INT
  TRANS consume:
        enable: !readyr
        assign: readyr'=true
                vr'=b
  TRANS receive:
        enable: !readyr
        assign: readyr'=false}
```

```
MODULE COMB(){
   VAR s,t: INT
   SENDER(s) |(send,get)| BUFFER(s,t) |(put,receive)| RECEIVER(t)}
```

Translations and Properties

- Differences in models imply not all properties can be maintained
- Classic approach: identify classes of properties that ARE maintained
- More general: transform the properties along with the model
- Two possibilities:
  - Importing: Property of the source has a transformed version true of the target
  - Back-implying: Property of the target has a version true of the source

Faithful Translations

- A theory for translations and properties
- Transform classes of temporal logic properties uniformly (syntactically)
  - *tr(f1, f2)* a property transformer
  - TR(m1, m2) a model transformer (compiler)
- Can have *import/ back-implication/ both*

Faithful Translations (cont.)

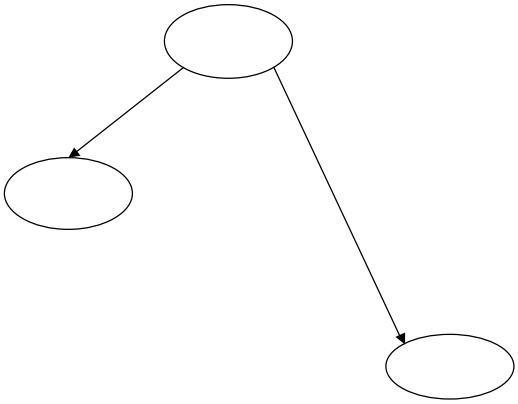
- Then TR and *tr* are
  - *import faithful*
  - *back-implication faithful*
  - *strongly faithful* w.r.t. M1, M2, L1, and L2, if whenever TR(m1,m2) and *tr(f1,f2)* then
  - m1 satisfies f1 $\Rightarrow$ m2 satisfies f2
  - m2 satisfies f2 $\Rightarrow$ m1 satisfies f1
  - m1 satisfies f1 $\Leftrightarrow$ m2 satisfies f2

Example: Grain of Atomicity

- In TR(m1,m2)
  - only m2 has *visible*
  - *visible* states of m2 are all of m1's
  - transition of m1 goes to sequence with intermediate $\neg$*visible*
- In *tr* L1 is CTL, L2 is result of :
  - **X**p……**X**($\neg$*visible* **U** (*tr*(p) $\wedge$*visible*))
  - p**U**q…..(*visible*$\Rightarrow$*tr*(p)) **U** (*visible*$\wedge$*tr*(q))
  - **F**p ……**F**(*visible* $\wedge$ *tr*(p))
  - **G**p ……**G** (*visible*$\Rightarrow$ *tr*(p))

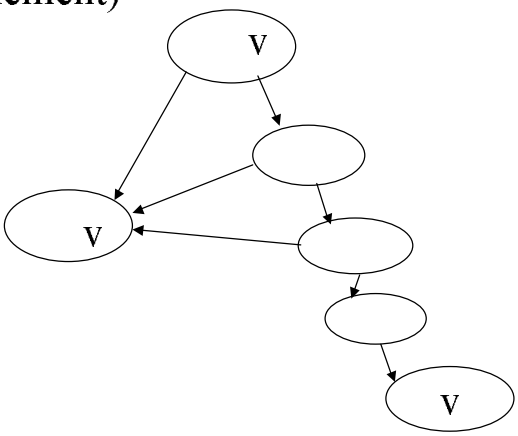
Semantic description of source

- Original tree of m1



Semantic changes define TR

- One transition becomes many (pure refinement)



Claim: TR and *tr* are strongly faithful

- Xp in source $\Leftrightarrow$
  $\mathbf{X}(\neg visible\ \mathbf{U}\ (tr(\mathrm{p}) \wedge visible))$ in target
- Gp in source $\Leftrightarrow \mathbf{G}\ (visible \Rightarrow tr(\mathrm{p}))$ in target

Effective Transformations

- The property transformation is strongly faithful BUT…
- Transforms CTL to CTL*
- For a target with LTL, translating LTL is OK
- When target is CTL, better to maintain nesting, replace AGp by AG(visible $\Rightarrow$ p) and AFp by AF(visible $\wedge$ p) when p is atomic

Additional semantic changes

- Complex initialization → special code
- Nondeterminism → choose and test, plus *fail*
- Final states → repeating plus *term* flag
- Built-in fairness → asserting `fair implies…'

Note: all are from real compilers (CDL-Murphi, Spin-CDL, CDL-SMV,…..)

Translations and Properties

- Differences in models imply not all properties can be maintained
- Classic approach: identify classes of properties that ARE maintained
- More general: transform the properties along with the model
- Two possibilities:
  - Importing: Property of the source has a transformed version true of the target
  - Back-implying: Property of the target has a version true of the source

Additional Information

- Source to target connections usually lost after translation
- Needed for
  - Tracing an error in the target back to the source
  - Improving sequences of translations (esp. translating back after slight changes)
- Related to incremental compilation and error messages of usual compilation
- Needed when many sources yield one target, or one source is split to many places in target

What information?

- Data transformations (changes in variable representations)
- Scope changes (local becomes global)
- Operation correspondences (two transitions from synchronized modules go to one target)
- Added variables and program counters—and control structures they represent

Given in XML construct for connecting XML definitions (here—the source and target)

Observations

- *Information gathered by the compiler* can determine appropriate *property transformations*
  - *Data mappings*
  - *Visible/hidden transitions*
- The *property transformations* can determine what is essential in the translation—permitting *optimizations* in the compiler
- *Small* transformation classes indicate *bad* model *translations* (source and target are unrelated!)

Are such proofs enough?

- Problem: what if the compiler *doesn't* actually do the semantic changes assumed?
- Possible solution: use translation validation idea due to Amir Pnueli
  - Validate that each model and its translated version satisfy the needed semantic relation
  - Generate verification conditions for a source and target, automatically verify.

For a Translation A → B

- *Compiler* from A to B (including optimizations from additional information about source/target)
- Generic property *transformations*
- Schema of *additional information on A → B*
- Description of intended *semantic changes*
- Abstract *proofs*: if the compiler makes the intended semantic changes, the transformations are …faithful for effective source sublanguage ..
- Compiler add-on for *validation proof obligations*

For Each Specific Translation (Activation)

- *Source* code
- *Target* code (result of applying the compiler)
- Specific *added information* gathered
- Automatically generated *proof obligations* for semantic assumptions to be validated
- BDD-based *proofs* that for this activation the intended semantic changes occurred (and thus the abstract proof applies this time)
- Properties already proven about source/target

Summary on Translations

- Common among proof methods and to/from notations without associated tools
- Need to carefully analyze the effect of translations on properties
- Faithful property transformations provide a measure of the quality of a translation
- Additional information gathered during translation can be used in defining property transformations and improving later translations