

A Framework for Translating Models and Specifications

Shmuel Katz

Computer Science Department

The Technion

Some Questions

- **What** is done today?
- **Why** translate?
- Is there **a better way**?
- Are required **properties affected**?
- Can we/do we need to **prove something** about the translation?
- **Can we integrate** different formal methods into one proof?

Some Existing Translations

- Murphi to PVS
- SMV to Spin
- Verilog to Cospan
- Java abstractions to SMV, Spin
- SMV to VHDL
- Statecharts to SMV, VIS, and Spin
- State machine to Petri net

Some More Translations

- Stochastic process algebras to Eden
- Class+object+Statechart to graph diagram with graph transformations
- RSDS to B, SMV, and Java
- Statecharts to B and B to Statecharts
- Object-Z + ASM to SMV
- SDL to bREAL
- SPW to HOL (hand coded?)

Reasons for Translating

- *Different tools handle different properties*
 - SMV: branching t.l. (possible interrupt)
 - STeP: infinite domain, real-time extension
 - Spin: linear, finite state properties
- *Sometimes tools fail*
 - Model checking has state explosion
 - Theorem proving needs clever invariants
 - Petri net analysis can explode
 - Process algebra bisimulations are too strong

Reasons for Translating (cont.)

- *A notation without tools as a source*
 - Statecharts have simulation, easy to look at
 - Verilog designs need to be analyzed
 - Translations provide verification
 - Need *'Back implication'*
- *A notation without tools as a target*
 - Part of usual production stream or for visualization
 - 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, 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

- Two stage verification:
 - Verify properties for convenient computations
 - Prove every computation can be *reduced* to a convenient one
- *Scenario-based* specification and verification
- Common: swap order of independent actions
 - Model check for convenient
 - Justify back implication of reduction with theorem proving

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

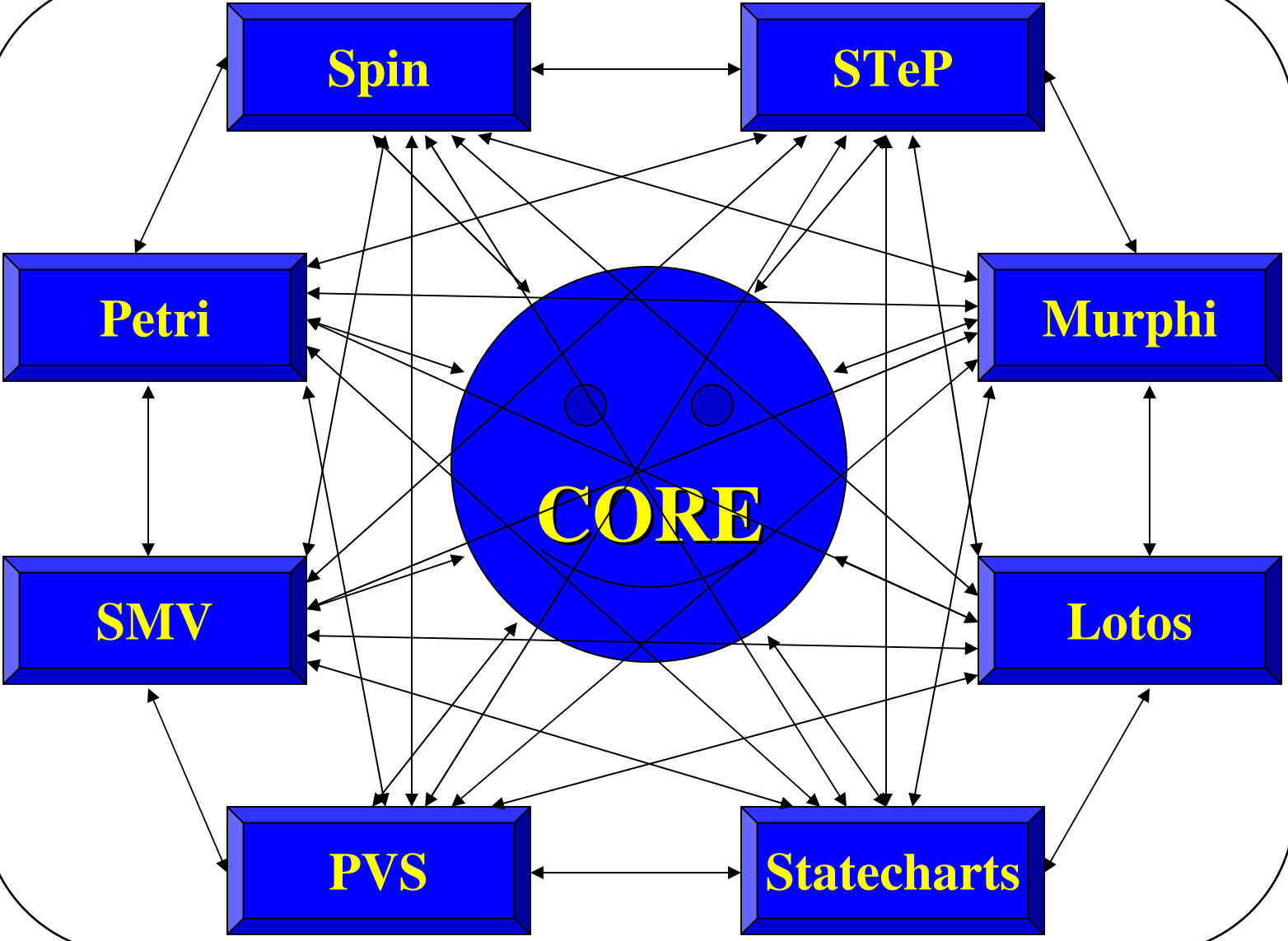
The VeriTech Project

(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/veritech>

Veritech

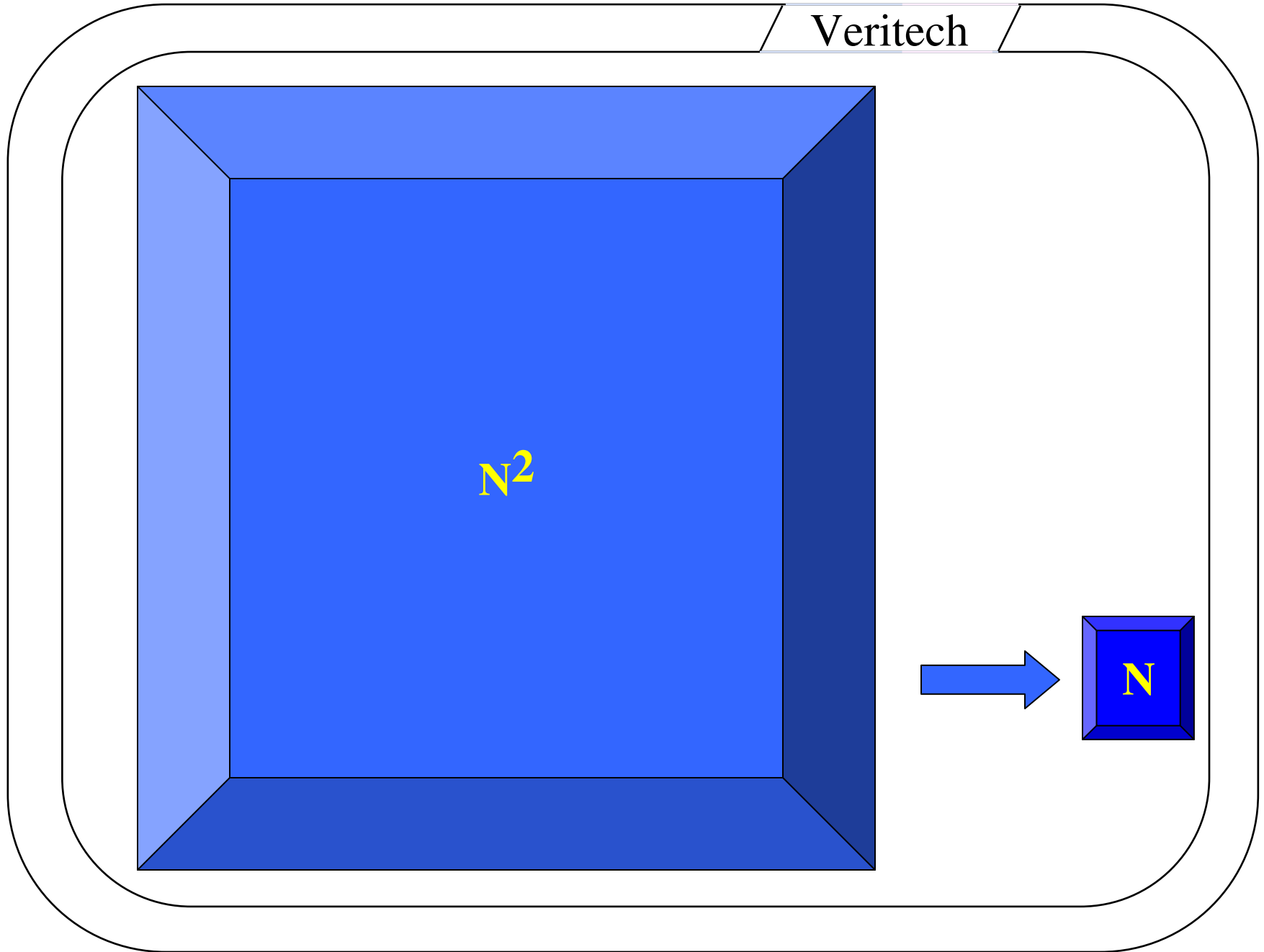


Veritech

N^2



N



Stages in the Project

- ✓ Import and install **existing tools**
- ✓ Design a **core** representation
- ✓ Implement **translations** to/from core (10 done with more to come)
- ✓ Develop **benchmark** case studies
- Add internal modules for **extra analysis**
- Gather **additional information** for tracing changes

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  $\wedge$  !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)}
```

Sample Difficulties

- Grain of Atomicity
 - An atomic action in one notation is translated to a sequence of actions in another
 - Requires visible/hidden, affects invariants

Synchrony and Asynchrony

- Require simulations
- Differ in assumptions on maintaining previous values

Present Work

- Creating a **translation environment** to make new translations easier (XML –based)
- Analyzing how real translations are related: **what is maintained** and **what is disturbed?**
- Conducting **case studies** to better understand which problems are best solved using which tools.
- Develop an **expert system** for selecting the best tool for a problem

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 source codes yield one target, or one source is split to many places in target

Example: CDL \rightarrow STeP initialization

- CDL has nondet. init. of x from any set A
- STeP has nondet. init. only from $0..n$
- Solution: array with A values, nondet. init. of index i , new transition N to set x to $A[i]$, disable all other transitions until after N
- Additional information: all parts added to treat the initialization... an XML version using links is under development

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)
 - $tr(f1, f2)$
 - $Tr(m1, m2)$
- Can have *import*/ *back-implication*/ *both*

Faithful Translations (cont.)

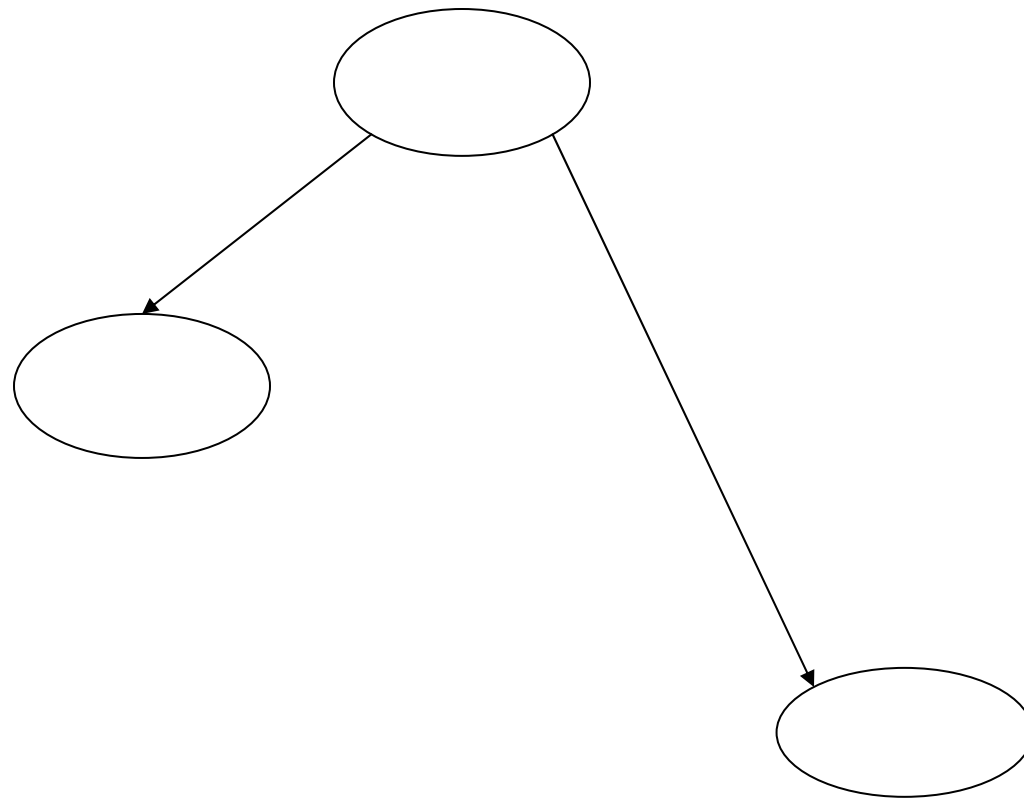
- Then TR and tr are
 - *import faithful*
 - *back-implication faithful*
 - *strongly faithful* w.r.t. M1, M2, L1, and L2, if whenever TR(m1,m2) and $tr(f1,f2)$ then
 - m1 satisfies f1 \Rightarrow m2 satisfies f2
 - m2 satisfies f2 \Rightarrow m1 satisfies f1
 - m1 satisfies f1 \Leftrightarrow m2 satisfies 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 :
 - $\mathbf{X}p \dots \mathbf{X}(\neg visible \mathbf{U} (tr(p) \wedge visible))$
 - $p \mathbf{U} q \dots (visible \Rightarrow tr(p)) \mathbf{U} (visible \wedge tr(q))$
 - $\mathbf{F}p \dots \mathbf{F}(visible \wedge tr(p))$
 - $\mathbf{G}p \dots \mathbf{G}(visible \Rightarrow tr(p))$

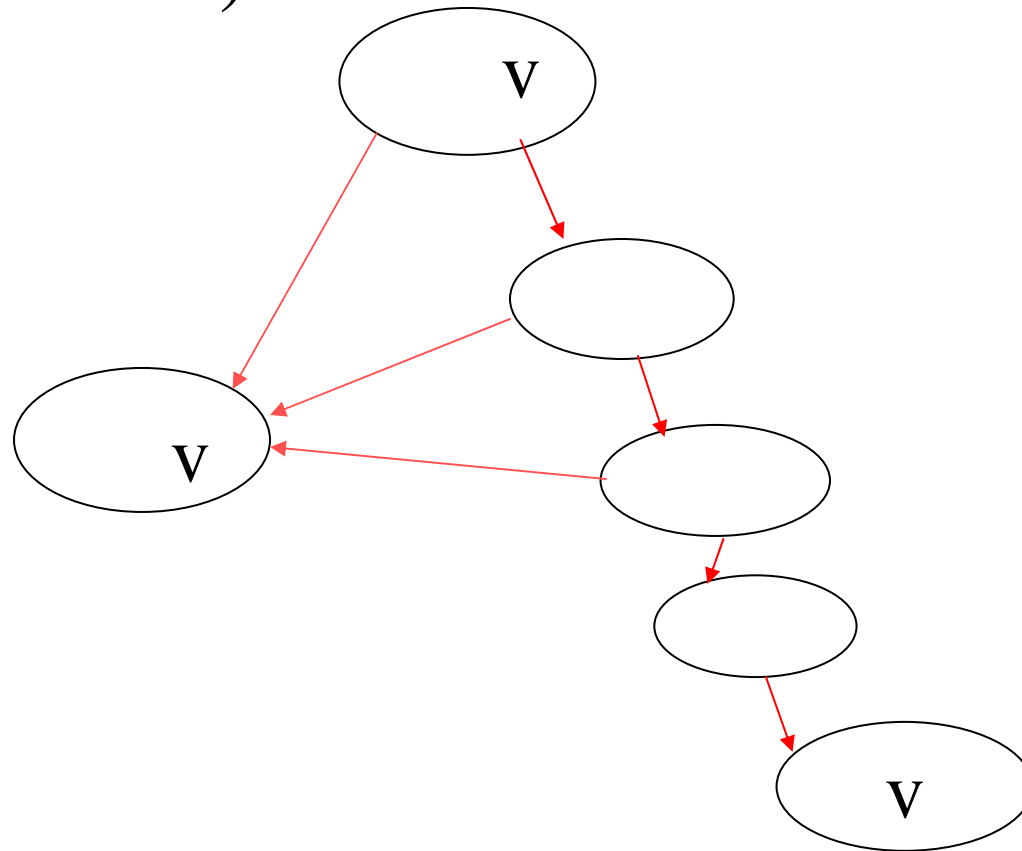
Semantic description of source

- 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
 $\mathbf{X}(\neg visible \mathbf{U} (tr(p) \wedge visible))$ in target
- Gp in source \Leftrightarrow $\mathbf{G} (visible \Rightarrow tr(p))$ 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 $AG(\text{visible} \Rightarrow p)$ and AFp by $AF(\text{visible} \wedge p)$ 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,.....)

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.

Observations

- The *additional information* can determine the appropriate *property transformations*
 - *Data mappings*
 - *Visible/hidden transitions*
- The *property transformations* can determine what is essential in the translation—permitting *optimizations*
- *Small effective* transformation classes indicate *bad* model *translations*

For a Translation $A \rightarrow B$

- *Compiler* from A to B (including optimizations from additional information about the source)
- Generic property *transformations*
- Schema of *additional information on $A \rightarrow B$*
- Description of intended *semantic changes*
- Abstract *proofs*:

If the compiler makes the intended semantic changes, the transformations are ...faithful for effective source sublanguage ...

For Each Specific Translation (Activation)

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

The kitchen sink approach

- All of the above, *plus*:
- Variants for finite or infinite state
- Partial order reductions
- Symmetry and cone-of-influence reductions
- Combinations of different aspects of a system described in different notations (glue)
- A database for every system under development

Conclusions

- A Framework for translations is good software engineering and allows analysis
- We need a theory of model translations
- It gives us valuable insight into the quality of translations and can improve compilers
- Translations can and should be part of the formal verification process—otherwise there are holes in our reasoning