On Efficient Program Synthesis from Statecharts

Andrzej Wąsowski

Joint work with Peter Sestoft

IT University of Copenhagen
Glentevej 67, 2400 Copenhagen NV, Denmark

`wasowski@itu.dk`
`http://www.mini.pw.edu.pl/~wasowski/scope`

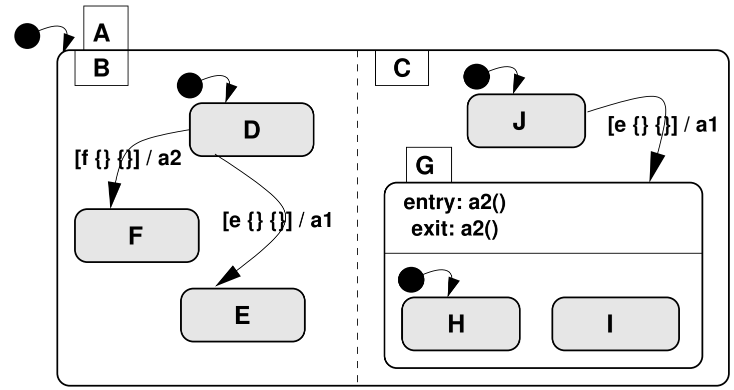
12 June 2003

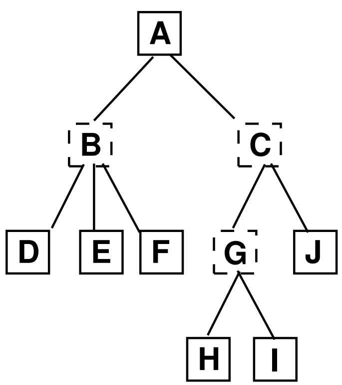
Content

- Brief overview of statecharts
- *Hierarchical* synthesis vs *Flattening* synthesis
- Implementation of SCOPE
- Theoretical and experimental evaluation
- Conclusion and future work

The Language of Statecharts

- State hierarchy:
  - parallel/sequential decompositions
  - The *root* is an and-state
  - Basic states (leaves) are and-states
  - Initial, history and deep history
- Entry/exit actions.





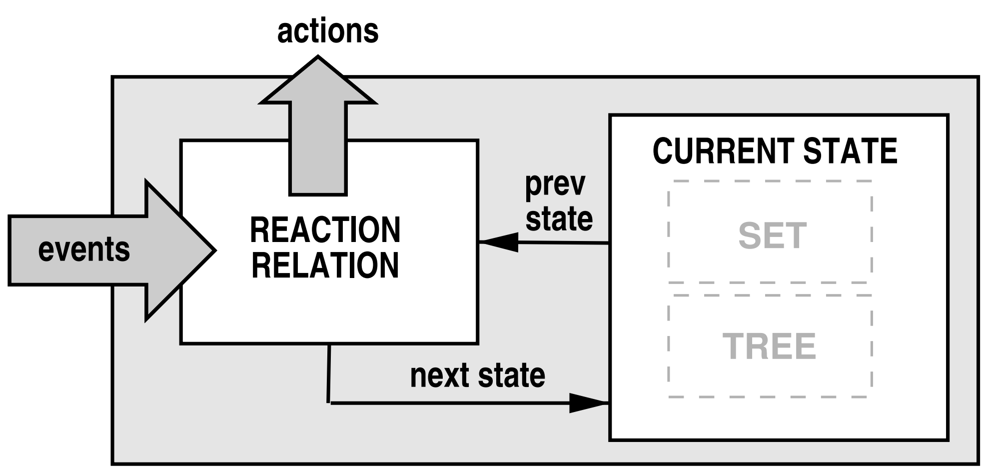
- Transitions: event + guard + action + targets
- Dynamic semantics relations: macrostep, microstep, fire, exit, enter, execute action, evaluate guard, init.

Runtime Overview

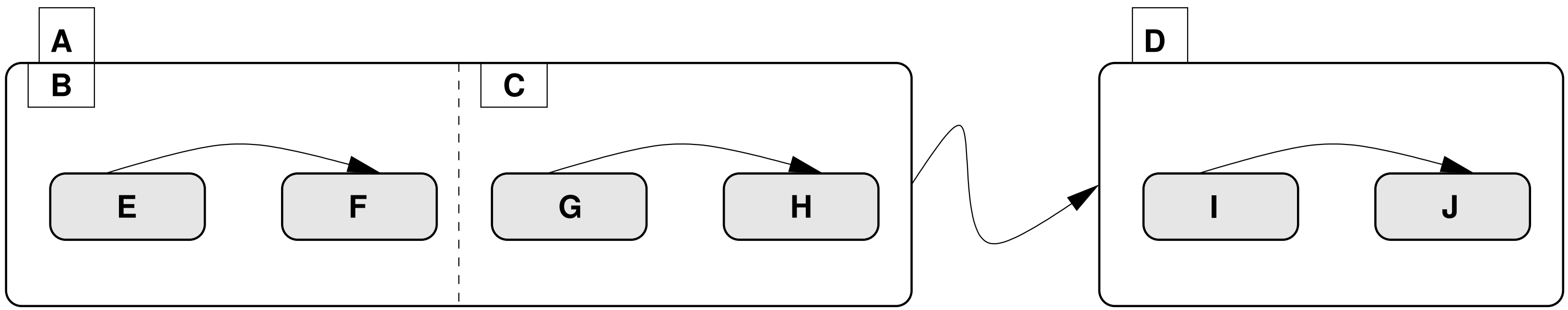
- Runtime engine

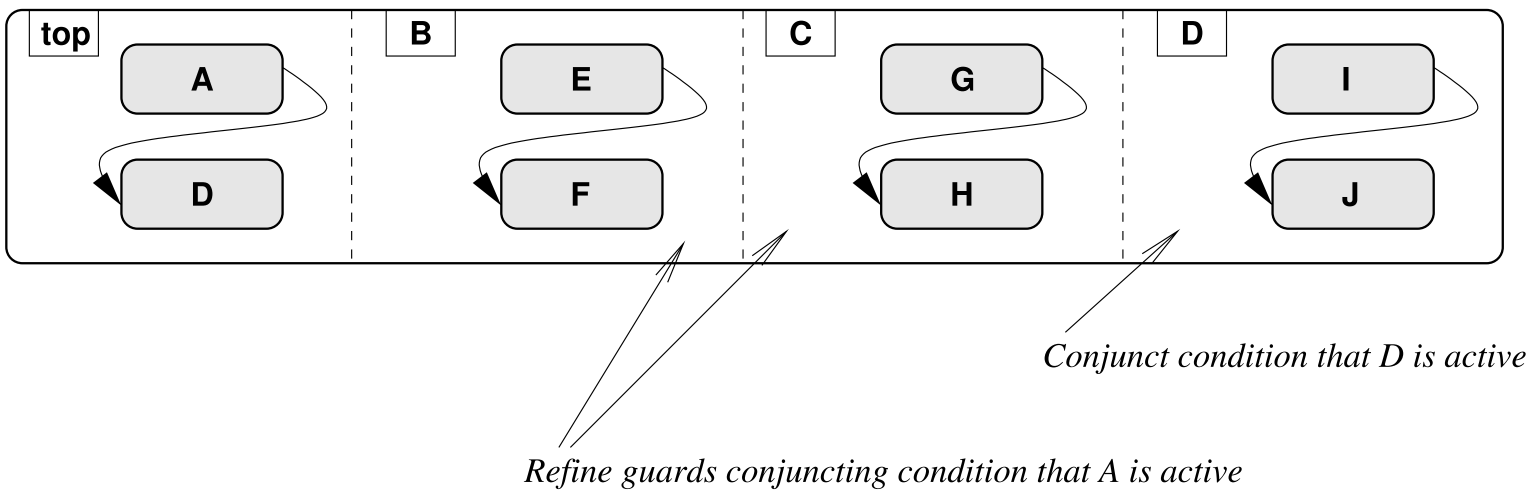
```
macrostep : event   * state -> state
microstep : event   * state * queue -> state * queue
fire      : tran    * state * queue -> state * queue
exit      : orstate * state * queue -> state * queue
enter     : targets * orstate * state * queue
                                    -> state * queue
```

- Runtime data structures



Flattening = Hierarchy Elimination

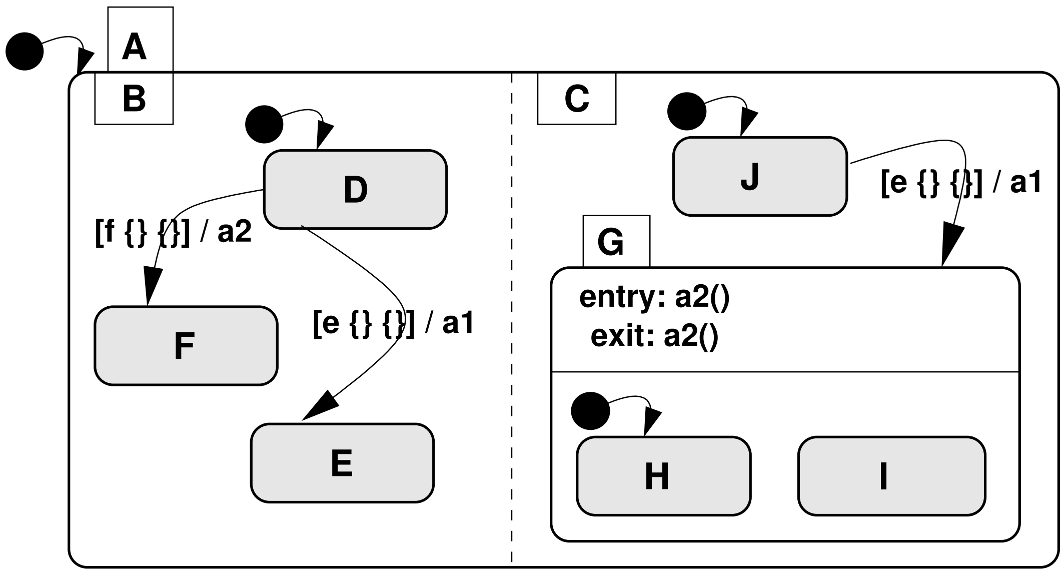


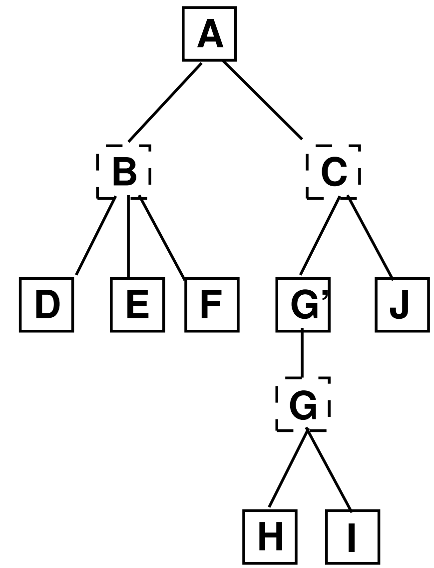


Code generated: flat set of rules and state vector, very simple runtime [visualSTATE] [Björklund, Lilius, Porres, Turku, Finland, 2001]

SCOPE: Hierarchical Code Generator

Hierarchy tree

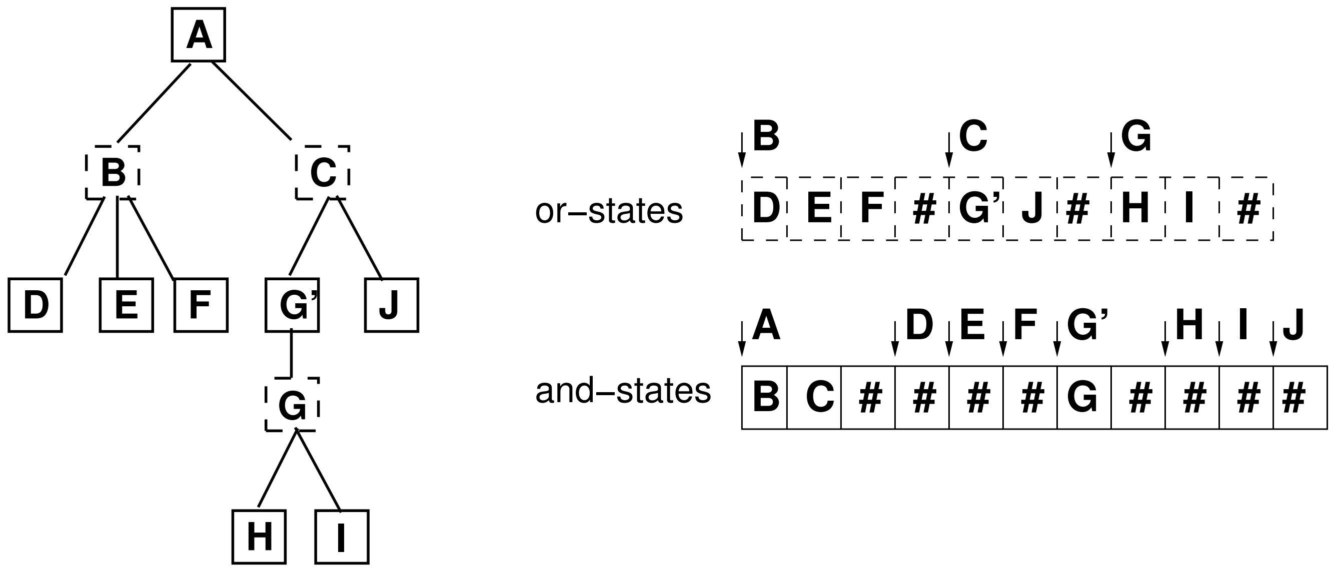




- Regular type alternation
- Separate namespace for and-states and or-states
- Shorter state identifiers at runtime
- No runtime type-checks for states
- Simpler runtime library

Hierarchy tree (II)

- The tree can be splitted in two arrays ($\#$ is *end-of-state*):



- State names are naturals
- Cheaper of following is chosen automatically:
  - Offsets in the state array
  - Consecutive numbers with dictionary of offsets

Active State Set

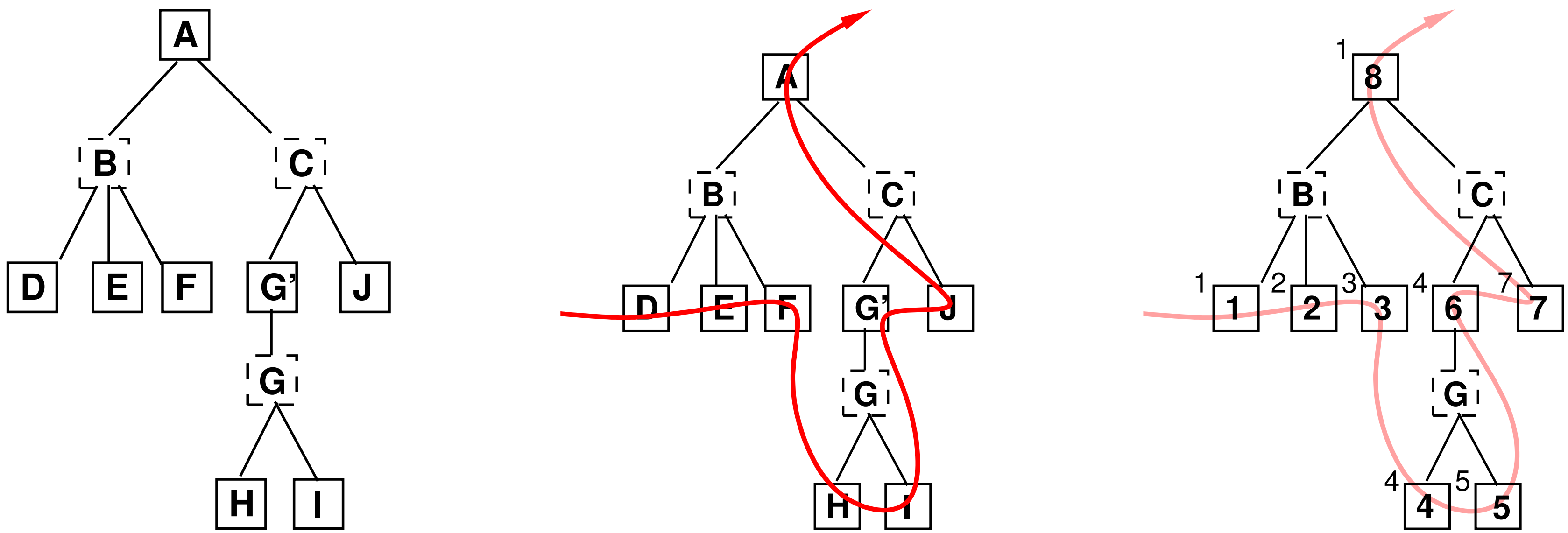
- Only active *basic* state set
- Implemented as prioritized buffer (no gaps)
- Safety demands a bound
- Trivial bound: number of basic states
- Simple improvement:

```
bound (Basic    s) = 1
bound (ORState  s) = maximum bound (children s)
bound (ANDState s) = sum     bound (chlidren s)
```

- This is exact for strictly sequential or strictly flat models

Interval Labeling

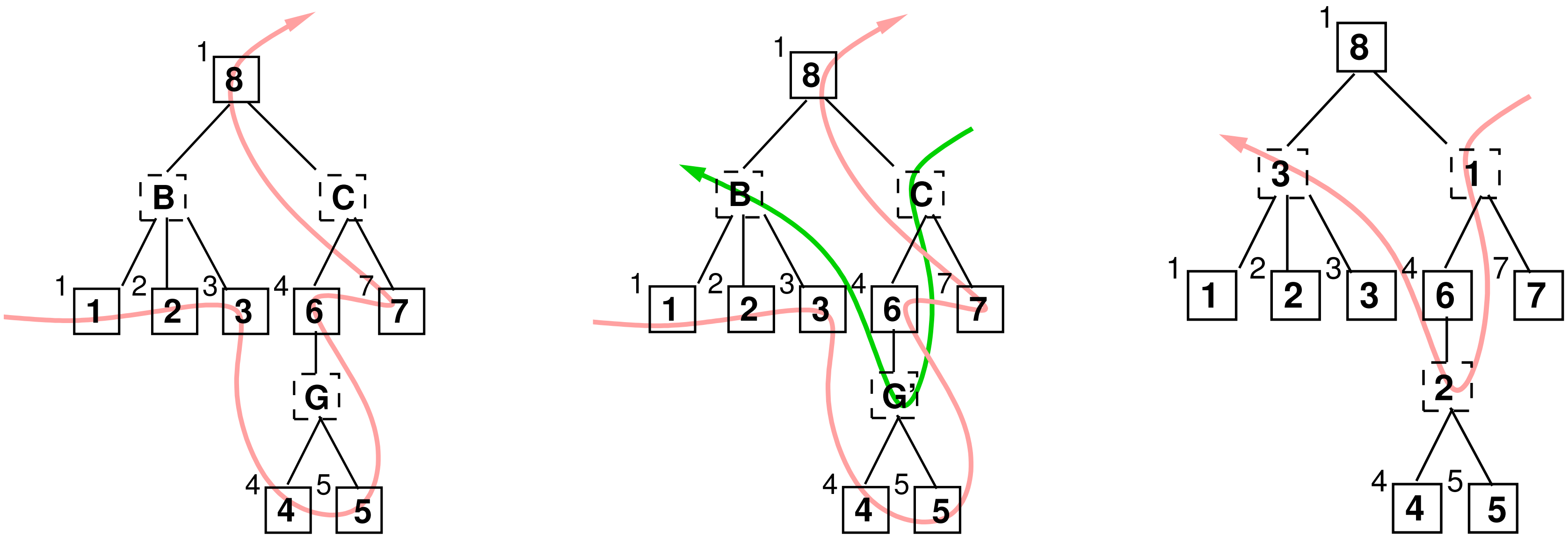
- Trivial method needs ancestorship checks – use state labeling
- Label and-states in depth-first-search post-order, left to right visitting of children



- Ancestorship is reduced to two comparisons: is given state in interval of descendants of s?
- Only *leftmost descendant* (LMD) needs to be saved in the array.
- *Exit-purity*

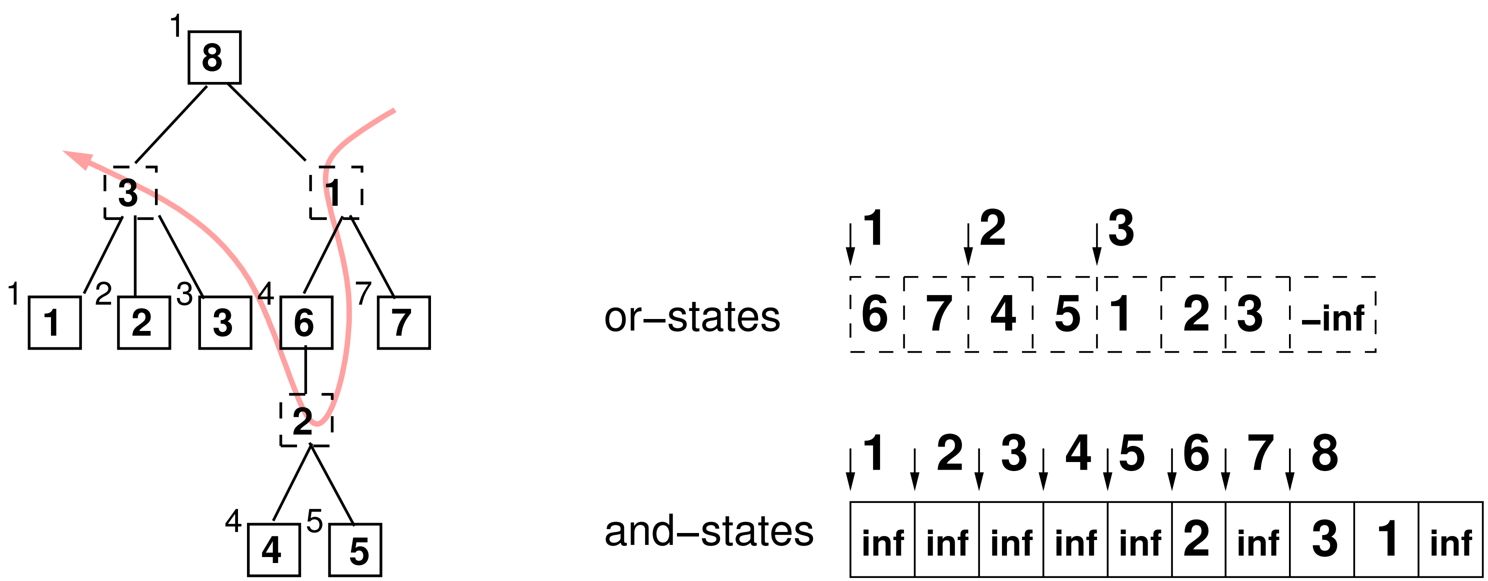
Dual labeling

Let's label or-states in order precisely dual to the one used for and-states

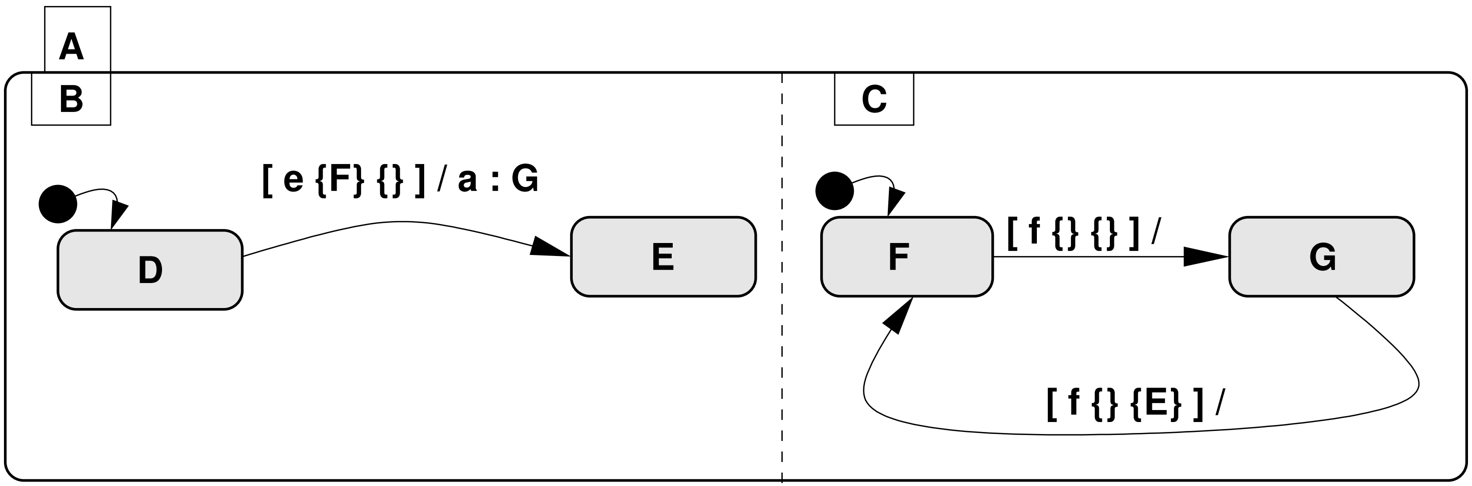


Removal of # marks

- Children lists are composed of monotonic sequences (increasing for or-states and decreasing for and-states).
- This can be used to remove end-of-state markers (#)



Scope of Transition

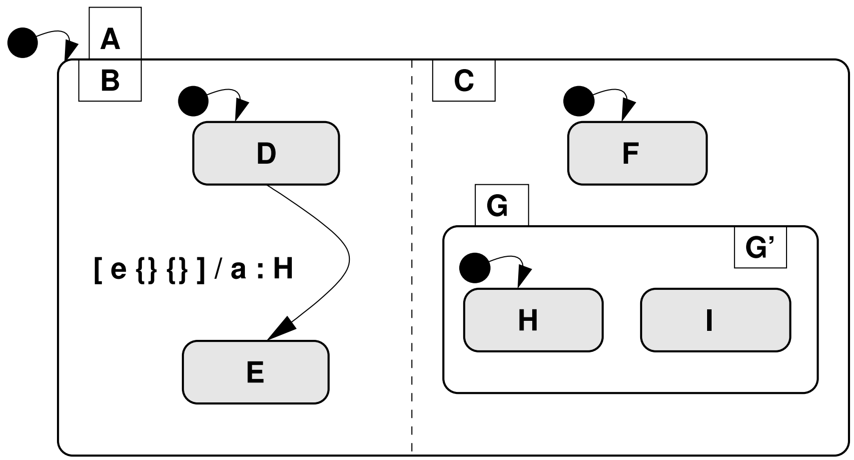


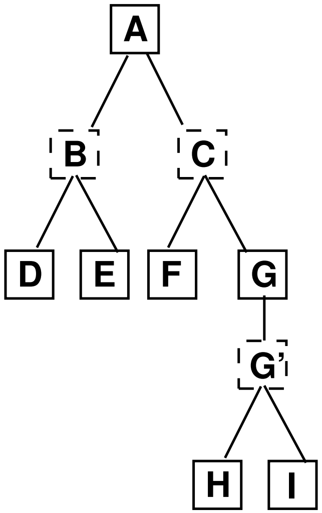
- Targets statically annotated with scopes:

$$\begin{array}{llllllll} t_1 & : [\ e & \{D, F\} & \{\} &]\ / & a & : & [B]E\ [C]G \\ t_2 & : [\ f & \{F\} & \{\} &]\ / & - & : & [C]G \\ t_3 & : [\ f & \{G\} & \{E\} &]\ / & - & : & [C]F \end{array}$$

Dynamic Scope

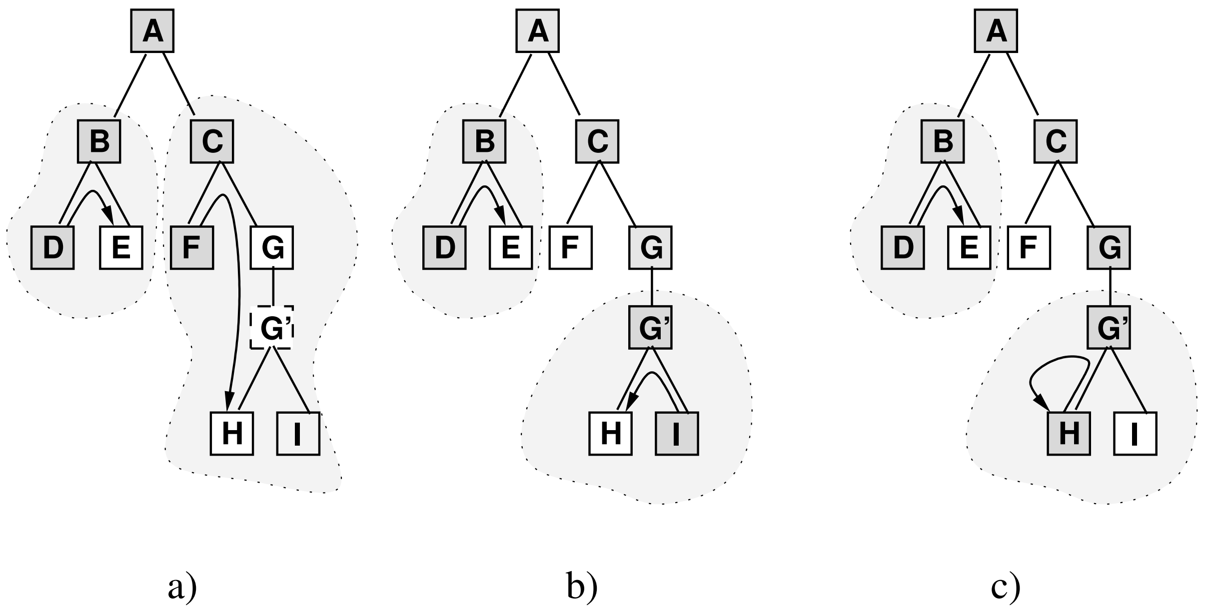
- Three legal configurations activating the transition.
- All contain D.
- Also contain one of F, H or I

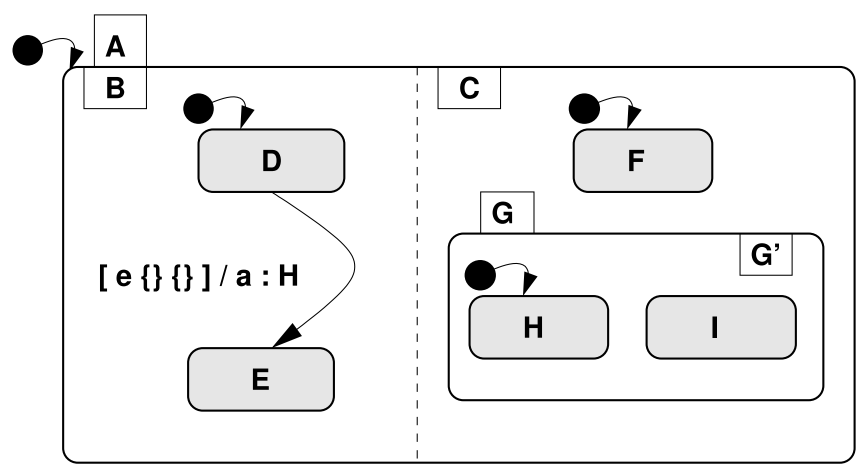




- Scope of target E is always B
- Scope of target H depends on active configuration of C ....

Dynamic Scope (II)

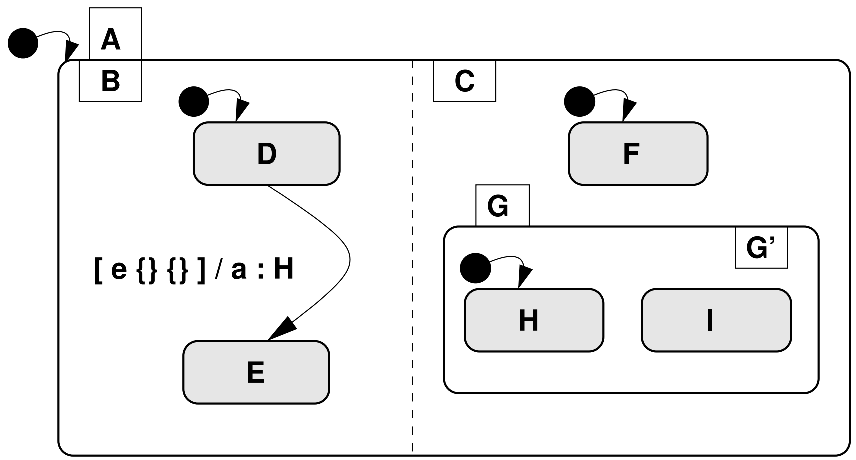




Runtime detection is slow and affects all transitions.

Dynamic Scope (III)

The problematic transition in our example:



can be rewritten with two rules:

$$\begin{array}{lcl} [\ e\ \{D,F\}\ \{\}\ ]\ /\ a & : & [B]E\ [C]H \\ [\ e\ \{D,G\}\ \{\}\ ]\ /\ a & : & [B]E\ [G']H \end{array}$$

- Adding extra positive conditions can ensure static scopes.
- Scope performs this rewriting analyzing number of possible solutions to scope-equation (BDD based implementation)

Complexity Evaluation

n – number of states, t – number of transitions
d – model depths, m – maximum over number of targets

- Linear size vs exponential size with flattening
- Scope resolution cost up to $O(d^m)$ space but constant in practice.
- Size of current state (constant factor difference):
  - flattening: linear in number of state machines
  - SCOPE: linear in number of leaves
- Elimination of end-of-state markers – constant saving

- Ancestorship test – constant time
- Activity test: $O(n)$ vs constant time of flattening
  - but exponentially less tests
- State update:
  - flattening: $O(d)$
  - hierarchical: at least $O(nd)$, improved with exit-purity

Experimental Evaluation

Pentium II 450 Mhz, GCC ver3.2, optimizing for size, bare executable sizes in bytes

Model	states	transitions	VS size	SCP size	ratio
actions01	4	1	3 596	3 840	1.07
drusinsky89	19	14	3 976	4 288	1.08
lift	18	19	4 452	4 496	1.01
peer	275	192	12 644	10 352	0.82
trios01	1121	840	28 164	19 944	0.71
trios03	1121	840	60 196	23 008	0.38

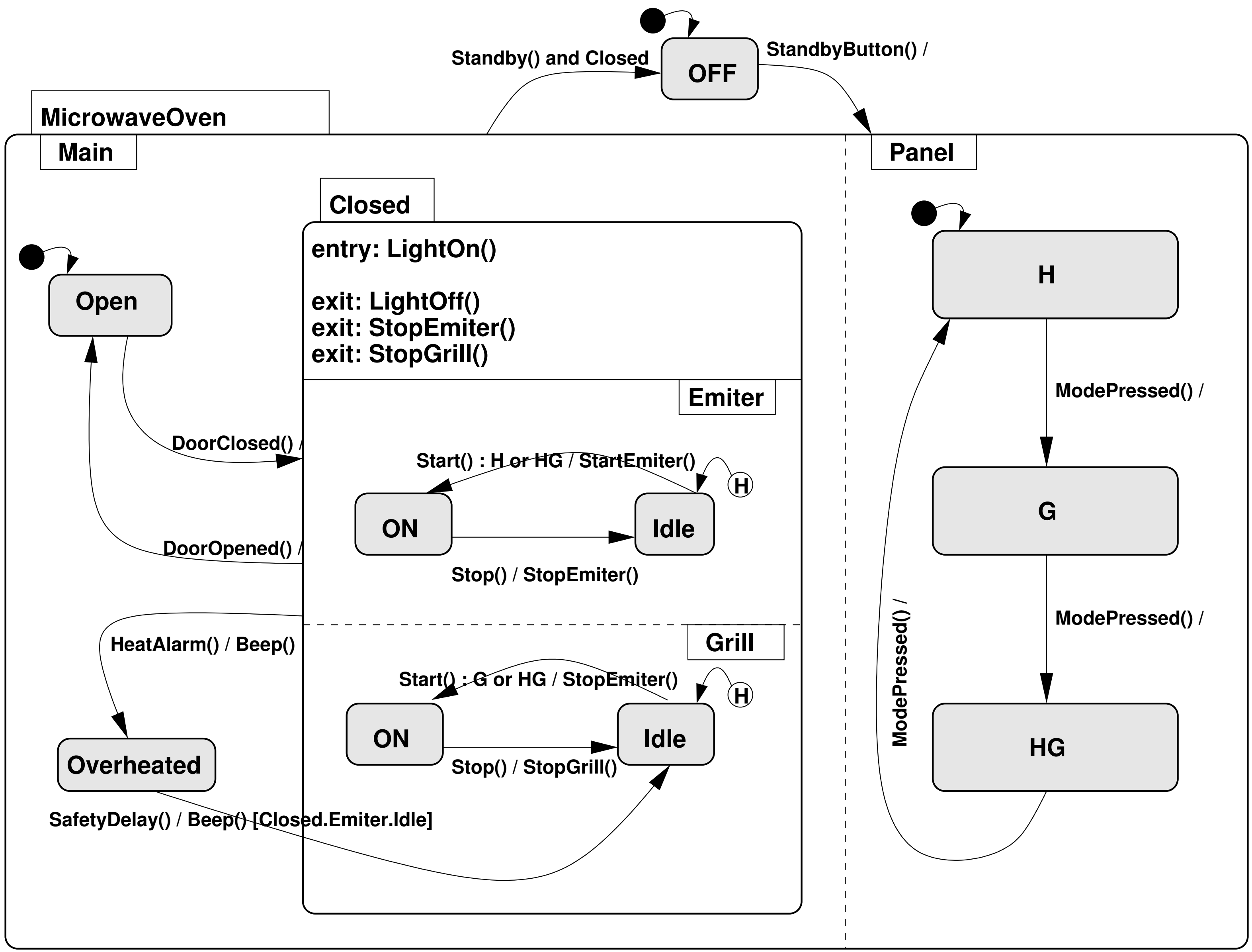
Running time measured with 10^7 random events, probability of reinitialization 0.01

Model	states	transitions	VS time	SCP time	ratio
actions01	4	1	6.24	4.69	0.75
drusinsky89	19	14	8.11	6.28	0.77
lift	18	19	14.08	29.78	2.11
peer	275	192	21.40	29.42	1.37
trios01	1121	840	534	712	1.34
trios03	1121	840	1137	763	0.67

Summary

- Conclusion
  - Experimental and complexity-theoretical evaluation of code generation methods have been presented.
  - Hierarchical code generation is feasible speed-wise
  - Hierarchical code generation can yield much smaller code
  - It performs decently for small models and well for bigger ones
- Future work
  - Optimize the structure of transitions (decision diagrams)
  - Make algorithm adaptive (meeting constraints)
  - Evaluate against self-implemented flattening
  - Evaluate against state-pattern code generation method

A Statechart Cooker



Transitions – Trivia

- Hashed into buckets by activating event
- Each bucket simply contains a list of transitions to be checked (and fired).
- This can still be optimized $\boxed{\text{TBD}}$

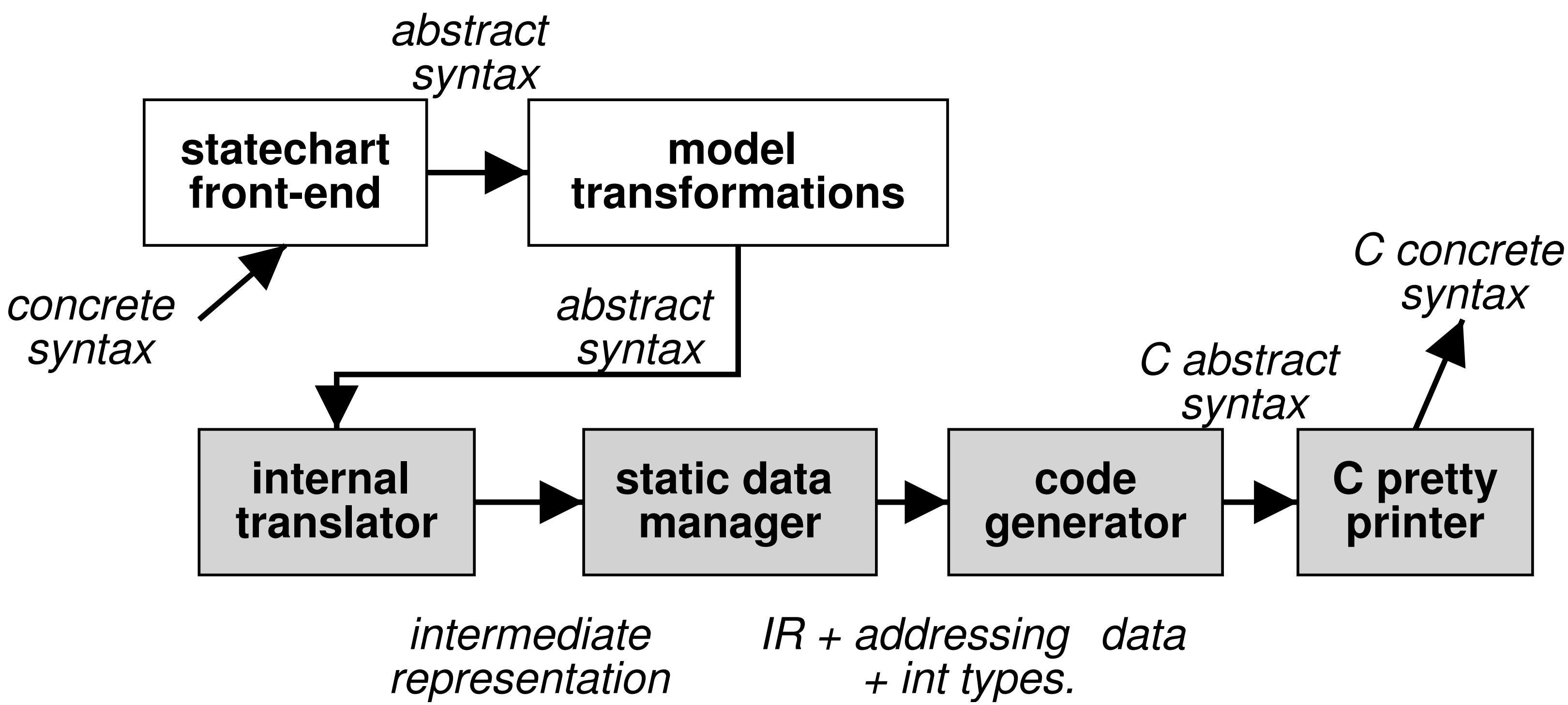
- Scope of transition
- Semantics of firing: exit scope – execute action – enter scope
- Assume scopes can be computed statically
- There should be a scope saved for each transition
- But not for flat transitions, for which it can be cheaply computed

Varia

- C compiler may not remove redundant code
- So do not generate it!

- Some elements are used more often than others
- Example: Initial markers may not be kept at all (just reorder states)
- Example: transition source, event and target should come "for-free", while other attributes may be more costly

Internal Structure of SCOPE



Experiment conditions

- Pentium II, 450 Mhz, running Linux
- GCC ver 3.2, optimizing for size (-Os)
- Bare executable sizes in bytes.
- Only control algorithm (structures $+$ runtime engine).
- External functions substituted with dummies.
- Size results similar for LCC on PC (non optimizing) and optimizing embedded systems compilers.
- Running time measured for feeding 10^7 random events with probability of model initialization before each event equal to 0.01.