Generic and Extensible Automatic Test Data Generation for Safety Critical Software with CHR

Ralf Gerlich, BSSE

Presentation for the
Seventh International Workshop on Constraint Handling Rules
CHR 2010

20th of July 2010
Edinburgh, Scotland

Contents

- Motivation
- Overview
- Implementation Issues
- Experimental Evaluation
- Outlook

Motivation

Testing takes up about 50% of the total effort for software development projects.

(For safety-critical systems – e.g. in aerospace – up to 80%)

⇒ High potential for effort reduction from automation of software test

Software test begins with selection of test inputs and expected outputs

=

Test cases

F. P. Brooks: *The Mythical Man-Month*, 1995
Myers et al: *The Art of Software Testing*, 2004

Test Input Selection

Given a list of portions of the Control-Flow Graph (CFG) of a program, find an input that, once given to the program, leads to activation of these portions in the given order.

Examples:

- **„Execute every node at least once“ (→*all-nodes*)**
- **„Traverse every edge at least once“ (→*all-edges*)**
- **„Traverse node u after node d_1 without traversing d_2, d_3, d_4, d_5 in between“ (→*all-defs*)**

S. Rapps, E. J. Weyuker: *Data flow analysis techniques for test data selection*, ICSE '82, 1982

Infeasible paths

```
void sort(int n, int[] a) {
    for (int i=0;i<n;i++) {
        int minElem=i;
        for (int j=i+1;j<n;j++)
            if (a[j]<a[minElem]) minElem=j;
        swap(a[i], a[minElem]);
    }
}
```

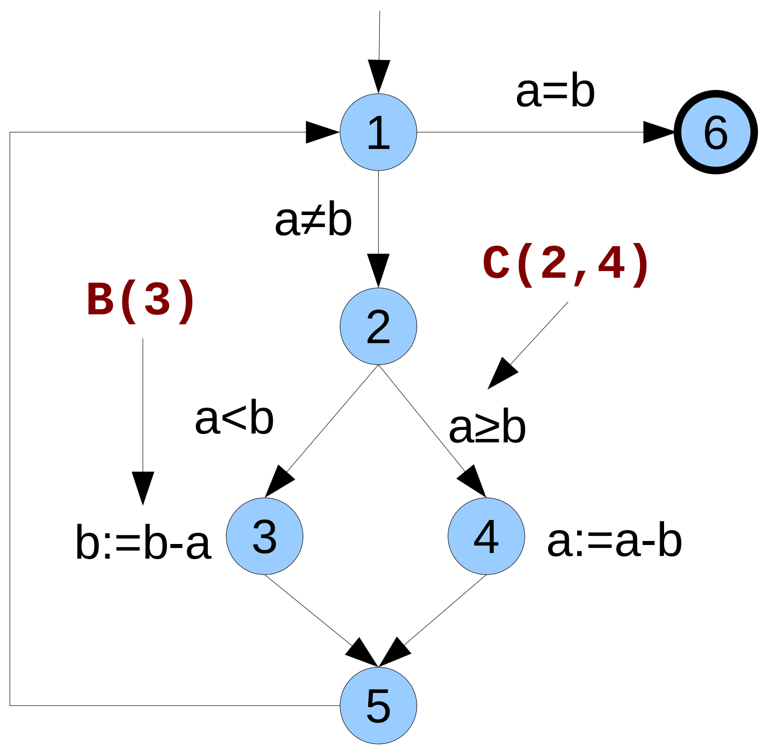
Inner loop depends on outer loop.
⇒Many paths in CFG have no associated input. (infeasible paths)

Infeasible paths are not rare enough to be ignored in practice.

Alternative Approach:
Avoid selection of infeasible paths by constraint-programming techniques.

S.-D. Gouraud: *AuGuSTe: a Tool for Statistical Testing – Experimental Results*, Technical Report, LRI, Paris, 2005

Augmented Control-Flow Graphs



- Nodes and Edges describe possible control flow
- Execution of nodes modifies program state
- Selection of edges by a set of predicates
- Relational expression:
  - `x B(3) y`
  - `x C(2,4) x`

Generic Path Constraint Relations

$$x \; \mathcal{S}_{[a,b]} \; y$$

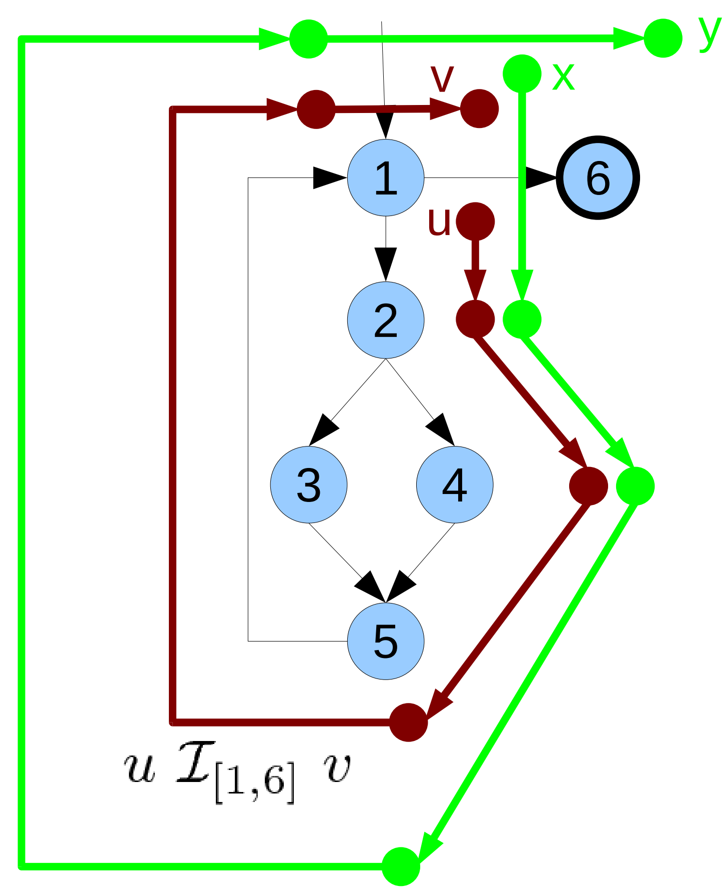
`spec(A,B,X,Y)`

There is a path from node a to node b, transforming input x to output y. (“Specification”)

$$u \; \mathcal{I}_{[a,b]} \; v$$

`ispec(U,A,B,V)`

There is a path from node a to node b, with u the output of node a and v the input of node b. (“Inner Specification”)



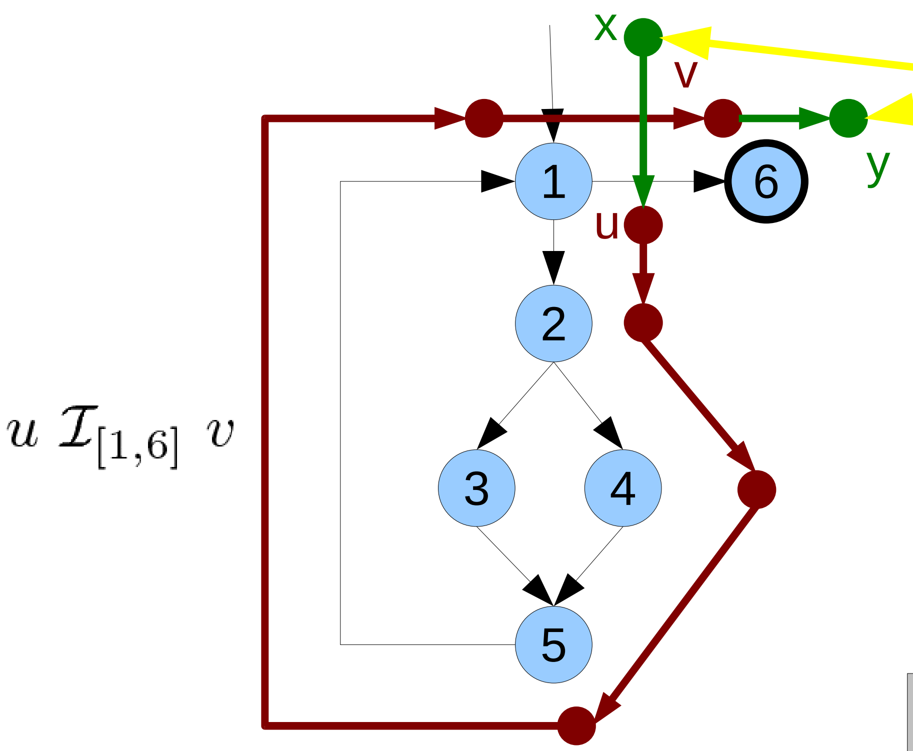
$u \; \mathcal{I}_{[1,6]} \; v$

$x \; \mathcal{S}_{[1,6]} \; y$

Built-In Constraints

Built-In Constraint	Semantics
`edge(U,V)`	There is an edge from U to V
`reachable(U,V)`	V is reachable from V via one or more edges
`body(U,X,Y)`	*X B(U) Y*
`cond(U,V,X)`	*X C(U,V) X*
`deffree(U,W,V)`	No path from U to W contains a definition of variable V
`onallpaths(U,W,V)`	All paths from U to V proceed via W
`value(X,Var,Val)`	Val is the value of variable Var in memory state X

Eliminate Specification



The specification differs from the inner specification by the additional bodies of the endpoints.

$$\mathcal{S}_{[1,6]} = \mathcal{B}(6) \circ \mathcal{I}_{[1,6]} \circ \mathcal{B}(1)$$

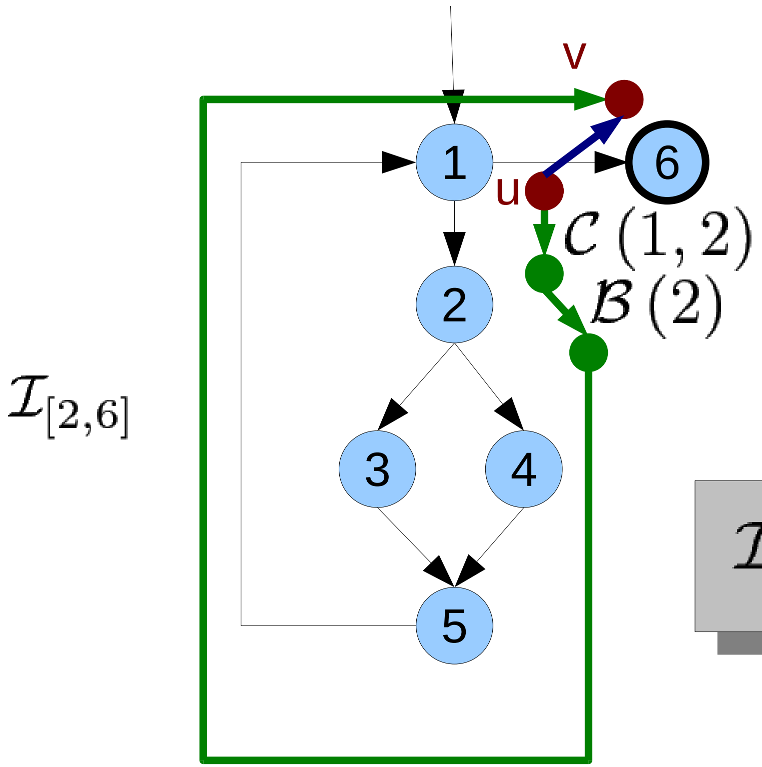
(Read concatenation right-to-left)

But:

$$\mathcal{S}_{[1,1]} = \mathcal{B}(1) \cup \mathcal{B}(1) \circ \mathcal{I}_{[1,1]} \circ \mathcal{B}(1)$$

```
spec_to_ispec @ spec(U,W,X,Z) <=>
    (U=W, body(U,X,Z));
    (body(U,X,Y1), ispec(Y1,U,W,Y2), body(W,Y2,Z)).
```

Forward Step

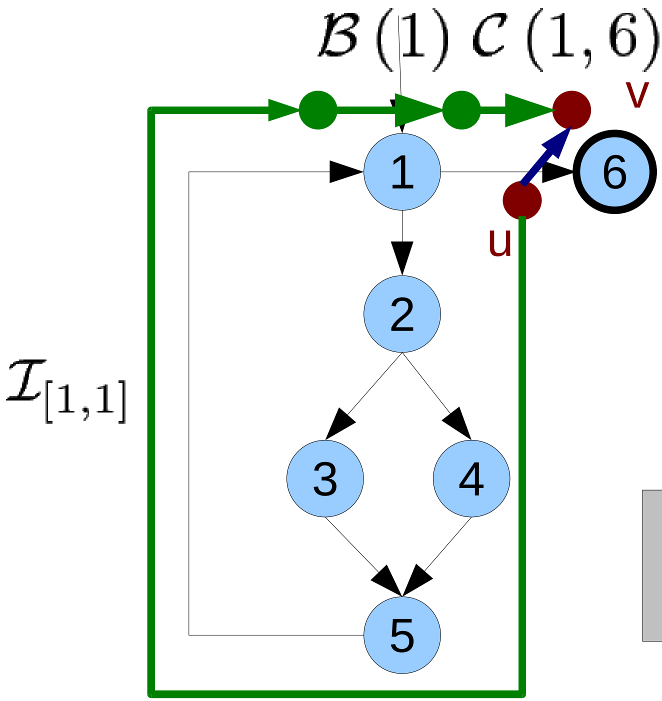


We can either traverse the edge from 1 to 6 or continue from 1 via 2 to 6.

$$\mathcal{I}_{[1,6]} = \underline{\mathcal{C}(1,6)} \cup \underline{\mathcal{I}_{[2,6]} \circ \mathcal{B}(2) \circ \mathcal{C}(1,2)}$$

```
step_fwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(U,V), reachable(V,W),
     cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
```

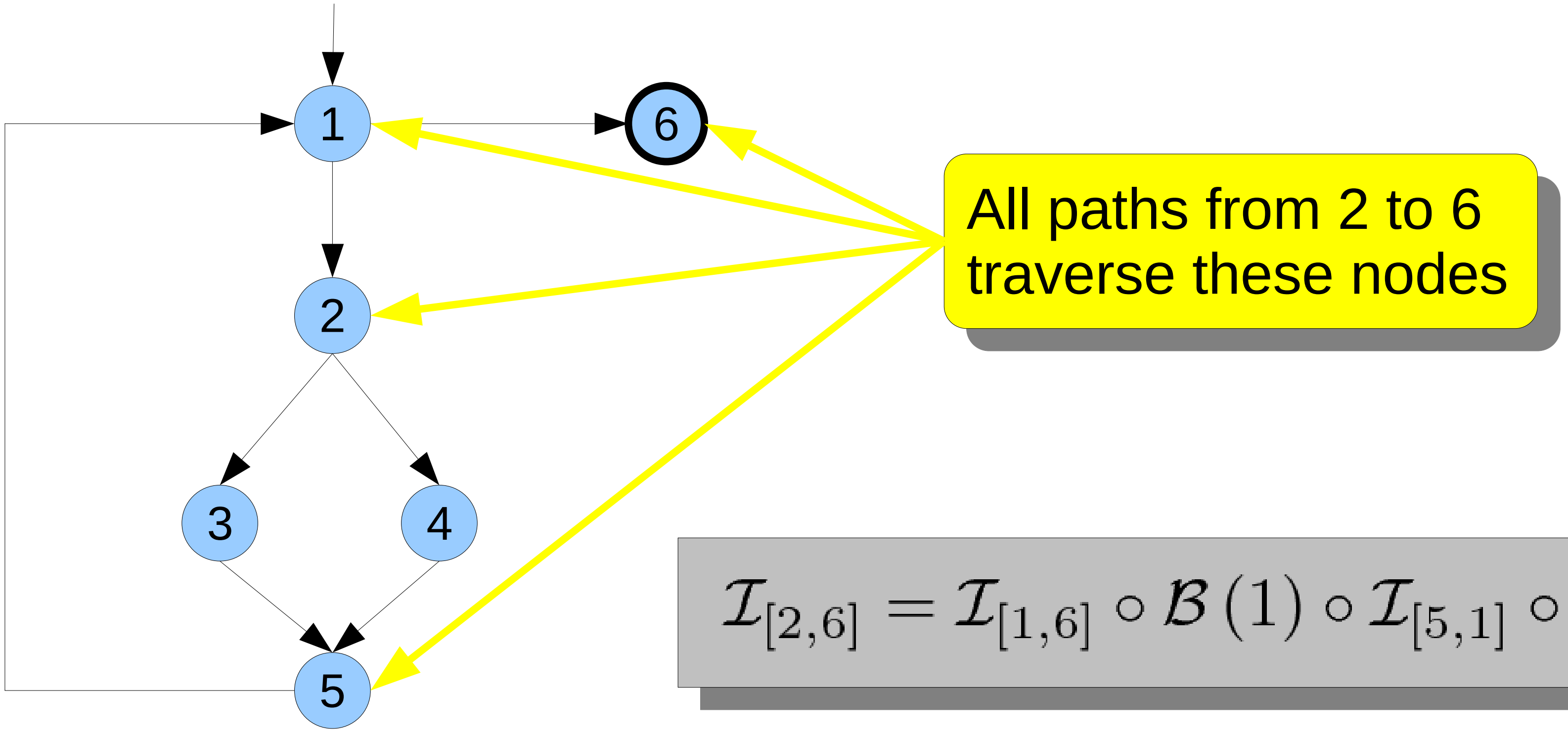
Backward Step



We can either traverse the edge from 1 to 6 or continue from 1 via 1 to 6.

$$\mathcal{I}_{[1,6]} = \underline{\mathcal{C}(1,6)} \cup \underline{\mathcal{C}(1,6) \circ \mathcal{B}(1) \circ \mathcal{I}_{[1,1]}}$$

```
step_bwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(V,W), reachable(U,V),
     ispec(X,U,V,Z), body(V,Z,Y), cond(V,W,Z)).
```

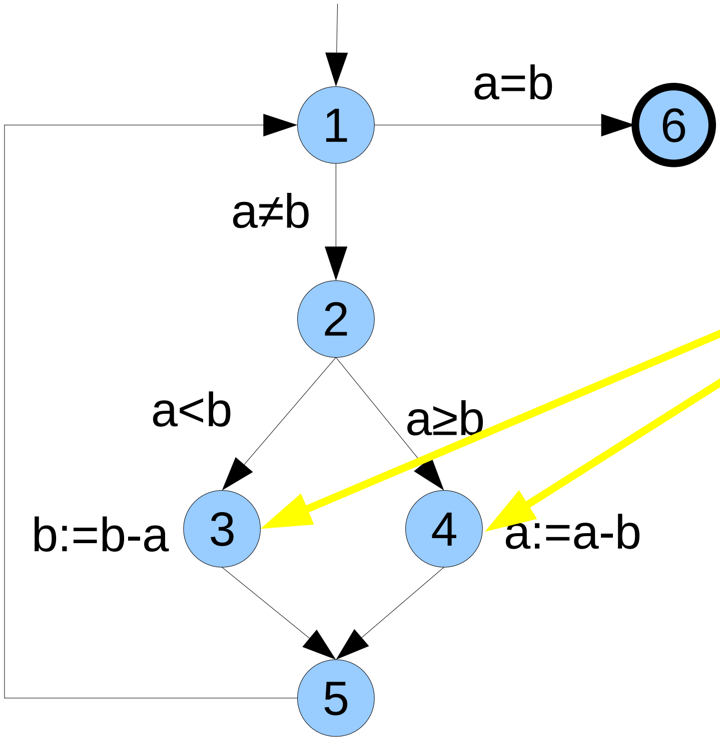


$$\mathcal{I}_{[2,6]} = \mathcal{I}_{[1,6]} \circ \mathcal{B}(1) \circ \mathcal{I}_{[5,1]} \circ \mathcal{B}(5) \circ \mathcal{I}_{[2,5]}$$

```
split @ ispec(X,U,W,Z) <=> reachable(U,W), onallpaths(U,W,V) |
    ispec(X,U,V,Y), body(V,Y,Z), ispec(Y,V,W,Z).
```

Instead of „rediscovering“ facts in all search branches, we try to „predict“ them and avoid throwing them away on backtracking.

Data-Flow Prediction



Variable typically keep their value through large parts of execution.

```
prop_var @ ispec(U,W,X,Y) ==> reachable(U,W), deffree(U,W,V) |
    value(X,V,V1), value(Y,V,V2), V1=V2.
```

We can use data-flow information to propagate information about the memory state across sub-path borders.

Complete CHR$^{\vee}$ Implementation

```
spec_to_ispec @ spec(U,W,X,Z) <=>
    (U=W, body(U,X,Z));
    (body(U,X,Y1), ispec(Y1,U,W,Y2), body(W,Y2,Z)).
prop_var @ ispec(U,W,X,Y) ==> reachable(U,W), deffree(U,W,V) |
    value(X,V,V1), value(Y,V,V2), V1=V2.
split @ ispec(X,U,W,Z) <=> reachable(U,W), onallpaths(U,W,V) |
    ispec(X,U,V,Y), body(V,Y,Z), ispec(Y,V,W,Z).
step_fwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(U,V), reachable(V,W),
     cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
step_bwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(V,W), reachable(U,V),
     ispec(X,U,V,Z), body(V,Z,Y), cond(V,W,Z)).
```

Complete? Not so fast!

Important Properties

Verifiability and Comprehensibility

⇒

Make use of connection between declarative and operational semantics

Different solutions are not equivalent and committed choice not possible.

⇒

Search

Random Test Case Selection

⇒

Probabilistic Search with „simple“ statistical model

Issue 1: Implicit Search

```
step_fwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(U,V), reachable(V,W),
     cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
```

The rule is existentially-quantified over V and solutions are not equivalent.
⇒Implicit Search; not supported by $CHR^{\vee}$

Operationally correct only if host language supports search over built-in constraints (e.g. Prolog) or if **edge/2** becomes a user-defined constraint, enumerating all alternatives.

Workaround: Use Prolog as host language

Issue 2: Deterministic Derivation

```
step_fwd @ ispec(X,U,W,Z) <=>
     (edge(U,W), X=Z, cond(U,W,X));
     (edge(U,V), reachable(V,W),
      cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
```

De-Facto Semantics of CHR$^{\vee}$: First alternatives first.
⇒Alternatives enumerate paths by length in ascending order

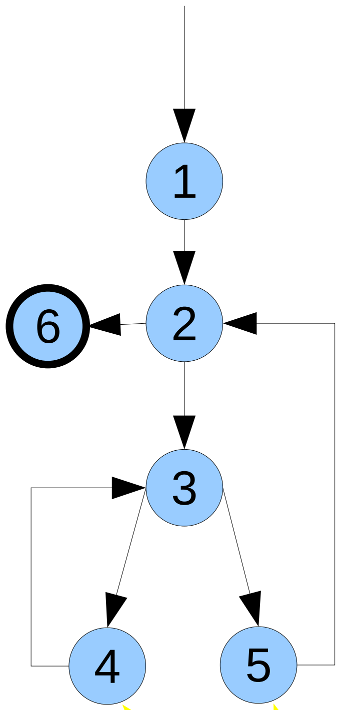
Swapping of alternatives could lead to infinite recursion.

Software Test requires some randomness in test case selection to avoid bias away from faults.

Solution: “Probabilistic CHR$^{\vee}$”, CHRiSM

CHRiSM was not yet available.

Digression: Handling loops probabilistically



The mean number of iterations of such an inner loop is 2 if the probabilities of continuation and termination is the same (0.5)

Consequence: Different probabilities for different values of V, depending on U and W.

```
step_fwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(U,V), reachable(V,W),
     cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
```

Exiting or continuing inner loops often is the choice between two successor nodes.

Neither PCHR nor CHRiSM support this

Issue 3: Probabilistic Search

```
step_fwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(U,V), reachable(V,W),
     cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
```

If both alternatives are selected with p=0.5, the mean path length is 2. Similarly, if all values of V have same probability, inner loops degenerate.

PCHR requires splitting this up into two rules to allow different probabilities for them.

By splitting up we are leaving the realm of declarative correctness.

Solution: CHRiSM

Yes, we'll try that!

Issue 4: Statistical Model

```
step_fwd @ ispec(X,U,W,Z) <=>
     (edge(U,W), X=Z, cond(U,W,X));
     (edge(U,V), reachable(V,W),
      cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
```

PCHR considers rule instances instead of rules when selecting randomly.

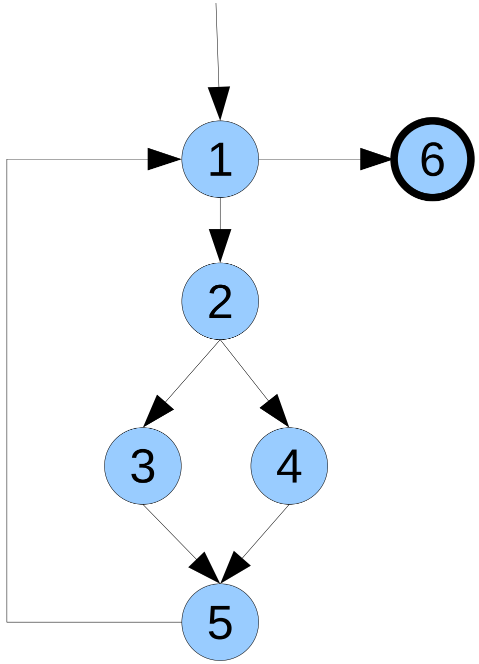
There are almost always more instances of the “step” alternative than of the “edge” alternative.
The statistical model for that is difficult to manage.

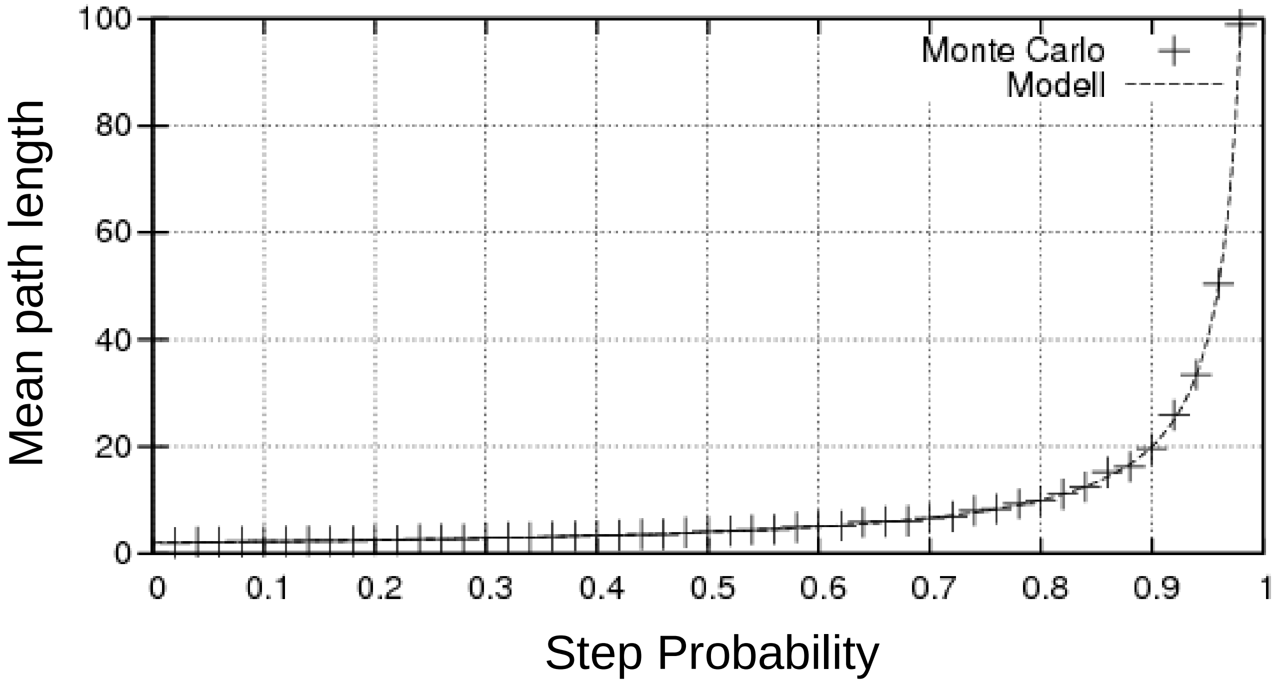
PCHR: uncontrollable path growth

Solution: CHRiSM

Yes, we'll try that!

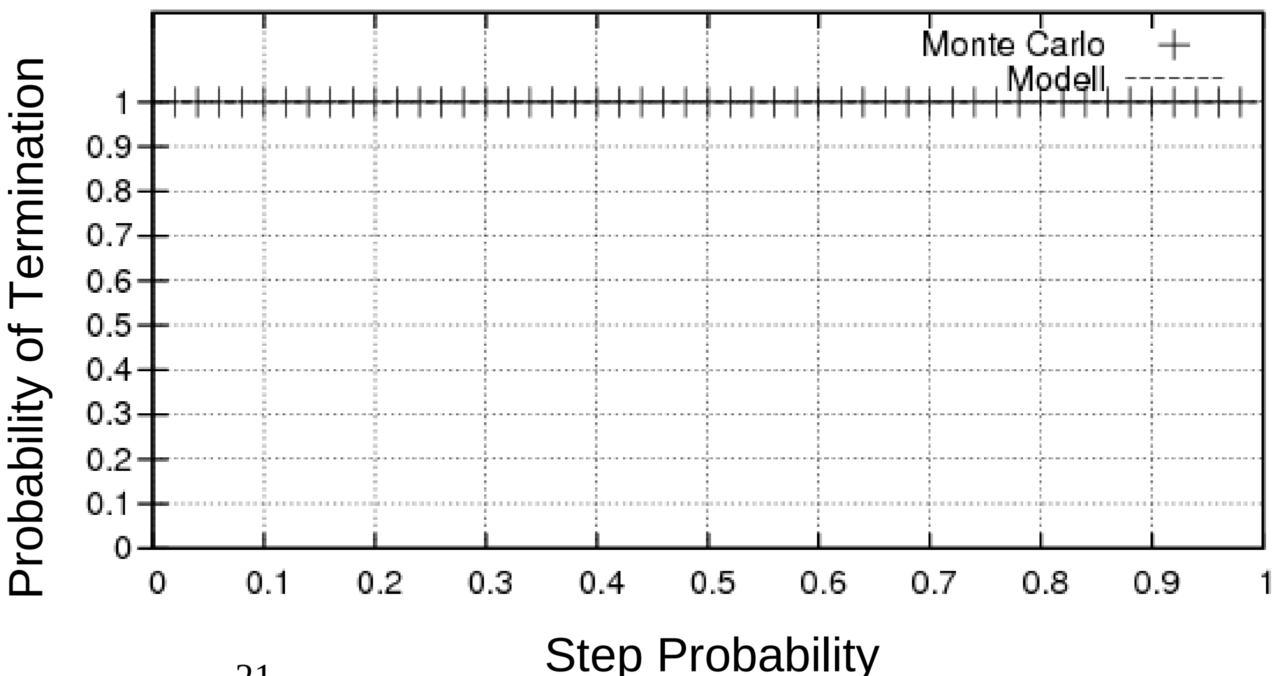
Evaluating the Statistical Model





Main Discoveries:

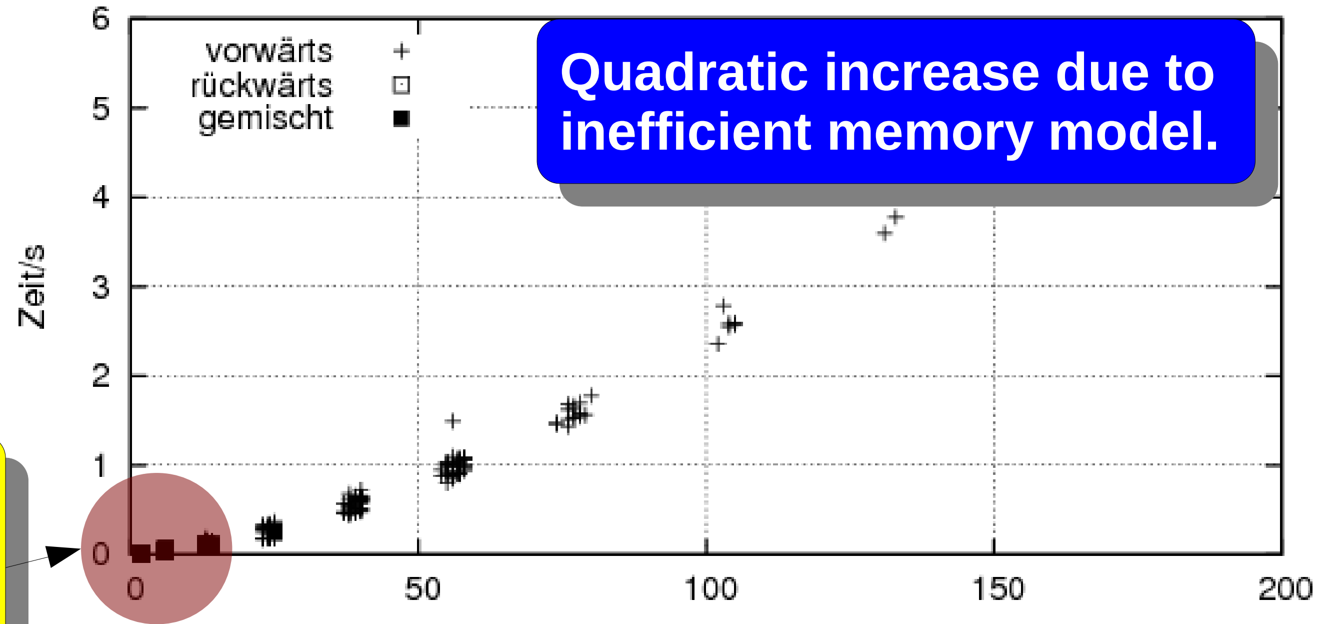
- **Bias for shorter paths**
- **Countermeasure: vary p**
- **Probabilistic Termination**
- **“modulo” Haltingproblem**



Runtime Complexity on Selection Sort

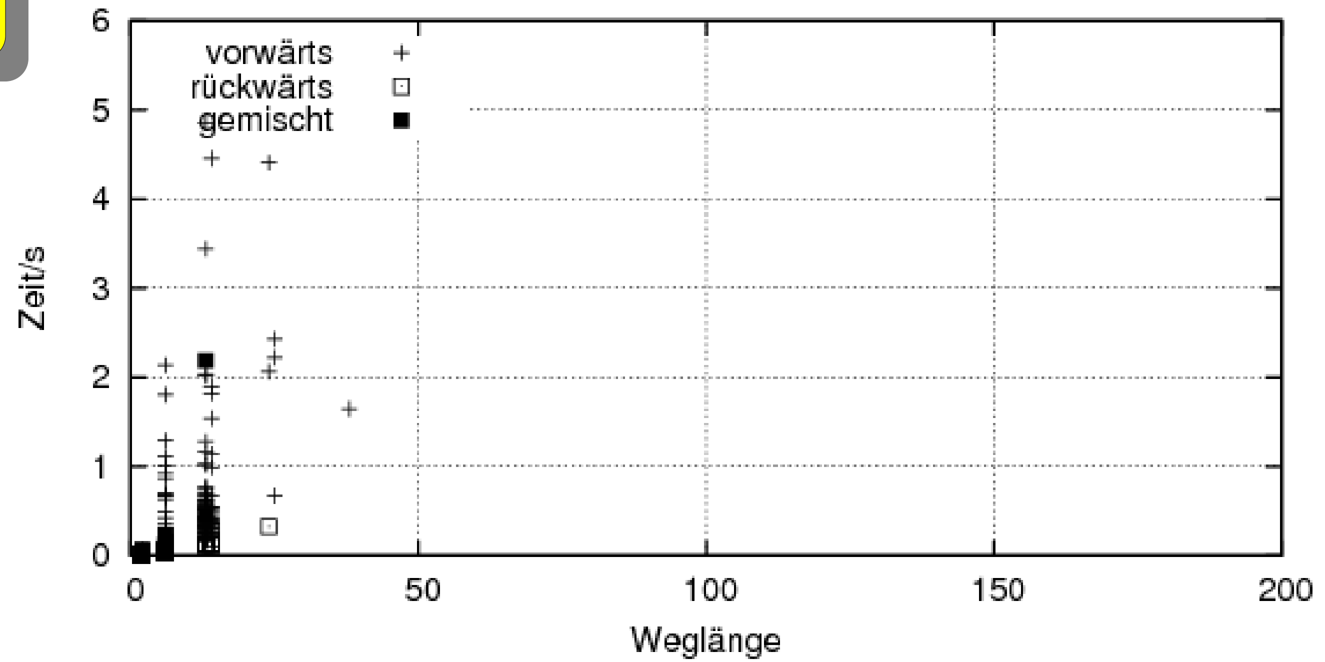
No Prediction

Timeouts for backward and mixed stepping directions



Quadratic increase due to inefficient memory model.

With Prediction



Length of array to sort is completely defined after first iteration of outside loop.

Comparison of Strategies

Example	Best Strategy (asympt. savings)		Worst Strategy
	No Prediction	With Prediction	
Fibonacci	Backward (ca. 49%)	Backward (ca. 46%)	Mixed w/ prediction
Selection Sort	Forward (0%)	n/a	Vorwärts w/ prediction
`strcmp` w/o `break`	Backward (n/a)	n/a	Mixed w/ prediction
`strcmp` w/ `break`	Mixed (ca. 10%)	Backward (ca. 28%)	Mixed w/ prediction
Array insertion	Mixed (ca. 7%)	Backward (ca. 68%)	Mixed w/ prediction

Conclusion

- **No optimal strategy**
- **No universally applicable strategy**

Actual CHR program sizes

Path Solver: 45 constraints, 76 rules (Many constraints for debugging or customised PCHR)

Built-in FD Solver: 26 constraints, 126 rules Optimised for detection of inconsistencies and domain filtering.

Both would not be handleable without CHR!

Conclusions

- Theory ≠ Practice in CHR
- Translation to CHR not straight-forward
- But: CHR-Program easier to read than manual implementation
- Extensibility due to CHR modularity

Outlook

- Complete implementation for C under way
- Enhancement of generic memory model
- Enhanced data-flow prediction
- Integrate results from abstract interpretation
- Worst-Case Execution Time analysis
- Evaluate CHRiSM!

Closing Comments

In theory, theory and practice are the same.

In practice, they are not.

CHR may be capable of being a general purpose language, but it is most useful as a special purpose language.

Questions?

