Automatic Test Data Generation and Model Checking with CHR

Ralf Gerlich, BSSE

Presentation for the
Eleventh International Workshop on Constraint Handling Rules
CHR 2014

July 18th, 2014
Vienna, Austria

BSSE System and Software Engineering

Advanced Software Technology
Consulting & Development
Technology & Management

Dr. Ralf Gerlich

Diplom-Informatiker
ralf.gerlich@bsse.biz

BSSE System and Software Engineering
Auf dem Ruhbühl 181
D-88090 Immenstaad
Germany

Phone: +49 7545 911258
Telefax: +49 7545 911240
Mobile: +49 178 76 06 129
www: http://www.bsse.biz/

Contents

- Motivation
- Path Construction
- Constraint Solver Approach
- Example of Use
- Open Problems
- Outlook/Conclusions

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

Contents

Test Input Selection

Given a sequence 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.

Example Test Goals:

- **„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

Design Goals

Verifiability and Comprehensibility:
Easy to prove in theory and simple to implement in practice

Performance:
Test data must be found in „acceptable“ time (not necessarily polynomial)

Avoidance of Bias:
Method should not favour one set of possible solutions over others

Automation:
No manual intervention necessary in solution process

Philosophy

In theory, problems are more generic.

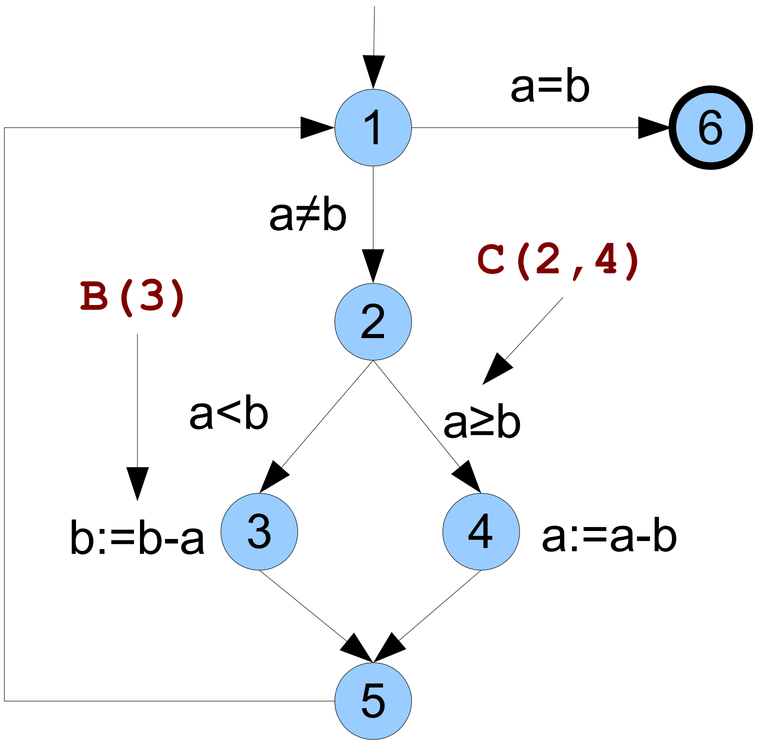
Genericity may mean theoretical absence of a solution (Halting Problem).

In practice, problems are more complex.

Complexity may mean absence of an efficient solution.

The trick is to find a solution for the practical problems without that realm being accurately defined!
⇒“Practice in the loop”

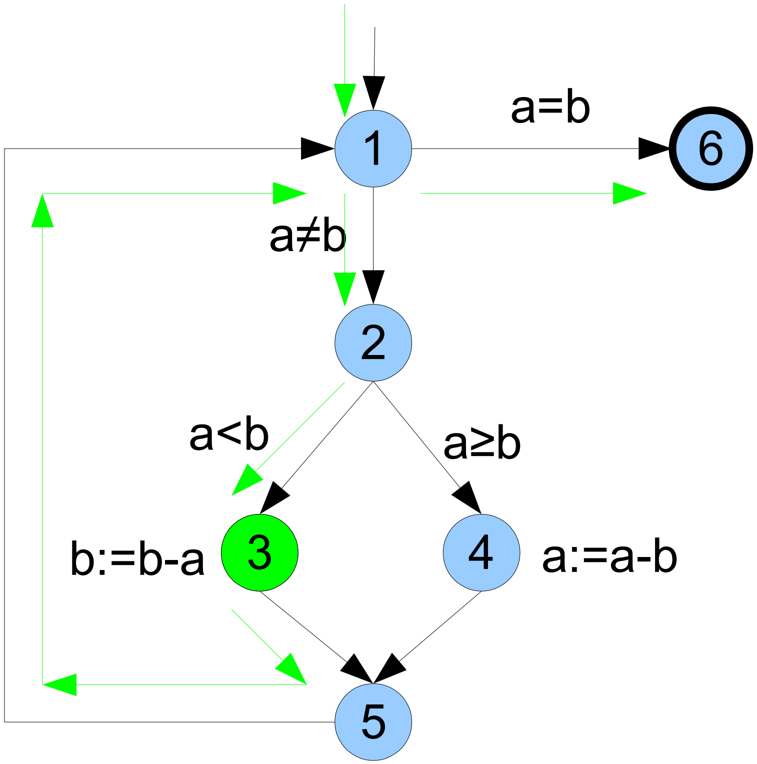
Augmented Control-Flow Graphs



- Nodes and Edges describe possible control flow
- Execution of nodes modifies program state
- Selection of edges by a set of predicates
- Boolean expressions are atomic (no conjunction, disjunction or negation, no side-effects)

Path Constraints: Forward Construction

Constructed Path:



Path Constraint:

$$a_1 \neq b_1$$

$$a_1 < b_1$$

$$b_2 = b_1 - a_1$$

$$a_1 = b_2$$

Solved Form: $b_1 = 2a_1 \wedge 0 < a_1$

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.

⇒Interleave path construction and satisfiability checking to avoid infeasible paths.

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

Constraint Theories to Combine

- Arithmetics and Relations over
  - Integers (Modulo-Arithmetics)
  - Floats (IEEE 754, various precisions)
- Bitwise Operations (AND, OR, XOR, Shifts)
- Addressable Memory
  - Integer Adresses
  - Various Types and Word Sizes
- Conversions

Linear Constraints

- Presburger Arithmetic
  - Symbolic Variables
  - Multiplication by Constants
  - Addition of Presburger Terms
- Relations: <, >, ≤,≥,≠,=
- Usual Approach:
  - Equations: Gaussian Elimination
  - Inequations (<, >, ≤,≥): Fourier-Motzkin-Elimination
  - Negated Equality (≠): Split up (<>).

Linear Integer Constraints

- Usual Approach does not work
  - Gaussian Elimination: Integers not closed under division (divisor$\neq$0)
  - Fourier-Motzkin Elimination: Integers are not compact
- Different Approach: The Omega-Test
  - Originally used for static aliasing analysis (e.g. in compilers)

The Omega Test

Implicit assumption: $a, b \in \mathbb{Z}$

Solution of Equations by Parameterisation:

$$3a - 2b = 0 \longrightarrow a = 2\alpha \wedge b = 3\alpha, \alpha \in \mathbb{Z}$$

Processing of Inequations by Over-Approximation

$$2b \leq 3a \wedge 2a \leq 3b \xrightarrow{\text{Eliminate a}} 2 \leq 5b \Leftrightarrow 1 \leq b$$

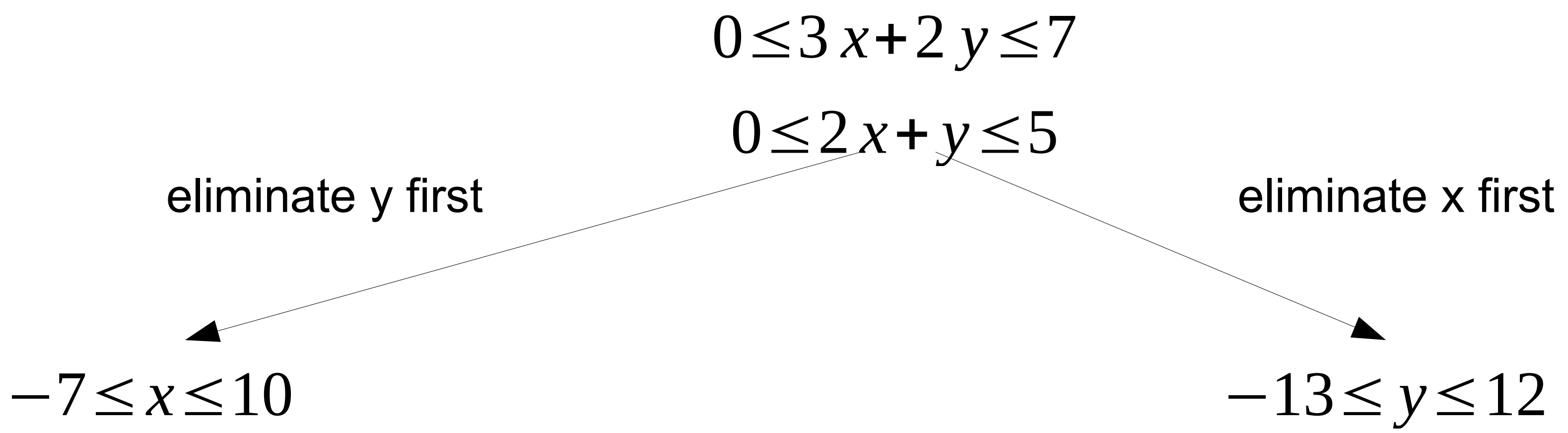
Any matching value of *b* will lead to a non-empty range of *a*, but

$$0 \leq 5b < 2 \Leftrightarrow b = 0$$

may lead to solutions as well (→exhaustive search).

W. Pugh: *The Omega Test: a fast and practical integer programming algorithm for dependence analysis*, Comm. of the ACM Vol. 8, pp. 102-114, 1992

The Omega-Test: Accuracy



$$0 \leq 3x + 2y \leq 7$$

$$0 \leq 2x + y \leq 5$$

eliminate y first

eliminate x first

$$-7 \leq x \leq 10$$

$$-13 \leq y \leq 12$$

Inaccurate!
<u>Possible</u> additional solutions:

$$-15 \leq y < -13$$

- Efficiency depends on order of elimination
- Best possible order may change online
  - Unification of Variables
  - New Inequations, New Variables

Floating Point Constraints

- Discrete, finite set of values
- Representation: Sign, Mantissa, Exponent
- Even linear operations are non-linear!
- 4 rounding modes
- 6 operations have unique results (IEEE 754)
    - Basic arithmetics (+,-,*,/)
    - Remainder
    - Square-Root
- Others are platform-dependent
- Current solvers not fast enough

CHR Experience

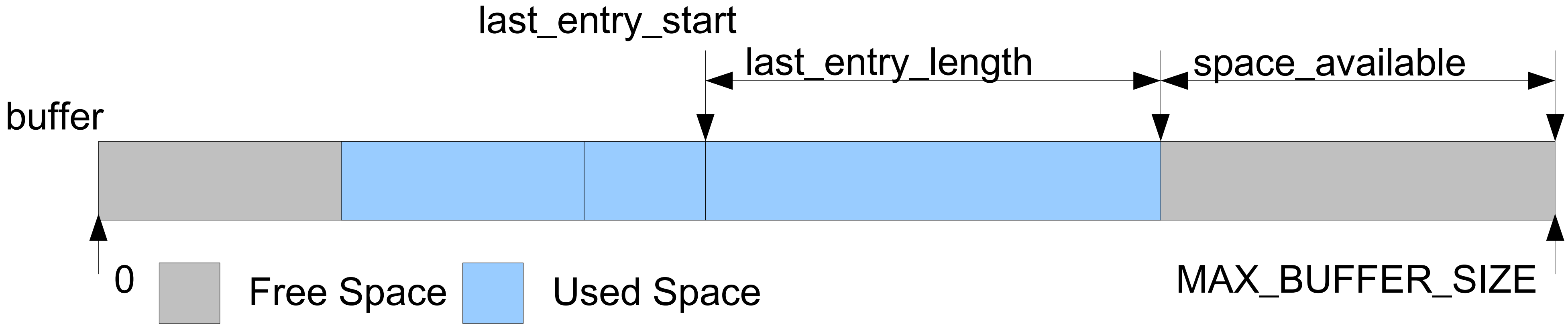
- Different aspects can be kept separate
  - Declarative nature of CHR
  - Compiler does the integration work
- Integration: mostly just another rule
- Global solution strategies require „hacks“
  - e.g. for (re-)ordering in the Omega-Test
  - Breaking declarational-operational link

Example: Context

- Satellite S/W had already been tested
  - Normal testing by S/W provider
  - Indepentend Software Verification and Validation (ISVV)
  - Static Analyzers had been used
- Study on effectiveness of random testing
  - Heuristic Oracles: crashes, timeouts, …
  - Instrumentation
  - Massive Stimulation (~10^3 stimuli per function)

Example: The Idea

- Satellite receives S/W patch
  - Stored into buffer
  - Flash Update is scheduled
- At scheduled time:
  - Data retrieved from buffer
  - Stored in Flash
- New packets may arrive before buffer empty



Example: The Code (simplified)

```
#define MAX_BUFFER_SIZE ...

char buffer[MAX_BUFFER_SIZE];

void store_into_buffer(char* data, unsigned int length) {
    const unsigned int last_entry_start = ...;
    const unsigned int last_entry_length = ...;
    unsigned int next_entry_start, space_available;

    next_entry_start = last_entry_start+last_entry_length;

    if ((MAX_BUFFER_SIZE - (length-1u)) < next_entry_start)
        next_entry_start = 0;

    space_available =
        (last_entry_start - next_entry_start) % MAX_BUFFER_SIZE;

    if (space_available >= length)
        memcpy(&buffer[next_entry_start],data,length);

    ...
}
```

Store block at start of buffer when not enough space at the end

Stimulation with random data led to crash here!

Example: Difficulties

- Reason for exception not obvious
    - Two experts, two opinions
- Overflow in C is not a runtime failure
    - Wraparound: Modulo-2^n-Arithmetics
    - Suspected to be the culprit here
    - Manual analysis error-prone
- The code was not as simple as shown here
    - Many paths to the target location
    - Additional calculations
    - Hindsight: None of them was relevant!

Example: The Query

```
   #define MAX_BUFFER_SIZE ...

   char buffer[MAX_BUFFER_SIZE];

   void store_into_buffer(char* data, unsigned int length) {
ⓐ       const unsigned int last_entry_start = ...;
        const unsigned int last_entry_length = ...;
        unsigned int next_entry_start, space_available;

        next_entry_start = last_entry_start+last_entry_length;

        if ((MAX_BUFFER_SIZE - (length-1u)) < next_entry_start)
             next_entry_start = 0;

        space_available =
             (last_entry_start - next_entry_start) % MAX_BUFFER_SIZE;

        if (space_available >= length)
ⓑ            memcpy(&buffer[next_entry_start],data,length);

        ...
   }
```

Is there a path from a to b so that

next_entry_start+length>MAX_BUFFER_SIZE

at b?

Answer: YES!

Example: The Bug

```
    if ((MAX_BUFFER_SIZE - (length-1u)) < next_entry_start)
        next_entry_start = 0;
```

One-off-mistake allows one-byte overflow!

- Fault conditions from constraint store
    - – next_entry_start+length=MAX_BUFFER_SIZE +1
    - – Verification of solver result
    - ➔ Bug Report
- ➔ Data corruption
    - – Later: Code corruption in Flash
    - ➔ Disruption of Service
    - – No permanent failure →„Safe Mode“

Example: The Aftermath

- Now: Systematic index checking in C
  - By instrumentation during test runs
  - Ada had this as a compiler option!
- More similar defects found
- Possibly problem from porting (Ada → C)
  - Ada: Arbitrary start of array indices
  - C: indices start at 0
- Did static analysers not find it?
  - Unknown...

Open Problems

- Performance
  - Problem is inherently complex
  - Many constraints over few variables (input parameters)
- Floating Point Arithmetics
  - Current Approach: Domain Filtering
  - Slow in reaching fix-point
  - Platform Dependency (sin, cos, ...)

Possible Solutions

- Slicing, Lazy Evaluation
  - Only consider constraints that contribute to decisions
  - Reduces number of floating-point constraints in practice
  - But: Aliasing problem
- Filtering Speed
  - Stop filtering once domain reduction less than defined bound

Conclusions

- CHR well-suited for implementation of complex constraint solvers
  - Applicable also to model checking on source-code level
- Declarative Semantics aids verification
- Global strategies often break link between declaration and operation
- Further research required for open problems

Outlook

- Industrial research on open issues ongoing
- One step at a time
  - Ignore theoretical limitations if not relevant in practice (Halting Problem)
  - Small improvements better than big theories

Questions?

Ongoing industrial research at BSSE is supported by a grant by the German federal government under grant number 50RA1339.

Backup

Generic Path Constraint Relations

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

$\longrightarrow$

spec(A,B,X,Y)

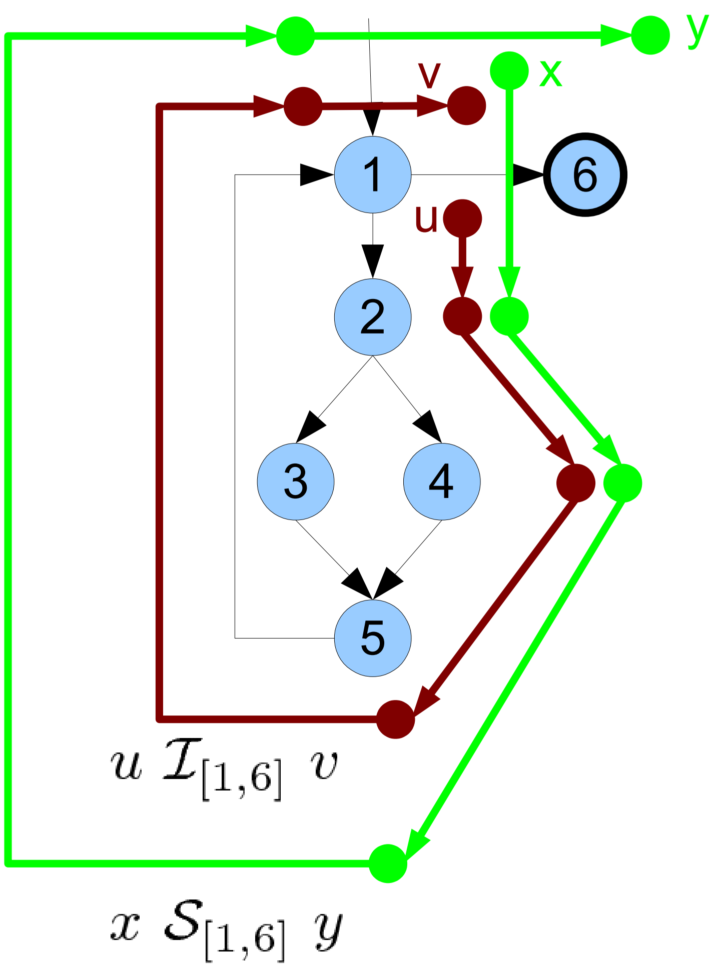
There is a path from node `a` to node `b`, transforming input `x` to output `y`. ("Specification")

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

$\longrightarrow$

ispec(U,A,B,V)

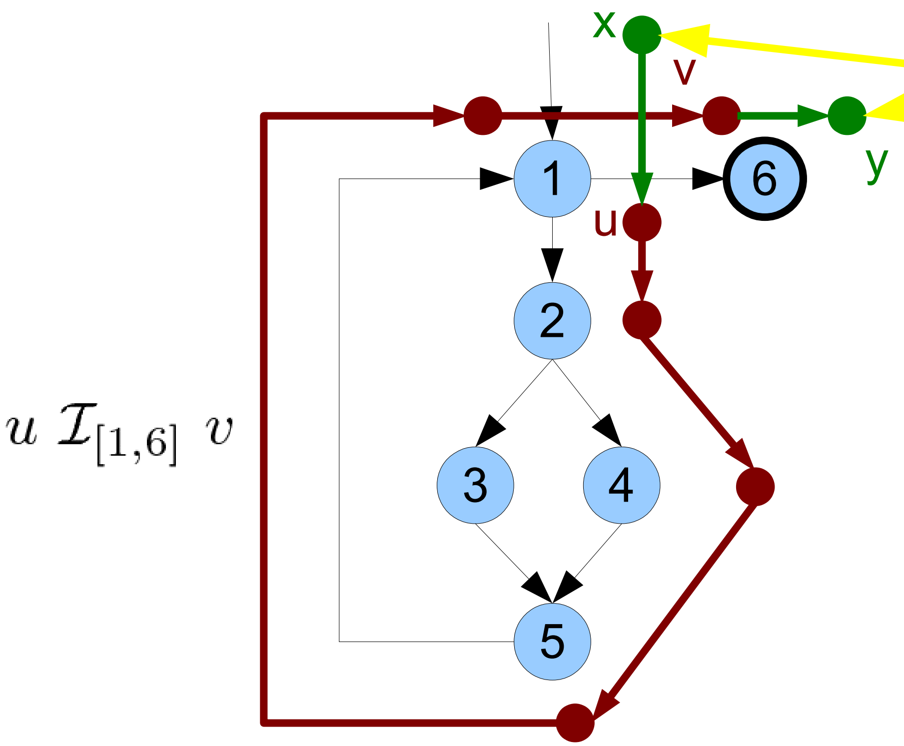
There is a path from node `a` to node `b`, with `u` the output of node `a` and `v` the input of node `b`. ("Inner Specification")



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.

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

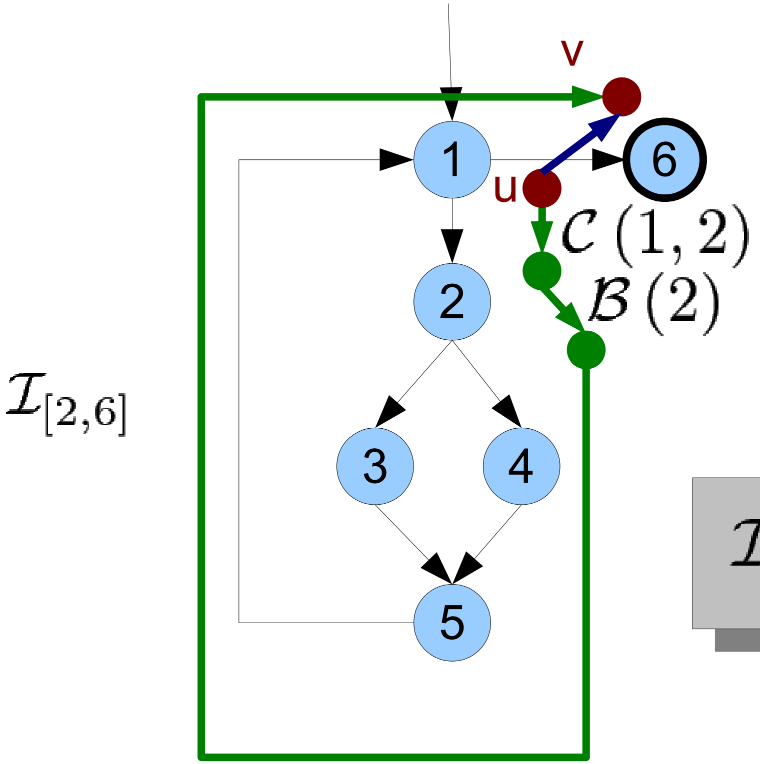
(Read concatenation right-to-left)

But:

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

```
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

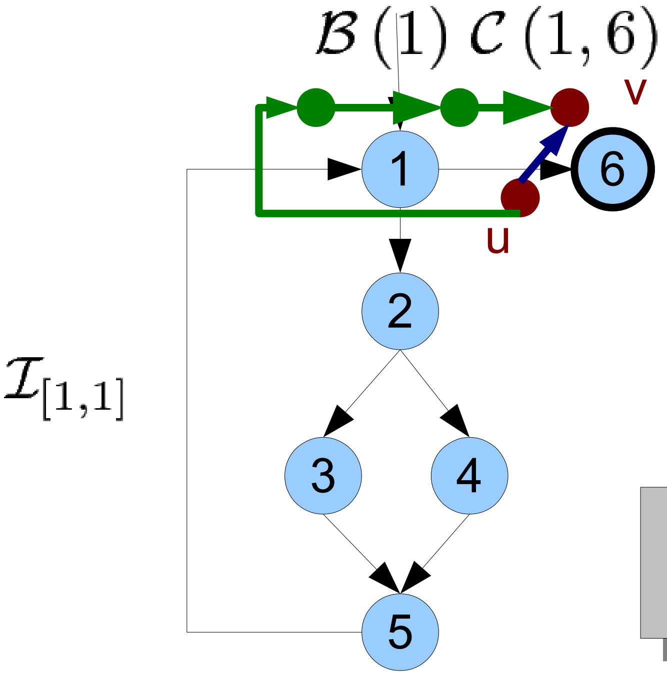


We can either traverse the edge from 1 to 6 or continue from 1 via 2 to 6.

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

```
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

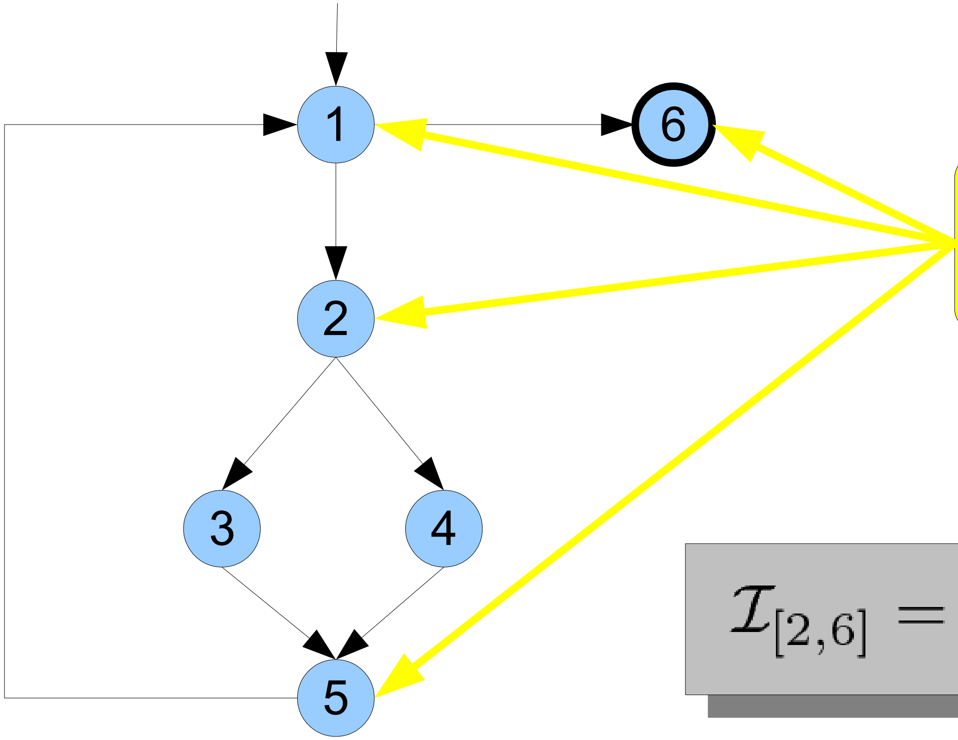


We can either traverse the edge from 1 to 6 or continue from 1 via 1 to 6.

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

```
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)).
```

Control-Flow Prediction



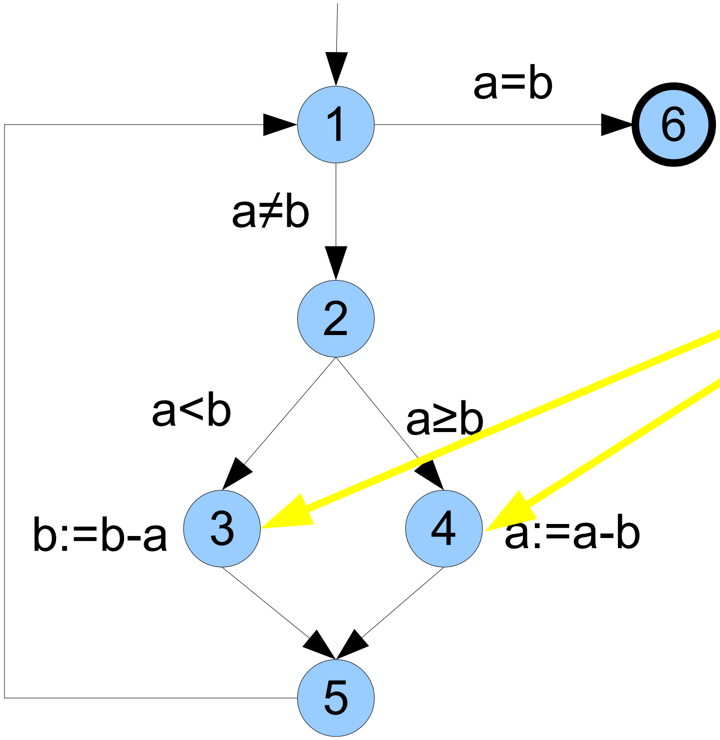
All paths from 2 to 6 traverse these nodes

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

```
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)).
```

ispec(X,U,V,Z), body(V,Z,Y), cond(V,W,Z)).

Complete? Not so fast!

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

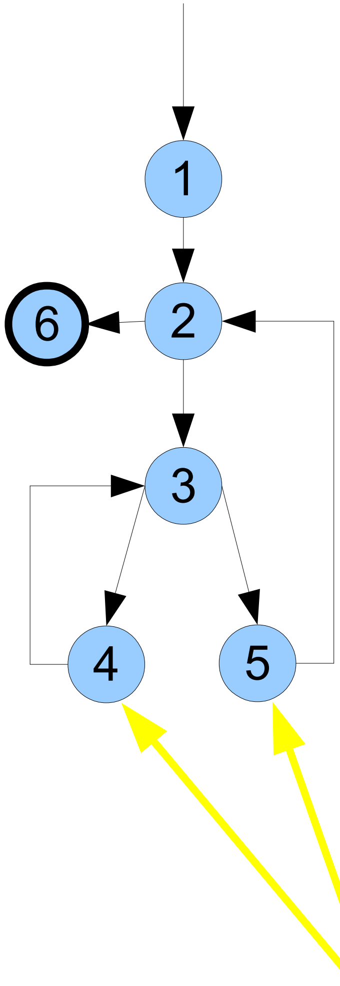
Swapping of alternatives could lead to infinite recursion.

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.

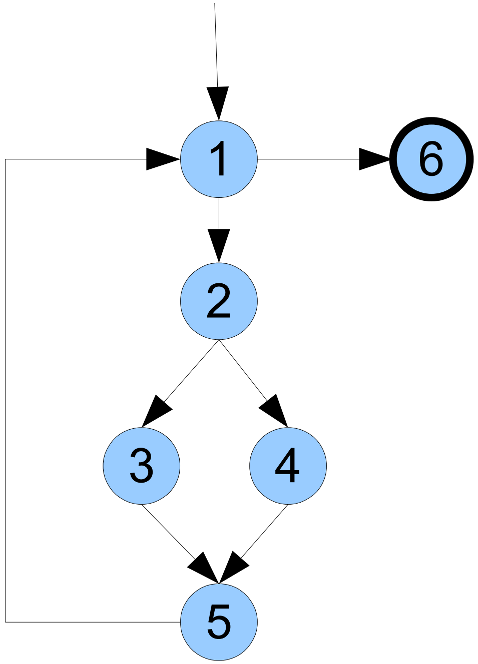
There are almost always more instances of the “step” alternative than of the “edge” alternative.
The statistical model for that is difficult to manage.

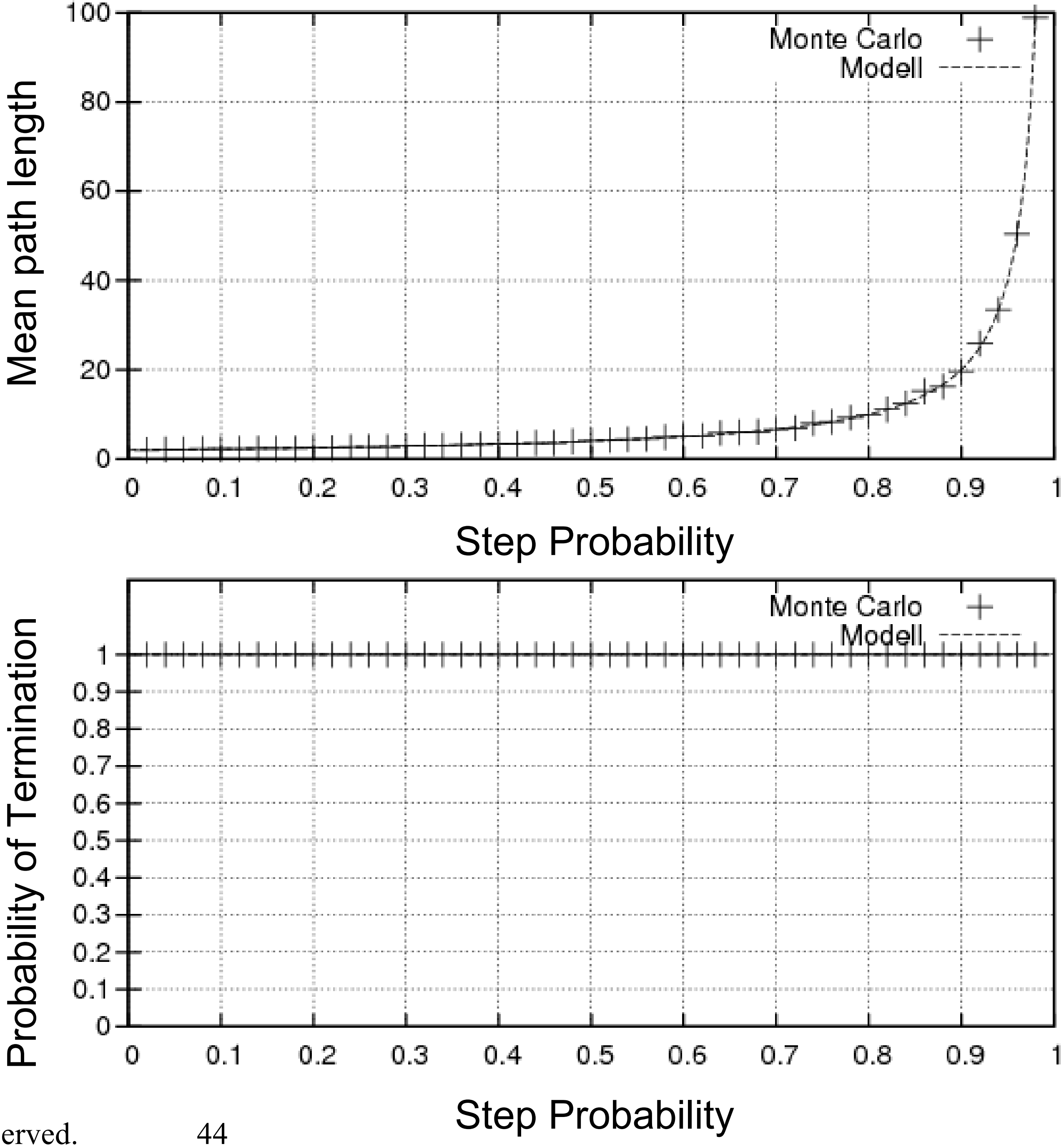
PCHR: uncontrollable path growth

Solution: CHRiSM

Yes, we'll try that!

Evaluating the Statistical Model



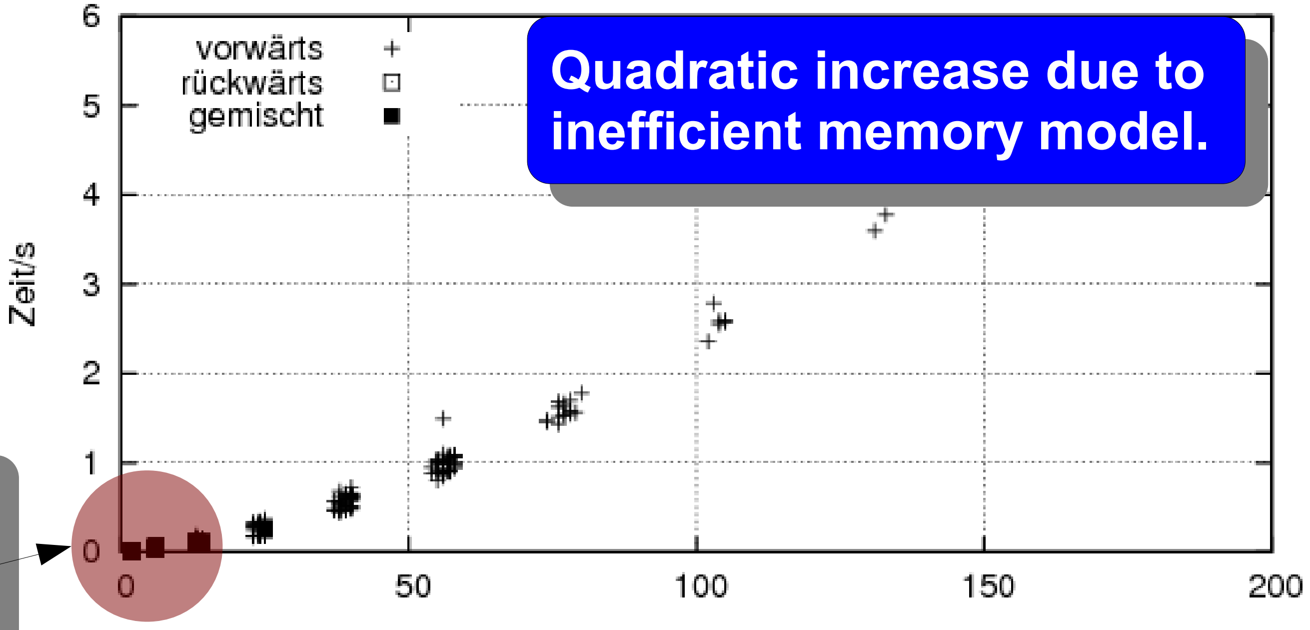


Main Discoveries:

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

Runtime Complexity on Selection Sort

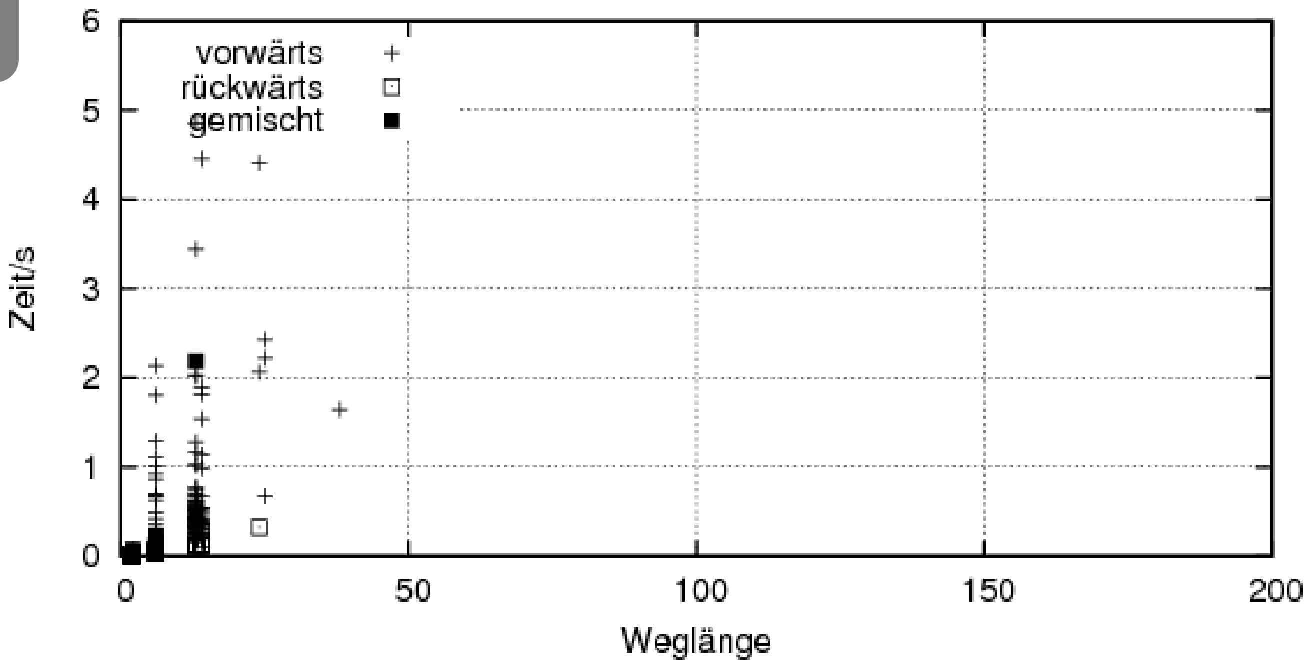
No Prediction



Quadratic increase due to inefficient memory model.

Timeouts for backward and mixed stepping directions

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	Forward 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!