cadence

INVENTIVE

Constraint-Based Test Generation

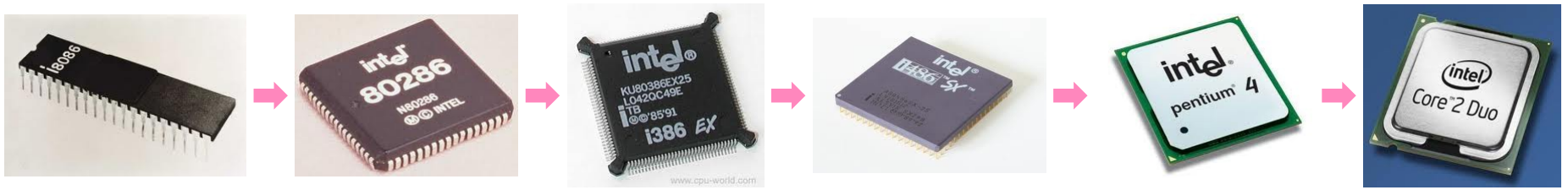
Vitaly Lagoon
(lagoon@cadence.com)

A Word About Cadence

- Cadence
  - One of the three major EDA (Electronics Design Automation) companies
  - Established in 1988, over 4000 employees
  - Wide variety of technologies and products in
    - Hardware design (inc. logic synthesis, power, timing, routing, etc.)
    - Hardware simulation
    - Formal and simulation-based verification
  - Never heard about us? You surely heard about our customers.
    - Intel, Cisco, TI, Canon, Phillips, Samsung, Nokia, …

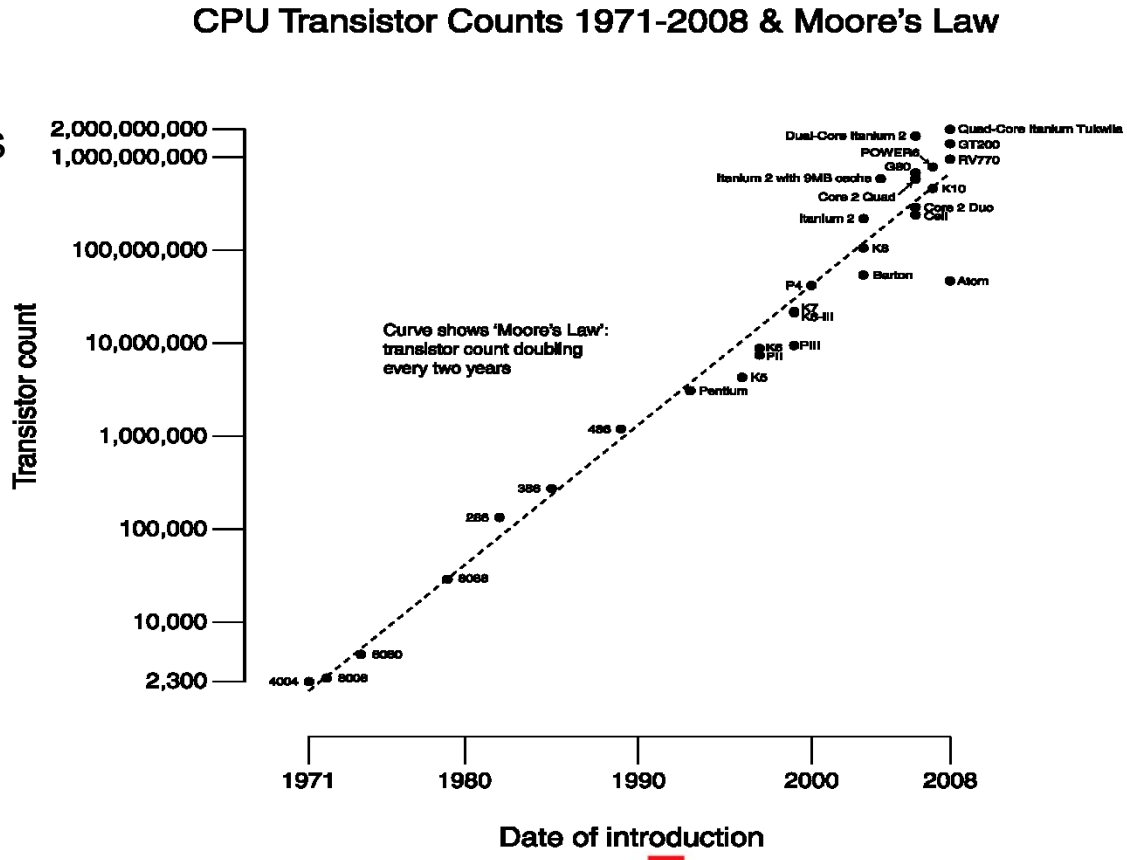
cadence®

Motivation



Chip complexity grows, and so does

- the likelihood of HW bugs
- the cost of HW bugs
- the need for verification
- investment in verification



cadence

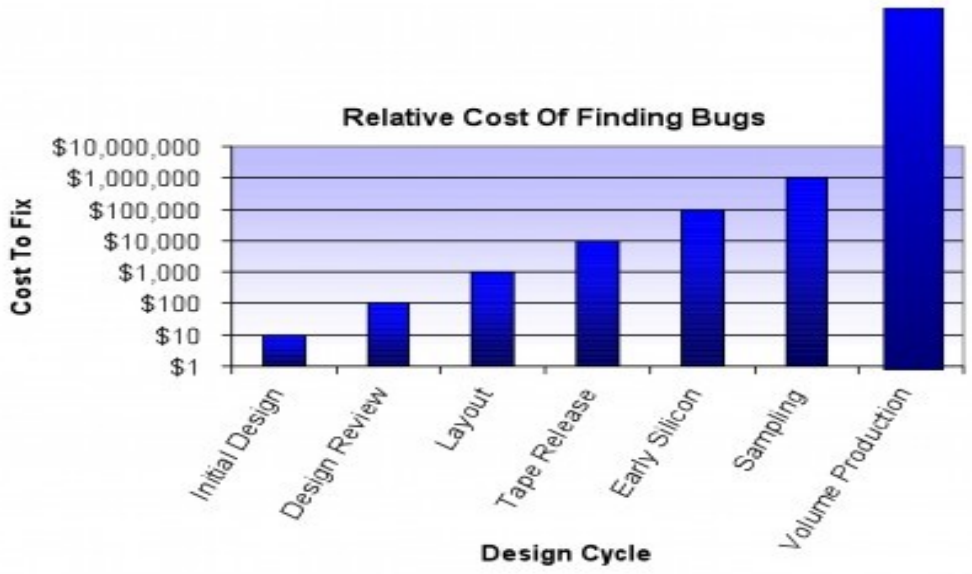
The Cost of HW Bugs

- **Intel FDIV, 1994, ~$475M**

$$\frac{4195835}{3145727} = 1.3338204491362410 02$$

$$\frac{4195835}{3145727} = 1.3337390689020375 89$$

- **Intel Sandy Bridge, 2011, ~$1B**



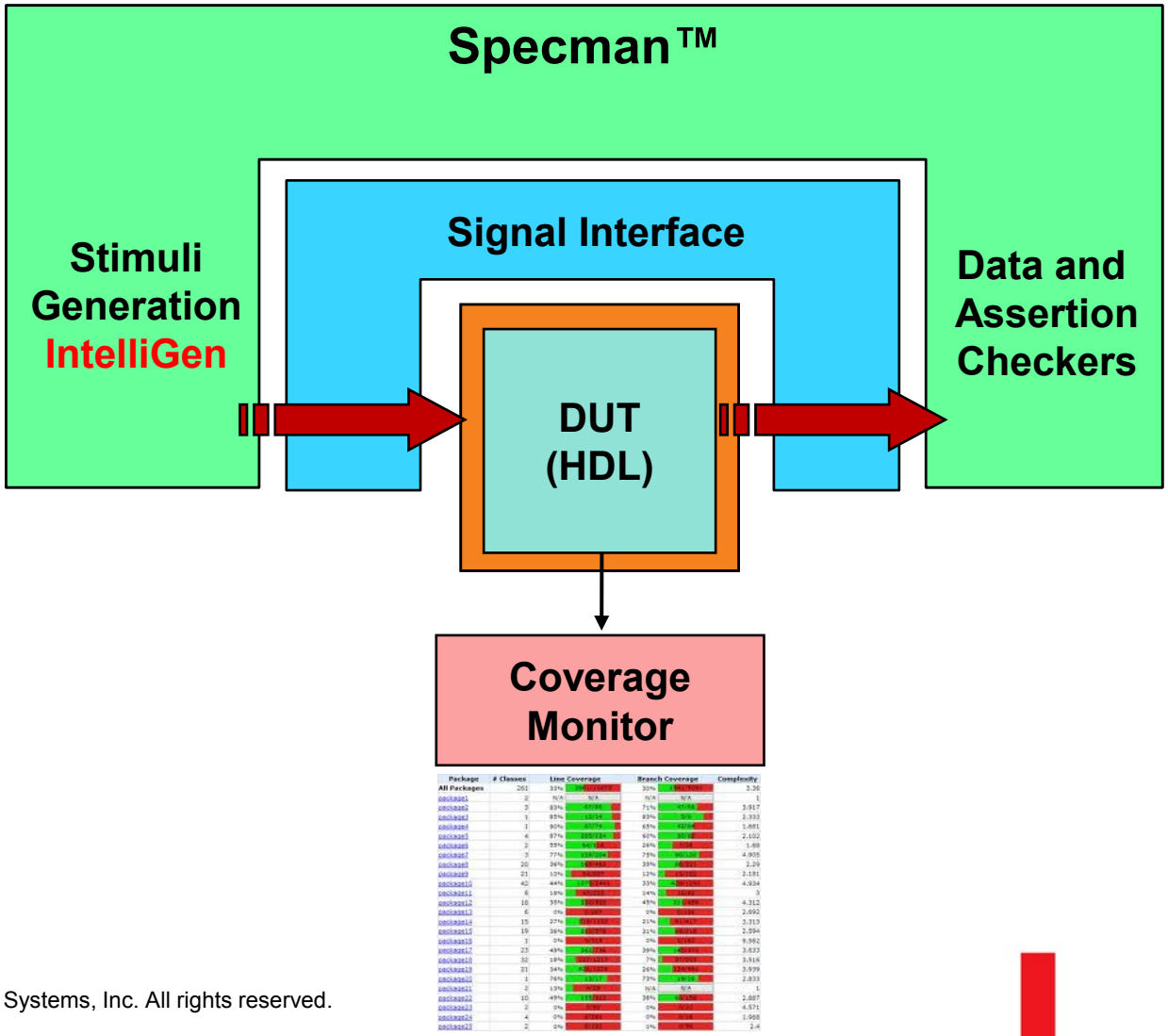
Silicon Debug, Doug Josephson and Bob Gottlieb, (Paul Ryan)
D. Gizopoulos (ed.), Advances in Electronic Testing: Challenges and Methodologies, Springer, 2006

cadence

Formal vs. Simulation

- Formal verification
  - \+ *Proves* properties of a HW design
  - \+ Substantial improvements in performance, capacity and scalability in the last few years
    - ✓ Improvements in SAT solvers
    - ✓ New approaches using abstraction and Solving Modulo Theories (SMT)
  - ✘ Cannot verify (yet?) a full system, only individual units
  - ✘ Verification environment cannot be reused for post-silicone testing
- Simulation-based verification
  - \+ High capacity and scalability
  - \+ Verification environment can be reused for post-silicone testing
  - \+ Easy to use
  - ✘ Experimentally verifies properties of a HW design
- ~80% of bugs in HW logics are still found through simulation

Verification by Simulation



cadence

Specman

- Cadence's major test bench automation tool
- Being used in the biggest and most advanced verification environments
- Works with all HDL simulators
- Uses ***e*** verification language [Hollander,Morley,Noy '01]

http://en.wikipedia.org/wiki/Specman

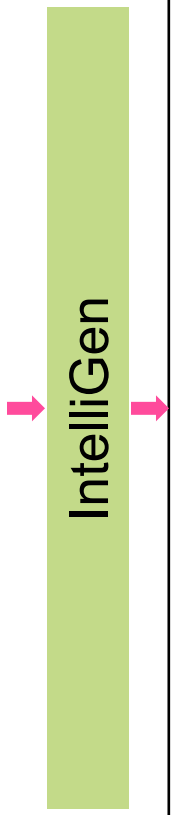
IntelliGen

- Constraint solver / test generator of Specman
  - Generates randomized tests based on constraint models
- Variety of data types:
  - signed/unsigned numeric, Boolean, string, arrays, pointers
- Variety of constraints:
  - arithmetic, logic, bit-wise, soft constraints, global constraints on arrays
- Based on an FD-core
  - Also integrates a few variations of BDD and SAT
- Includes a visual constraint debugger

Random Test Generation in Specman/e

```
struct CPU_instruction {
    opcode  : [LDA, STA, ADD, SUB,
               JMP, JGE, JNE, STP](bits:4);
    operand : uint(bits:12);
    keep opcode == STA => operand > 1023;
};

struct CPU_test {
    program : list of CPU_instruction;
    sz : uint[10..20];
    keep program.size()==sz;
    keep program[sz-1].opcode==STP;
    keep for each (instr) in program {
        index<sz-1 => instr.opcode != STP;
        instr.opcode in [JMP, JGE, JNE] =>
           instr.operand<sz;
    };
};
```



item	type	opcode	operand
0.	instruction	JMP	6
1.	instruction	JMP	11
2.	instruction	ADD	2301
3.	instruction	JGE	1
4.	instruction	SUB	312
5.	instruction	LDA	2603
6.	instruction	JMP	7
7.	instruction	JGE	11
8.	instruction	JNE	4
9.	instruction	STA	3913
10.	instruction	JMP	12
11.	instruction	SUB	1783
12.	instruction	LDA	2035
13.	instruction	ADD	1310
14.	instruction	JNE	15
15.	instruction	JMP	12
16.	instruction	LDA	3258
17.	instruction	STP	3964

cadence

IntelliGen.
The Requirements

- Powerful and flexible constraint language
  - Mixed integer and bitwise models
  - Mixed declarative and procedural code in models (function calls)
  - Non-scalar data: structures, arrays, strings
  - Soft constraints for modeling preferences
  - Directives for controlling randomness and distribution
- High scalability
  - Huge models (hundreds of thousands variables and constraints)
  - Solving the same problem many times (long runs)
- Find many random solutions
  - Try to meet distribution requirements
  - Be *random-stable* as much as possible
- Typical problems are not hard
  - Extensive search is usually not required
  - Backtracks typically indicate bad modeling

IntelliGen.
The Integrated Framework of Solvers

- There is no “best” solving technology.
  - A combined approach is required in practice
- BDD
  - \+ Bit-level, fast, complete control over the distribution
  - ✘ Capacity problems
- SAT
  - \+ Fast, scalable
  - ✘ Translation to CNF is expensive, limited control of randomness
- Finite-Domain solver
  - \+ Word-level, fast, scalable, extendable
  - ✘ Limited control over distribution
  - ✘ Bad in proving UNSAT
- Local Search / SMT / ILP… ?

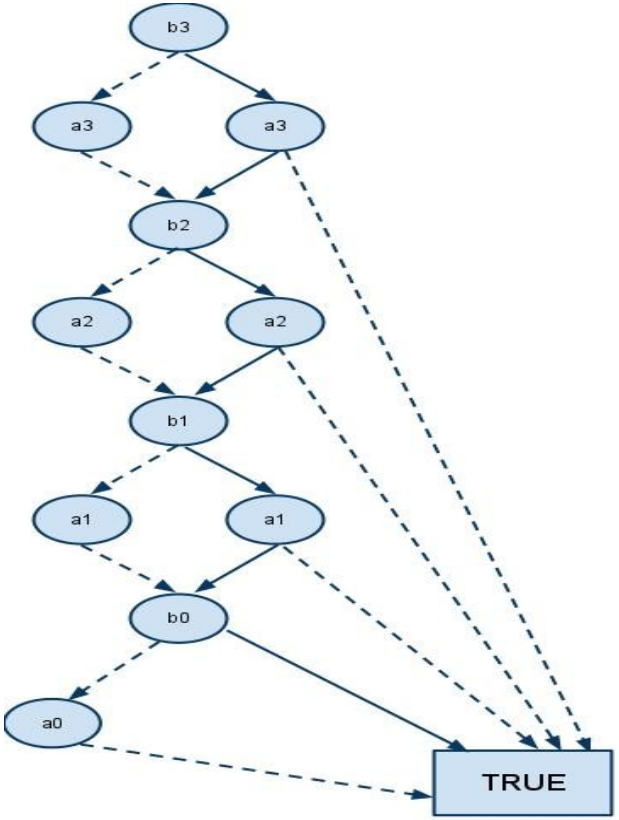
cadence®

Solving Technologies: BDD

- Build the BDD once
- Generate solutions as random walks from the root to TRUE

+ Good performance when generating many solutions
+ Complete control over the distribution

✘ Limited capacity
✘ Very sensitive to the order of variables
✘ Infeasible for some types of constraints



$\langle a_3,a_2,a_1,a_0 \rangle \leq \langle b_3,b_2,b_1,b_0 \rangle$

cadence

Solving Technologies: SAT

- Convert the constraint model to a CNF
- Give it to a state-of-the-art SAT solver

+ High capacity
+ High performance
+ Improvements are free

✘ Randomization is tricky, no guarantee of distribution
✘ High bootstrap cost
✘ Loss of high-level context:
  – Role and association of bits
  – No GAC (loss of propagation) e.g., all-different [Bessiere, Katsirelos, Narodytska, Walsh, 09]

$$
\begin{aligned}
&(\neg a_3 \vee b_3) \wedge \\
&(\neg a_3 \vee r_2) \wedge \\
&(b_3 \vee r_2) \wedge \\
&(\neg r_2 \vee \neg a_2 \vee b_2) \wedge \\
&(\neg r_2 \vee \neg a_2 \vee r_1) \wedge \\
&(\neg r_2 \vee b_2 \vee r_1) \wedge \\
&(\neg r_1 \vee \neg a_1 \vee b_1) \wedge \\
&(\neg r_1 \vee \neg a_1 \vee r_0) \wedge \\
&(\neg r_1 \vee b_1 \vee r_0) \wedge \\
&(\neg r_0 \vee \neg a_0 \vee b_0)
\end{aligned}
$$

$\langle a_3,a_2,a_1,a_0 \rangle \leq \langle b_3,b_2,b_1,b_0 \rangle$

cadence

Solving Technologies: FD Solver

- Maintain *domains* of variables as sets of intervals
  - x : [1..100], y : [1,3,5..8]; z : [1..100,1000..2000];
- Use propagators to enforce domain consistency
- Combine search and propagation
- *Randomize the choice of variables and values*

+ Cheap on simple problems
+ Scales very well to large problems
+ Word-level processing
  - Easy to extend: new constraints, global constraints, randomization policies
  - Easy to explain e.g., in constraint debugger

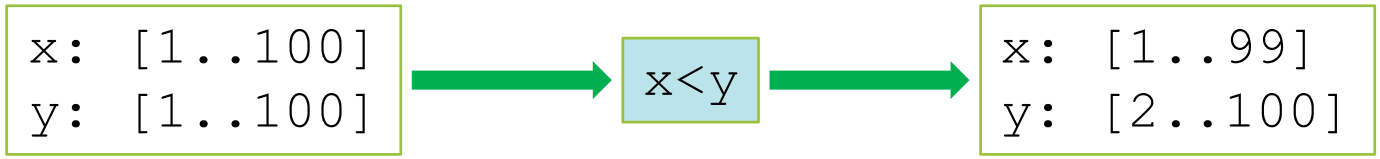
✘ Limited control over randomness and distribution
✘ Bad in proving UNSAT

IntelliGen's FD Solver

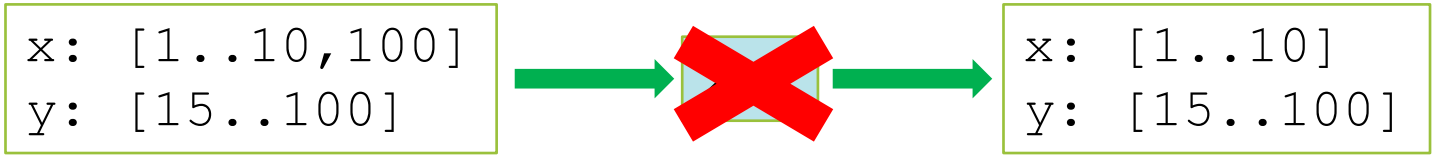
```
Solve(V,C) : [TRUE, FALSE]
    if all variables in V are assigned  then return TRUE
    choose an unassigned variable x in V
    repeat
        choose a value k from the domain of x
        if Propagate(V·[x/k], C)  &&  Solve(V, C)  then
            return TRUE
        else
            undo the last reduction
            remove k from the domain of x
    until range of x is empty
    return FALSE
```

Propagation

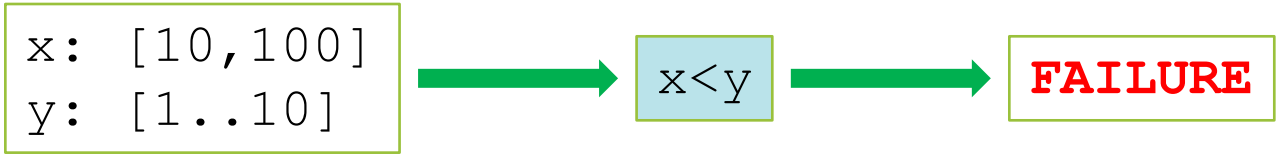
- Removes inapplicable values from domains



- Prunes the model by removing trivially satisfied constraints



- Fails if domains are inconsistent



cadence

More On Propagation

- A more interesting example

x:[1..100], y:[1..100], x<y && x*x<300 && x+y>40 ➡ x:[1..17], y:[24..100], x+y>40

- IntelliGen includes propagation algorithms for
  - Relations == != < > <= >=
  - Boolean logic **and or not** => ?:
  - Arithmetic + – * / %
  - Bitwise operations << >> | & ~
  - Arrays (global) constr. my_list.all_different()
  - Pointers, strings, etc. (p==p1 **or** p==p2 **or** p==NULL)

The Hybrid Domain Representation

- **Problem:** intervals are inadequate for representing bitwise information
  - The interval representation the domain of x is `[0..2,4..6,8..10,12..14…]`
  - interval representation has a billion fragments!!!
  - … and bitmask representation can't help!
- **Solution:**
  - Use a hybrid domain representation combining intervals and BDDs
  - Do lazy updates between the two

```
x: uint;
keep x[1:0] != 0b11;
```

BDD Propagation [Lagoon and Stuckey, CP'04]

- Assume a binary constraint c(x,y) represented by a BDD
- Assume two domains d(x) and d(y) of x and y
- The updated domains d'(x) and d'(y) are computed as

$$d'(x) = d(x) \wedge \left[c(x,y) \wedge d(y) \right]_x$$

$$d'(y) = d(y) \wedge \left[c(x,y) \wedge d(x) \right]_y$$

- The propagation fails if we get FALSE in any conjunction
- The propagator becomes redundant if

$$d(x) \wedge d(y) \Rightarrow c(x,y)$$

- Straightforward extension to any number of variables

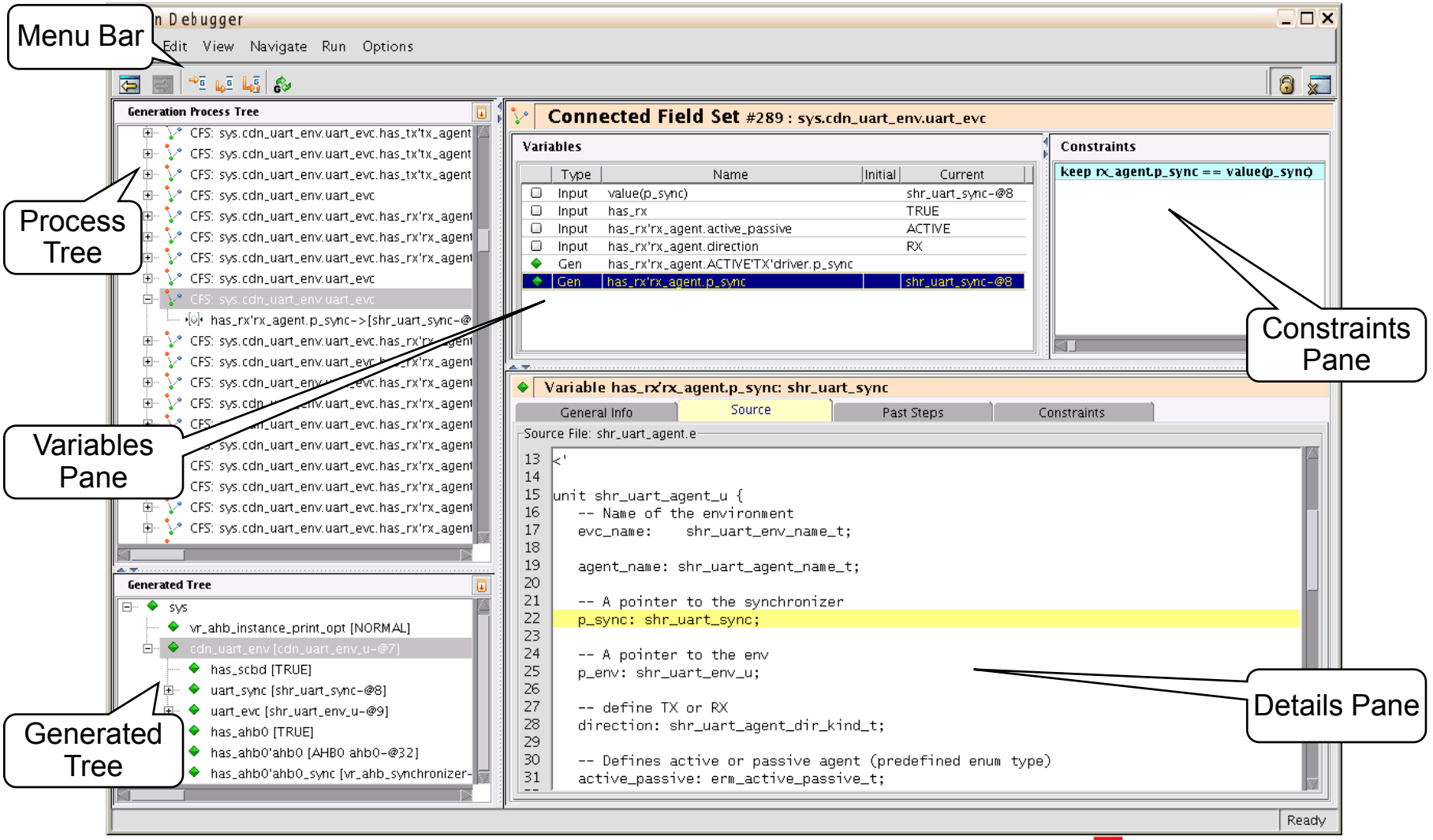
Many Refinements of the Search Mechanism

- Heuristic choice of variable/value
  - Take into account domain size, role, connectivity, etc.
- Smart graph-based backtrack mechanisms
  - Save only the relevant domains before each assignment
  - Save on backtracking through independent sub-graphs
- Restarts
- Local and global backtrack limits
  - Get out of unproductive corners of the hyperspace quickly
  - Stop and signal an error rather than take forever in search

Gen Debugger
[Alexandron, Lagoon, Naveh and Rich, HVC'09]

- The types of "constraint bugs"
  1. It can't find me a solution
  2. It finds a solution with unexpected values
  3. It never finds a solution with expected values
  4. It takes way too long/forever to find a solution
- The main principles of constraint debugging
  - Visualization
    - See the information you need in a clear and accessible way
  - Navigation
    - Get to the information you need
  - Minimization
    - See only the relevant information

Visualization



cadence

GenDebugger: Navigation

- Re-use the standard concepts of procedural debugging
  - Generation breakpoints / trace-points
    - Stop at any solving step, or at a specific partition, data type, object, field, variable, etc.
  - Step-by-step debugging
    - Step through atomic operations of the FD solver: propagation, assignment, backtrack
    - Walk into or walk over the generation of nested objects
    - Continue to the end of the context or to the next breakpoint
- GUI
  - See the hierarchy of objects, the time line of the generation, the relevant variables, and constraints, the source code
  - Access all constraints of a variable, all variables of a constraint, all past steps for a variable

GenDebugger: Minimization

- It is essential to minimize explanations:
  - In debugger, explaining propagations
  - In error messages explaining infeasibility
- Conservative minimization
  - Mark only the constraints that caused domain changes or failure
  - \+ Does not cost anything
  - \+ Sufficient in most cases
  - ✘ Explanations may have redundancy
- Aggressive minimization
  - Try to remove constraints one by one
  - \+ Produces a minimal set
  - ✘ Significant performance overhead

```
struct my_data {
    x : uint;
    y : uint;
    z : uint;
    keep x[4:0] == 0b11111;
    keep x<y;
    keep y<z;
    keep x+y<20;
    keep x+z<20;
};
```


cadence

Conclusion





- Verification is important
- Testing/simulation is (still) the main workhorse of verification
- CP is the backbone of today's test generation
- The requirements differ from the "classic" CP
  - Problems are often easy (not much search)
  - Problems are often HUGE (hundreds of thousands elements)
  - Many random solutions required
  - Need to support many data types, including non-scalar
- There is no "best" constraint solving technology
  - Need to combine FD, BDD, SAT, etc. in a unified framework
  - Make different technologies benefit from each other
- Constraint debugging and debuggers are necessary
  - But mostly overlooked by the research community


cadence

Questions

cadence®


cadence®