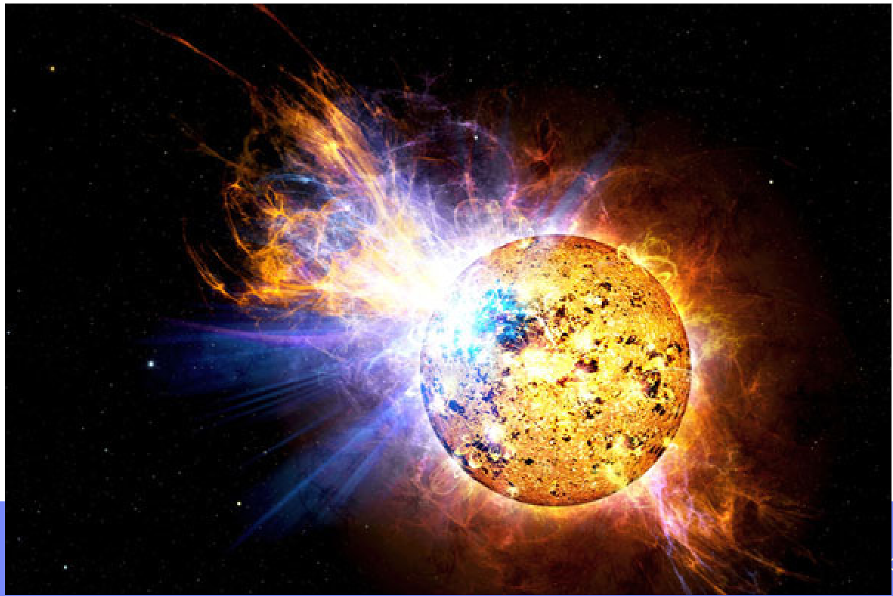
Challenges in Constraint Programming for Hardware Verification

Eyal Bin
IBM Haifa Research Lab
bin@il.ibm.com

Who are we?

- Verification Technologies Department (Simulation Based verification)
  - Part of IBM Research
  - Center of competence for verification technologies in IBM
  - Over two decades of experience in development of verification technologies for simulation based verification
    - Core level, System-level, Unit level



For this presentation, special thanks to:

Yeuda Naveh

Michal Rimon

Oz Hershkovitz

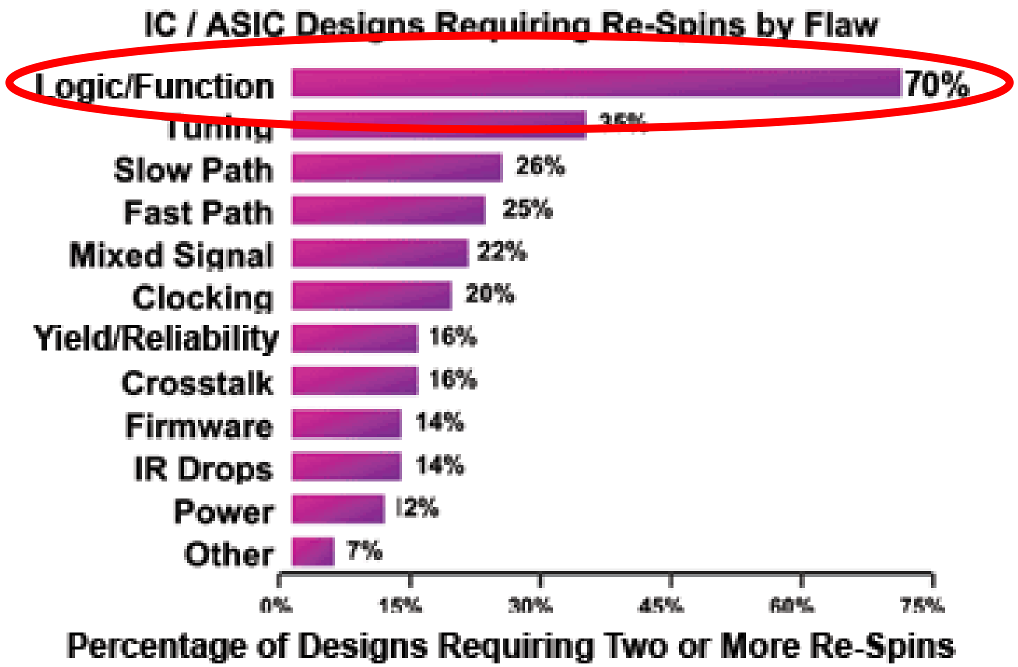
Ofer Peled

Yoav Katz

Wesam Ibraheem

The significance of functional verification

- Roughly 70% of the design effort (time, resources, …) is invested in functional verification
- Industry practice: verification == over 90% simulation based verification
- A design re-spin may cost many millions of $
  - Masks
  - Person-month
  - Time-to-market
- Typically 3-4 re-spins for complex designs (processors)



bin@il.ibm.com

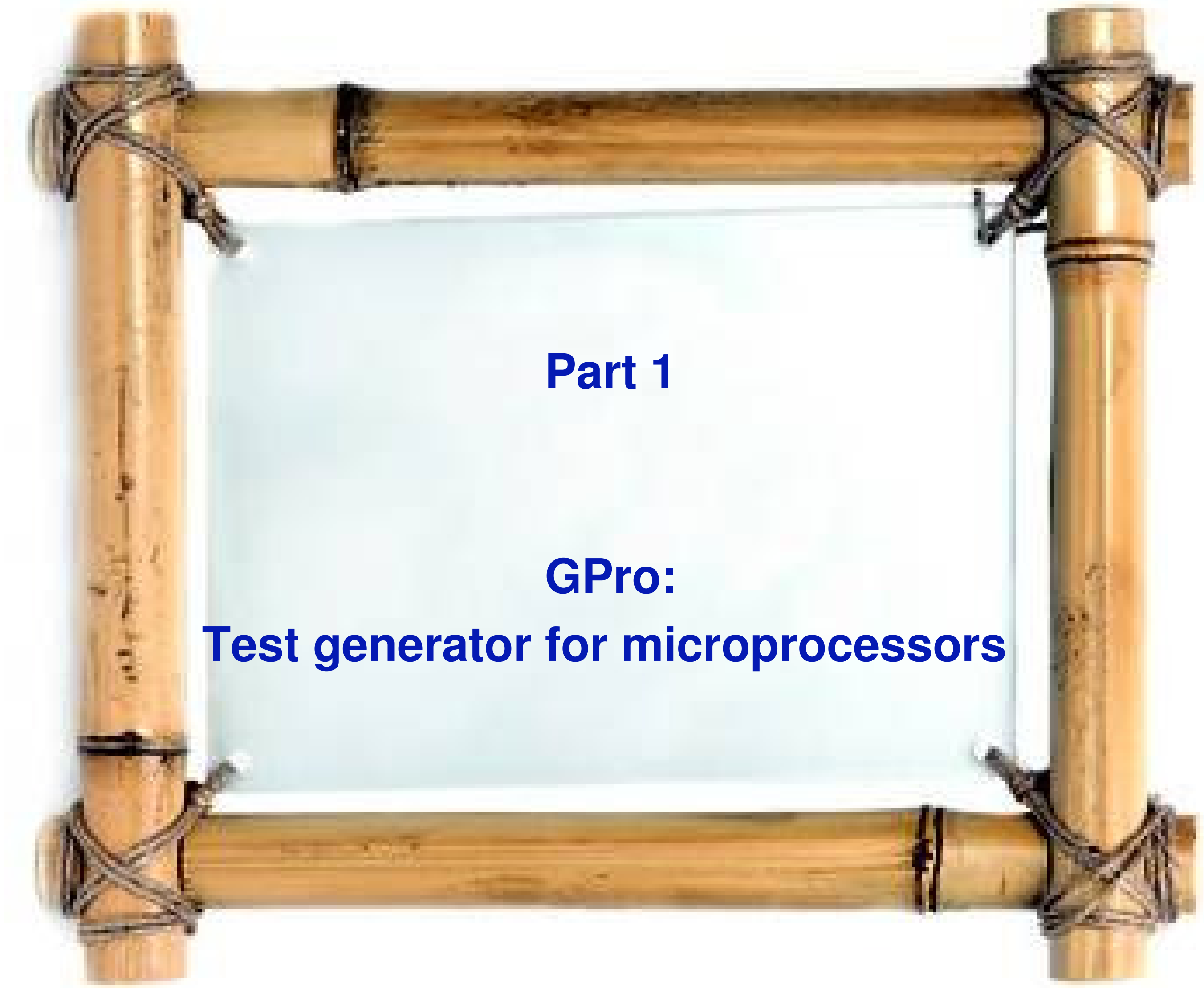
Key Technologies for Processor Verification

- Genesys-Pro
  - State-of-the-art test generator for full processor and multi-processor verification
  - Used by all IBM processors and licensed to external companies
    - Adaptable to any architecture
    - Applied in Power, zArch , ARM, and others
- FPGen
  - Dedicated generator focused on floating point verification
- XGen
  - A test generator for verification of systems
- ThreadMill
  - Post-Silicon and emulation exerciser

bin@il.ibm.com

Content

- Part 1: GPro - Test generator for microprocessors
- Part 2: CSP characteristics and challenges
- Part 3: PRB – The new CSP approach



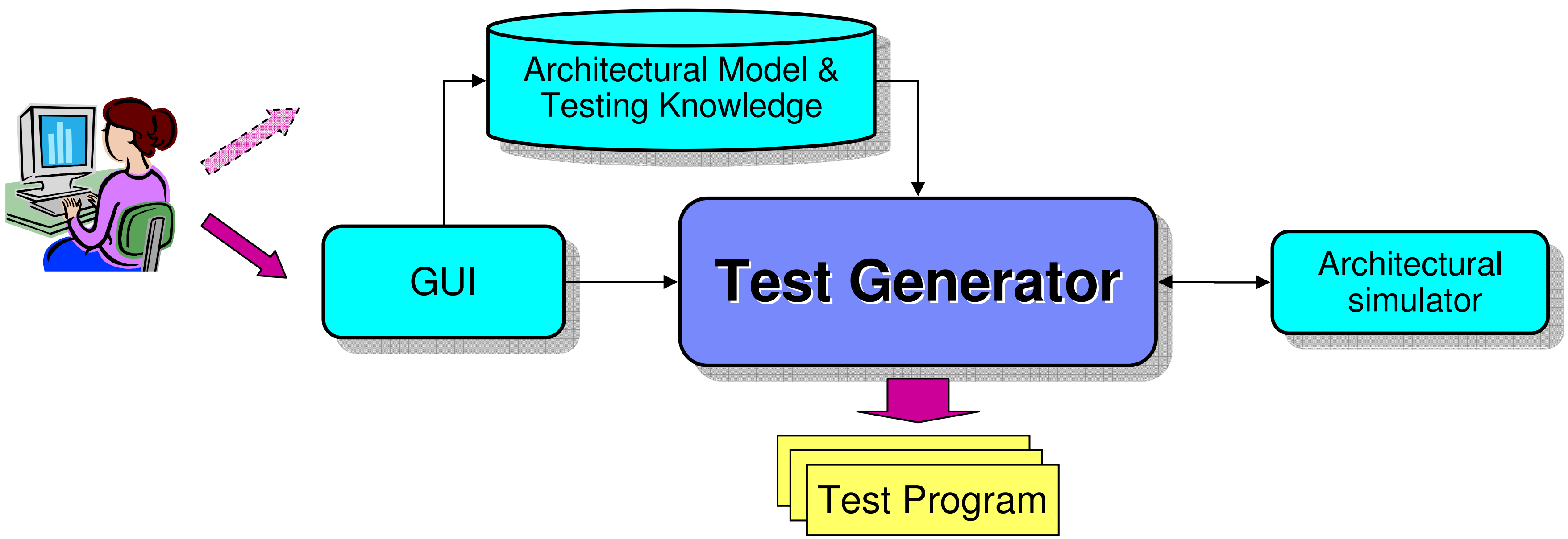
Part 1

GPro:
Test generator for microprocessors

bin@il.ibm.com

Genesys-Pro: Model-Based Test Generation

- **Generic** architecture-independent test generation engine
- External formal and declarative **architecture description**
- **Behavioral simulator** used to predict instruction execution results
- Graphical User Interface to define **generation directives**



bin@il.ibm.com

Resulting test case

Resource initialization

Instruction sequence

Predicted Results

```
INITIALIZATIONS: DATA MEMORY
D 06FFFFA4  4ADDEB16
INITIALIZATIONS: REGISTERS
R R5             050C4340
R R6             09801460
INSTRUCTIONS
I 230DB4B4 46610FE7 * STRAL R6,[R15,-R6,ASR #0x02]
                    * VA=FEADAFA4 (New) RA=06FFFFA4
I 230DB4B8 B3E77FEA * BAL 0x030D538C //0x7FE7B3
                    * VA=030D538C (New) RA=4ECBC38C
I 4ECBC38C 0E50BDE7 * LDRAL R5,[R13,+R14]!
                    * VA=40BC6FA4 (Used) RA=06FFFFA4
RESULTS: REGISTERS
R R5             09801460
R R6             09801460
RESULTS: DATA MEMORY
D 06FFFFA4  60148009
```

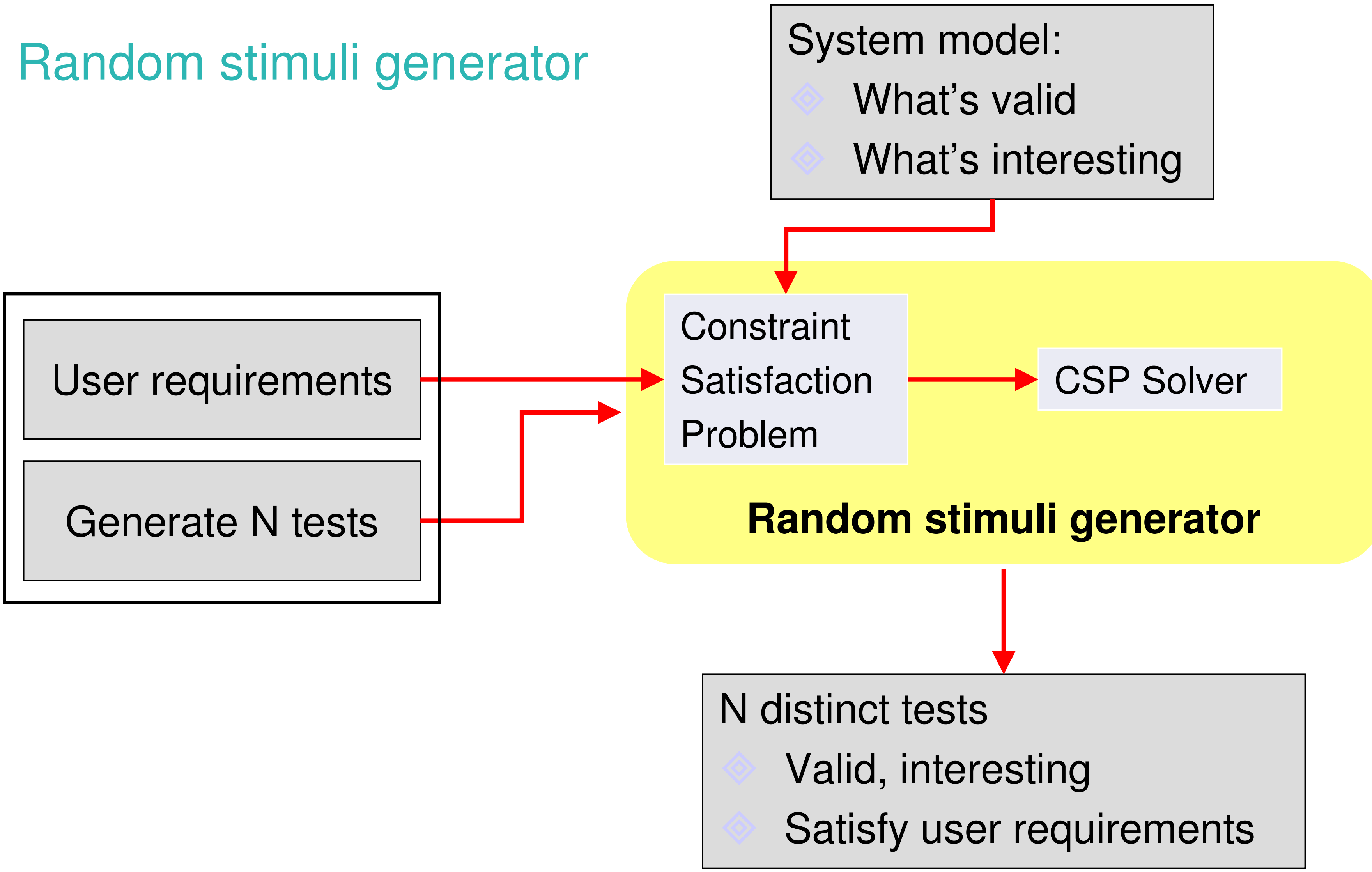

bin@il.ibm.com

Generation scheme – user view

1. Choose the **next instruction** to generate, according to:
   - Test template definition (test's specification)
2. **Generate** instruction
   - Initialize resources as required
3. Call **reference model** to simulate instruction
4. Repeat until all test template statements generated

bin@il.ibm.com

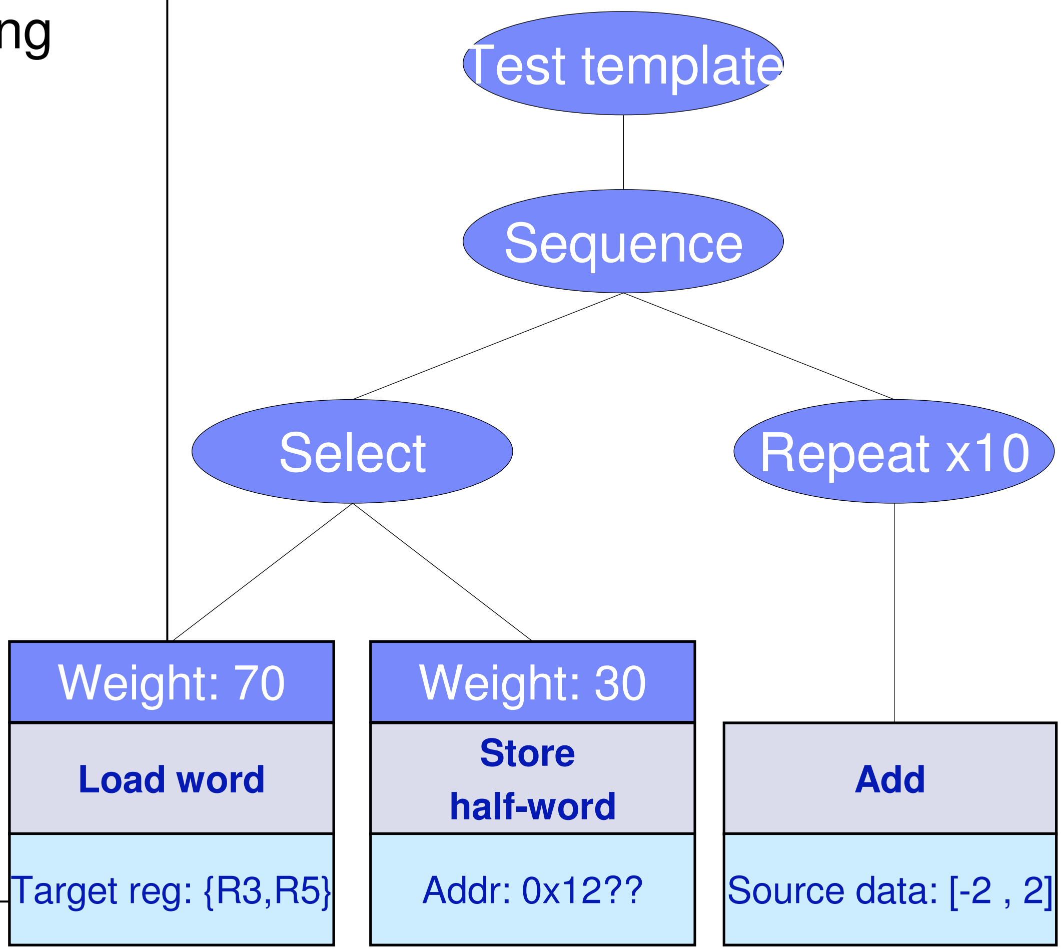
Random stimuli generator



bin@il.ibm.com

GenesysPro system input: test template basics

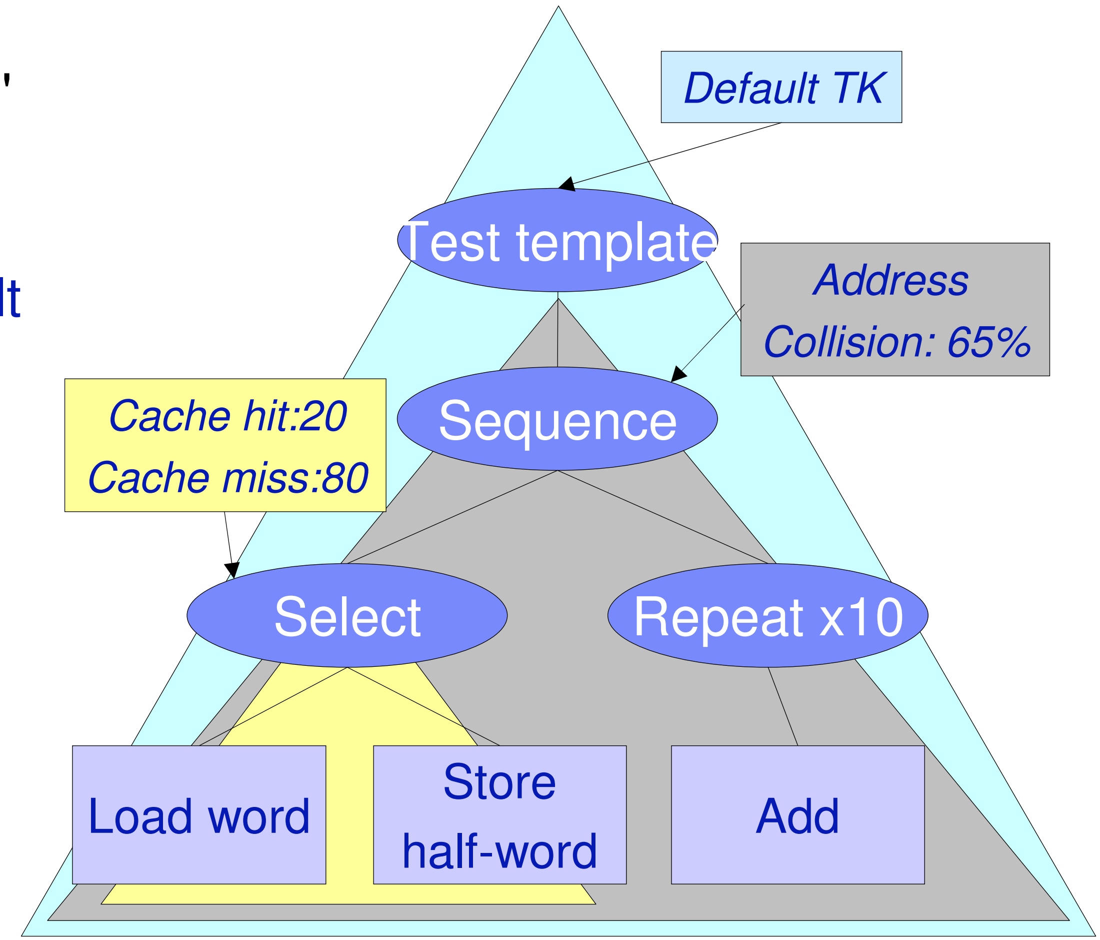
- Instruction as the basic building block
- Full control over instruction properties:
  - Data, Address, Length,…
- A hierarchy of higher level statements
  - Select: weighted random choice
  - Repeat
  - Sequence



bin@il.ibm.com

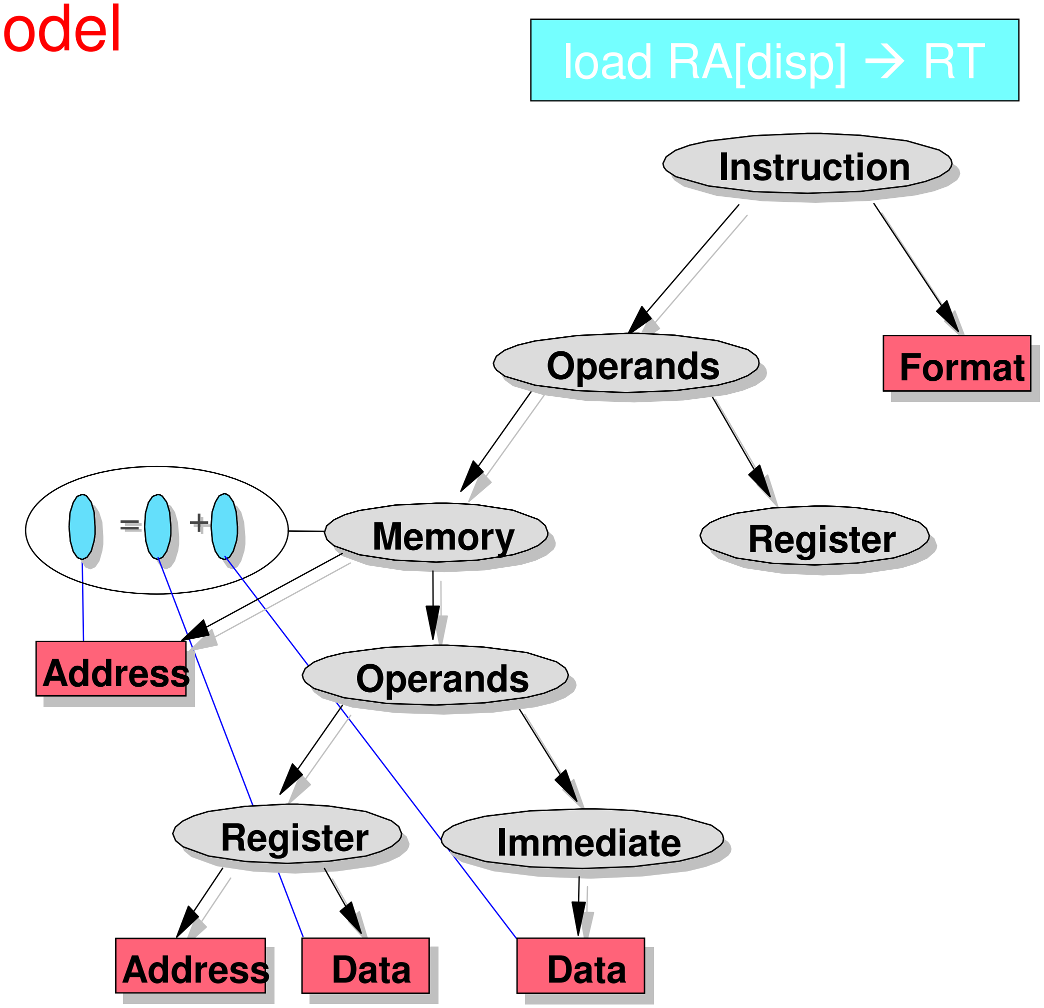
Test template: testing knowledge and directives

- **Directives** as 'volume knobs' to control TK characteristics
  - Testing knowledge also affects the test by default
- Directives present in the test template take precedence
- Scope based influence



bin@il.ibm.com

Instruction model

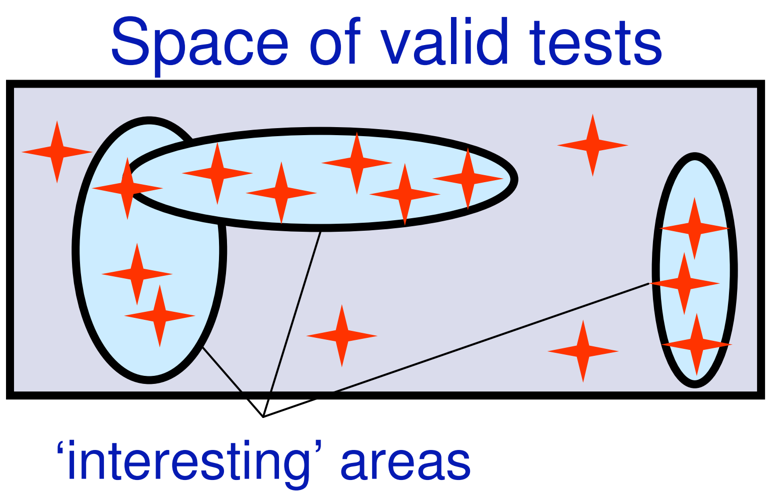


Object-oriented ontology language with a focus on constraint modeling

bin@il.ibm.com

The concept of generic testing knowledge

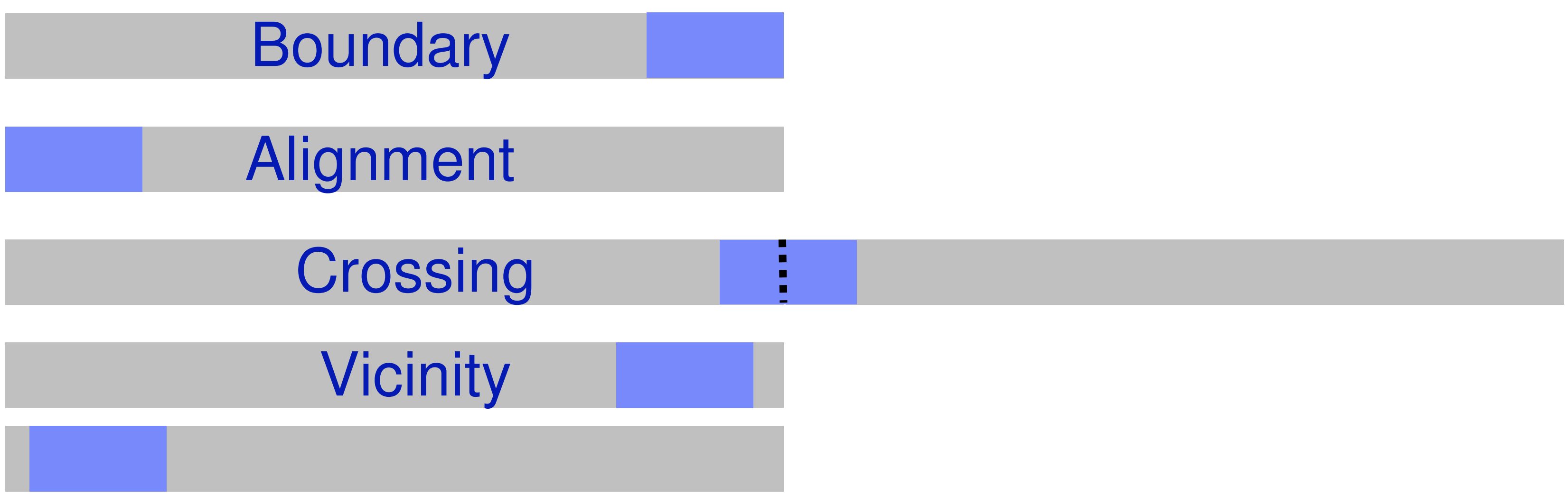
- A set of mechanisms that aim at **improving test-case quality**
- Capitalize on **recurring concepts**
- The basic mechanism: non-uniform random choice
  - Bias towards 'interesting' areas
- Affects all generated test-cases
  - But can be controlled by users
- Examples:
  - Resource collisions
  - Translation table entry reuse



bin@il.ibm.com

Testing knowledge example - placement

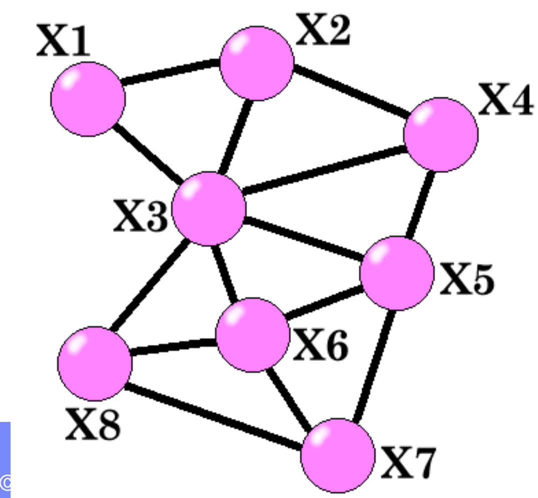
- A storage partition is a contiguous piece of memory
  - L2 cache line, page, word, half-word...
- Four types of events



bin@il.ibm.com

Why CP?

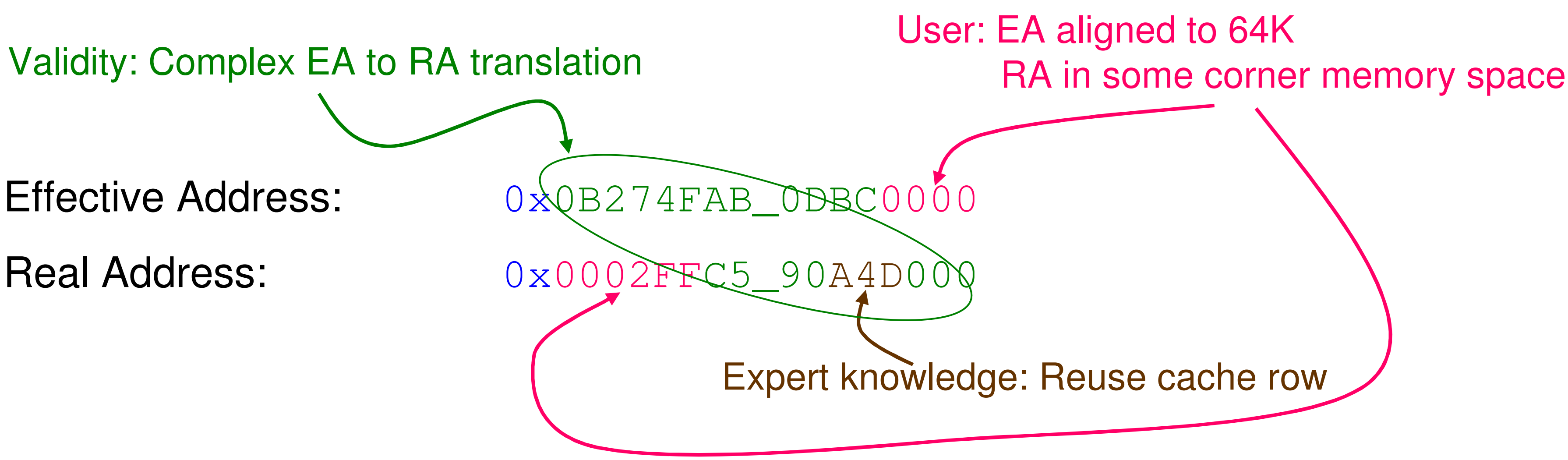
- CP enables requests coming from **different resources**
- CP gives the option to constraint **results**
- CP solvers enable approximation of **uniform coverage**
- The microprocessor **specification** is written declaratively
  - Easy translation into constraints
  - non-linear constraints
- Mandatory and **bias** (not mandatory) requests



Why CP?

Constraints originate from three sources

1. Validity of the stimuli: Constraints defined by the specification
2. Verification task: Constraints defined by the user
3. Bias towards interesting tests: Soft constraints defined by domain experts



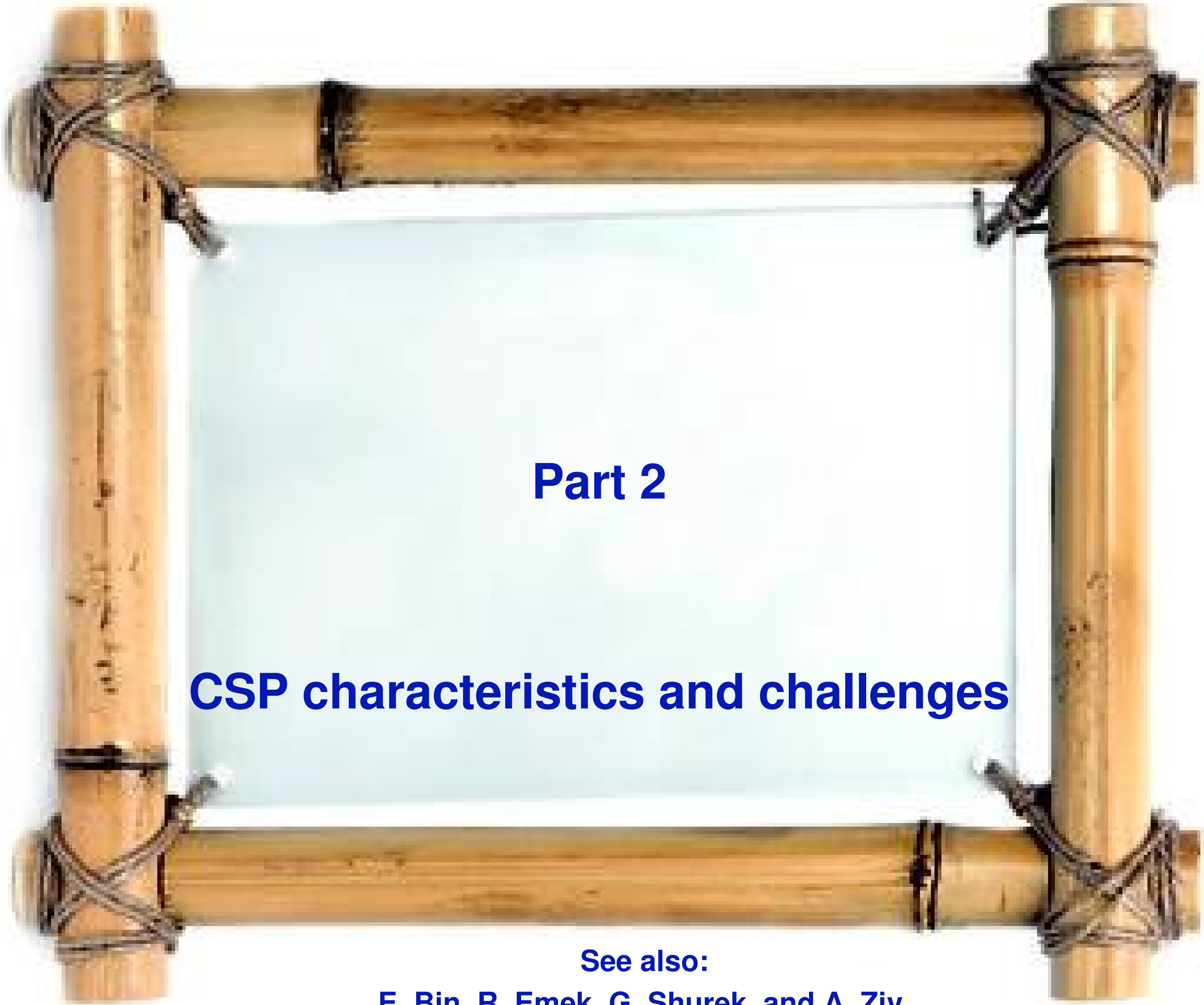
bin@il.ibm.com

Not just IBM

- Constraint satisfaction is the basis for modern stimuli generation across the industry
- 42nd DAC:
  - The largest conference of the EDA industry, 6000 participants
  - A full-day tutorial about constraint satisfaction for stimuli generation
- A typical industrial advertisement:

"   ***Constraint-Driven Test Generation***
With Specman Elite's constraint-driven test generation, you can now automatically generate tests for functional verification. By specifying constraints, you can quickly and easily target the generator to create any test in your functional test plan …"

bin@il.ibm.com

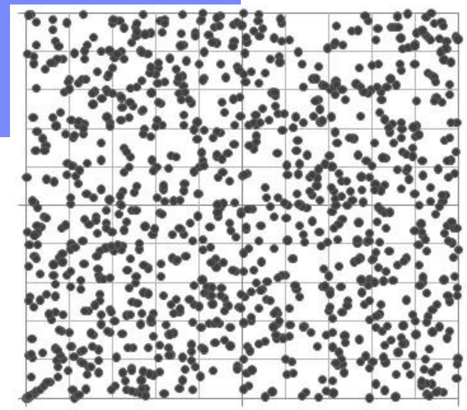


Part 2

CSP characteristics and challenges

See also:
E. Bin, R. Emek, G. Shurek, and A. Ziv,
Using constraint satisfaction formulations and solution techniques for random test program generation,
IBM Systems Journal 41, 2002

bin@il.ibm.com



Random Solution

Requirement:

- Find many random, uniformly distributed, solutions of the same CSP
  - Many different tests from the same template
  - As opposed to one, all, or 'best' solution
  - Motivation: Test different computation paths of the microprocessor

Solution:

- Uniform solution distribution is approximated by random variable and value ordering

See also: Dechter et al., AAAI 2002

bin@il.ibm.com

Huge domains

Requirement:

- The domain of many variables is 2^{128}
  - Example: address space
  - In conjunction with arithmetic, bit-wise, and other types of constraints
  - Representation and operations on sets becomes an issue

Solution:

- Inaccurate representation (over approximation)
- Using also bit-vectors representation

bin@il.ibm.com

Domain (set) representation example: bit-vectors

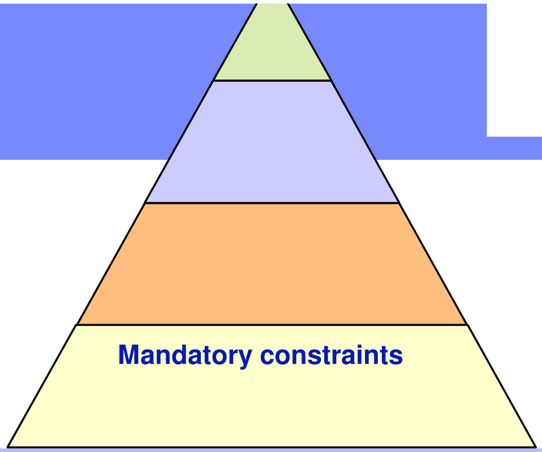
- All the addresses such that:
  - addr = base + displacement : architectural
  - addr[3:6] = 01x1 : cache line
  - addr ϵ [0x20000000 : 0x10FFFFF] : memory space

- 'Masks' (bits vector) representation:
  - 0b01x1 → 0b0101, 0b0111

- Exponential explosion
  - 01010101 + 0x0x0x0x
    {10101010, 01101010, 10011010, 01011010,
    10100110, 01100110, ..., 10010101}

bin@il.ibm.com

Hierarchy of constraints



Requirement:

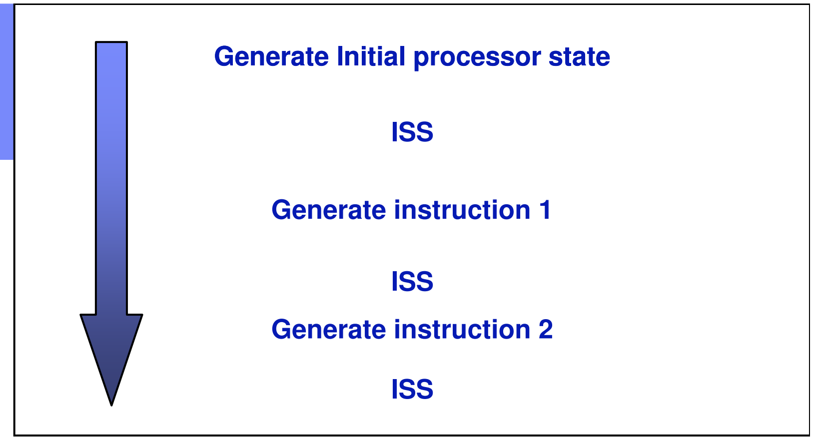
- Different priority of constraints
  - Mandatory: test case validity
  - Non-mandatory: makes the test 'interesting‘
  - Multiple levels of soft constraints – according to level of interest

Solution:

- Modeler specifies the constraint priority
- In each MAC, Mandatory constraints propagate first. Then one bias, mandatory constraints again, …

bin@il.ibm.com

Coupled CSPs



A challenge:

- Cannot generate all instructions simultaneously
  - Instructions' semantics is not modeled
  - Problem is too large
  - Constraint propagation computationally hard

A Partial Solution:

- Instructions are generated one at a time, and then executed by an ISS (Instruction Set Simulator)
- But … Instruction 3 may require a specific configuration

bin@il.ibm.com

Conditional CSP

A challenge:

- Parts of the problem’s variables and constraints should not exist in the solution

A Solution:

- A tree representation. A node may have an ‘exists’ Boolean variable.
- Constraints within the existence node work as long as the ‘exist’ variable is not false.
- External variables have a shadow. The shadow var is synchronized with the real one when the exists variable becomes true.

See also: F. Geller and M. Veksler, "Assumption-based pruning in conditional CSP", CP 2005

bin@il.ibm.com

External variables (Remote)

A challenge :

- Some CSP variables can not be represented as a set of discrete values
  - In the solution the variable is not a single element
  - The variable is shared in several CSPs
  - Example: content of memory

A Solution:

- The engine holds a variable having no domain.
- The relations communicate with the data base during propagation
- Relations mark the propagated variables as ‘modified’, so the engine knows which other propagators to call.

bin@il.ibm.com

Run time performance

Requirement:

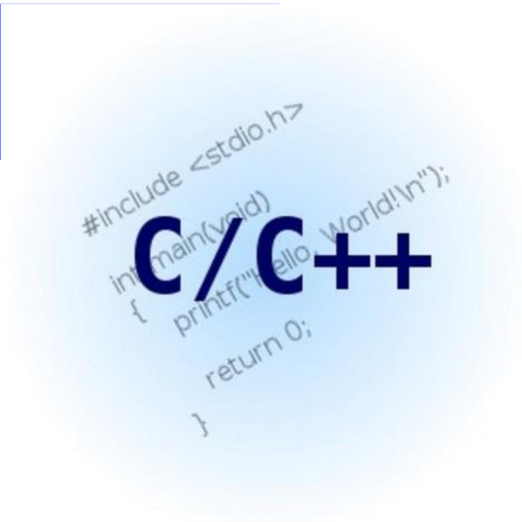
- Generation of a test should not take more time than its simulation time

A Solution:

- Instructions are generated one at a time
- Similar problems are cached and reuse

bin@il.ibm.com

Our major CSP solver



- GEC
  - **Systematic**, based on MAC-3
  - Since 1995, many person-years invested
  - Finite domain set libraries: “PD” (**primitive domains**)
    - Bool, int, bit-vector, object, string
  - Generic **expression propagator (ERP)**
    - Given a first order logic expression over variables, creates a propagator
  - Interfaces for user-defined **C++ propagators**
  - Arc-consistency on **conditional problems**
  - Support application specific CSP variables (**remote** variables)
  - Written in C++
  - designed to be generic
    - i.e., not specific for verification

See also: IAAI 2006, AI-Magazine 2007

bin@il.ibm.com

New challenges coming from the hardware

- More complex **micro-architecture**
  - Example: **SMT** (Simultaneous Multi Threaded)
  - More directed scenarios required
  - More requirement on inter instruction constraints
- More complex **architectures**
  - Example: Translation
  - Complex CSP, solving issues
-
- **Virtualization**
  - Translation CSP problem replicated
  - A scalability issue

bin@il.ibm.com

Status for 2010

- ERP: the declarative constraints language
  - **No access** to generator's internal values
  - Just **primitive operators**
  - **Insufficient expressiveness**

- Many constraints are written in C++:
  - C++ code produces a better run-time performance

↓

Maintenance **cost** and modeling new designs become an issue

bin@il.ibm.com

Why propagators C++ coding is not recommended

- Less **readable**
- Much more **lines** of code
- No **reuse**
- Hard to **maintain**
- Hard to **debug** (log file does not show the semantics)
- Does not enable **composition** of operators
- Some times written with just **partial propagation**
- Much more **time** to code it
- The CSP engine sees it as a **black box**

bin@il.ibm.com

Part 3

PRB:
The new CSP approach

bin@il.ibm.com

In short

- PRB address the shortcoming ERP of the
  - Allow greater **expressive**
  - Allow seamless **integration** with the application
  - Support **random** decisions in non-mandatory constraints
  - Built in solving **heuristics**

PRB is a generic module.
It is used also by non-verification CSPs

PRB. PRopagator Builder. Principles:

- Primitive types:
  - **New** primitive types
- Constraints
  - Constraints are written **declaratively** (not in C++)
  - Many **new operators**
  - Operators can be **composed**
  - **Macros**
- Interface:
  - PRB **communicates** with the application
  - Application can **configure** PRB
- Solving:
  - Generic management of representation **explosion** problem
  - No modeling of propagation ordering
  - **Semantics** based variable and value **ordering**

bin@il.ibm.com

Example: Direct access

C++ propagator

```
GP_STATUS GP_MATCH_LPIDR_VALUE(PD_BitStream &RS)
{
  TRY (GP_MATCH_LPIDR_VALUE)
  PD_BitStream word0, LPIDR;
  RS.GetSubField(0, 31, word0);
  static ROI_ObjId LPIDR_ID = REL_Kernel::GetMnemonicResourceId("LPIDR");
  REL_Kernel::GetRegisterContents(LPIDR_ID, LPIDR);
  Intersect(word0, LPIDR, "GP_MATCH_LPIDR_VALUE");
  Intersect(LPIDR, word0, "GP_MATCH_LPIDR_VALUE");
  RS.SetSubField(0, word0);
  if (RS.IsEmpty()) return GP_EMPTY;
  return GP_EXACT;
  CATCH
}
```

PRB propagator

```
PRB_MATCH_LPIDR_VALUE:
        subField(data,32,63) = resources.LPIDR
```


bin@il.ibm.com

```
constraint ERP:ERP_MaskAligned
( Aligned_Addr : bitstream,
  Unaligned_Addr : bitstream,
  in Alignment : integer
  %FormalFacet:: < precondition: SingletonPrecondition, range: < > >
)
["let AlignmentMask (PD_Int) : PD_BitStream {
   (2)   : 0xFFFF_FFFF_FFFF_FFFE,
   (4)   : 0xFFFF_FFFF_FFFF_FFFC,
   (8)   : 0xFFFF_FFFF_FFFF_FFF8,
   (16)  : 0xFFFF_FFFF_FFFF_FFF0,
   (32)  : 0xFFFF_FFFF_FFFF_FFE0,
   (64)  : 0xFFFF_FFFF_FFFF_FFC0,
   (128) : 0xFFFF_FFFF_FFFF_FF80,
   (256) : 0xFFFF_FFFF_FFFF_FF00
}
(
  A: PD_BitStream << 0xXXXXXXXXXXXXXXXX >> ,
  B: PD_BitStream << 0xXXXXXXXXXXXXXXXX >> ,
  C: PD_Int << {2, 4, 8, 16, 32, 64, 128, 256} >>
) :
{
  a in A;
  b in B;
  c in C;

  a = b bit_and AlignmentMask(c);
}
";];
```

ERP propagator

```
instance PRB_MaskAligned:  PRB_GeneratorMacro = <
  parameters: "Aligned_Addr, Unaligned_Addr, Alignment",
  body: "Aligned_Addr =
    maskLsbField(Unaligned_Addr, log2(Alignment)-1, ZERO)">;
```

PRB propagator

bin@il.ibm.com

Constraints types

- All the following types reduce values from variables.

- Propagator
  - A deterministic logical / arithmetical algorithm.
  - Reduce values that do not have a support.
  - Used within MAC algorithm
- Restrictor
  - A non-deterministic logical / arithmetical algorithm.
  - It draw values
  - Used within MAC algorithm

- In addition to the priority of the constraint (mandatory / bias)

bin@il.ibm.com

Operators wealth

- The more operators in the language
  - The modeling is shorter
  - More readable
  - Better propagation
  - Better run-time

bin@il.ibm.com

Better propagation for higher level operators

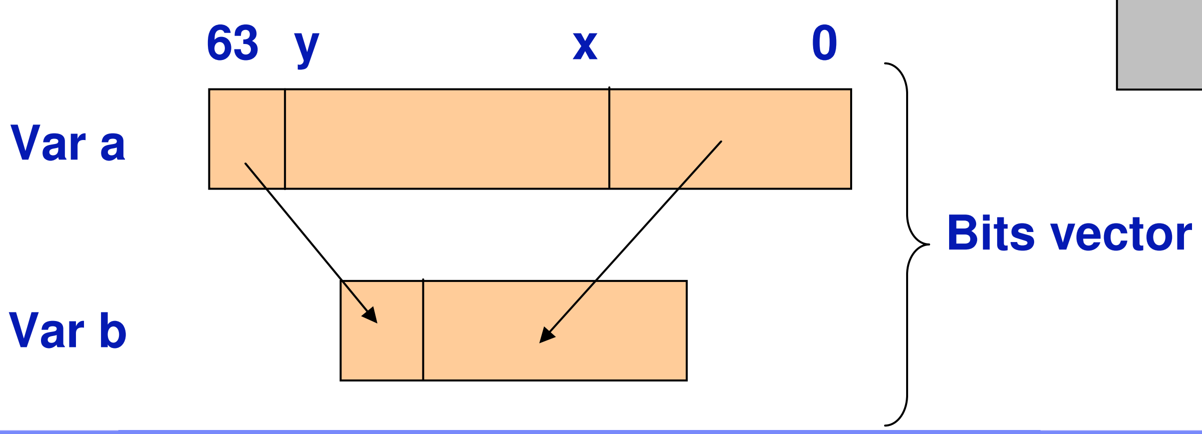
Option 1: b = concat(subField(a, y+1, 63), subField(a, 0, x-1))

Option 2: b = pullOutSubField(a, x, y)

Option 1 produces weaker propagation:

1. **A delay: When x, y are not a single element**
2. **Tightness: 'concat' collect too many elements**

a = { 00 11 11,
01 00 00}
option 1: b = {0011,
0000,
0111,
0100}
option 2: b= {0011,
0100}



bin@il.ibm.com

Old primitive set types

- Integers
- Boolean
- String
- Enums
- Bits vector

bin@il.ibm.com

New primitive types

- Bits vector now have different formats
  - Plain bits
  - Unsigned integers
  - Signed integers *
  - Decimal representation
  - Floating point representation *
- Interval
  - Each interval holds two primitive sets for ‘start’ and ‘length’

- Dates

* Not done yet

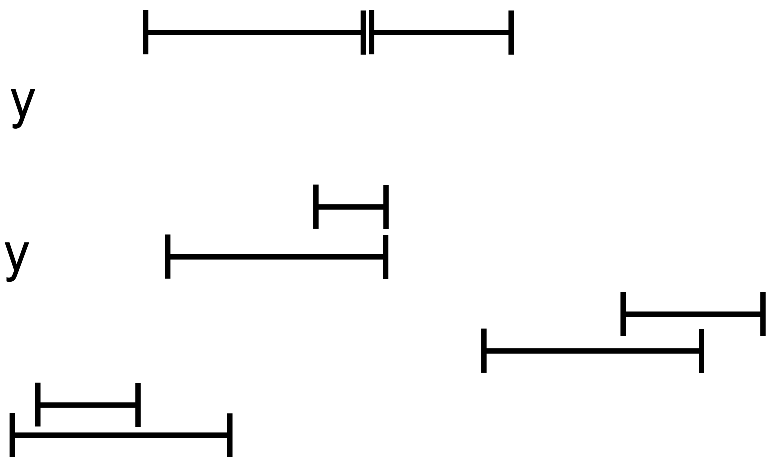
bin@il.ibm.com

Operators

- To have a feel of the operator library, we will see different operators
  - Just examples (there are more)
  - We will not understand the semantics of all of them (a quick session)
  - The syntax is not the issue

Intervals geometric operators: examples

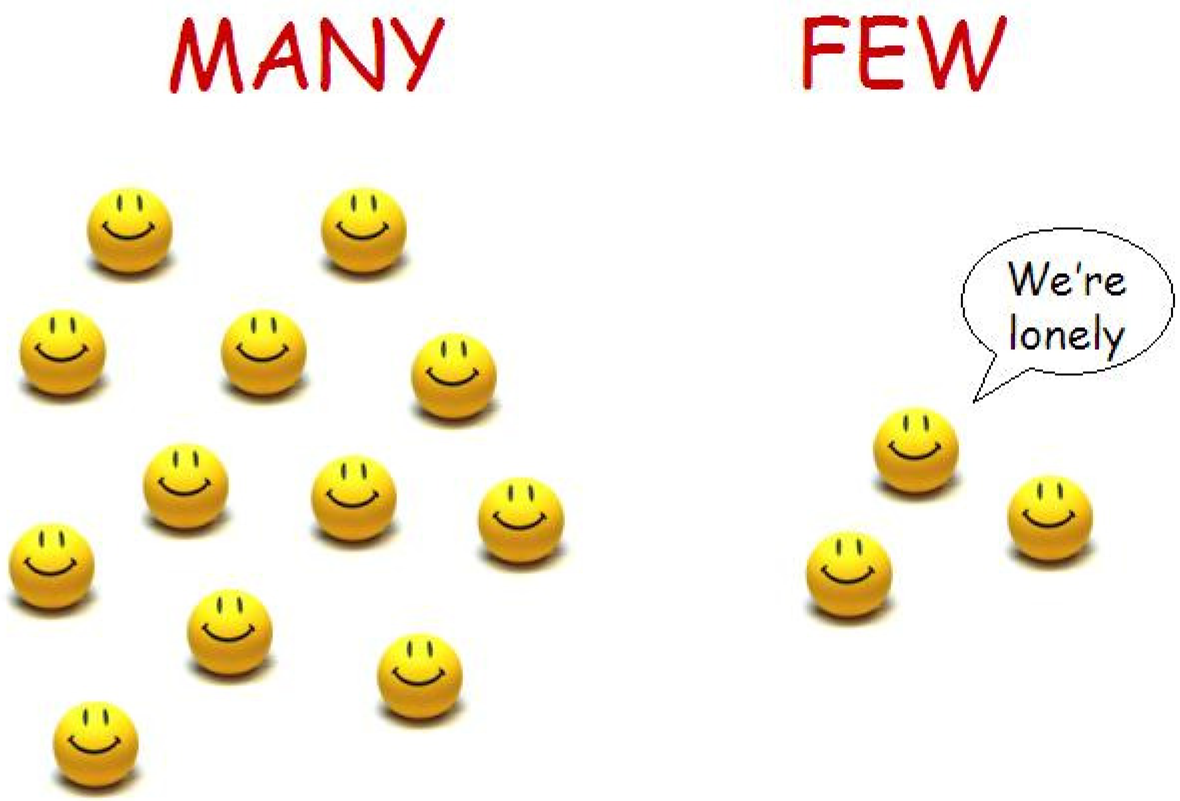
- x before y
- x conscutivesTo y
- x adjacent y
- x crossesBeyond y
- x crosses y
- x sameBoundary y
- x overlaps y
- x contains y
- x shorterThan y



bin@il.ibm.com

Global constraints: examples

- allDiff
- sumOf
- numOf
- exist
- collect
- select
- forAll
- forEach
- minOf
- maxOf



bin@il.ibm.com

Properties of global constraints

- **Similar syntax** for all global constraints
- Formats:
  - Using vectors: ***forAll(i*, 0, 7, vec[i].size > 0)**
  - Using objects: ***forAll(i*, *homes*({employes}), i.salary > 20000)**
  - Using items: ***forAll(i*, *items*({from, to}), shape.i < 100)**
- Conditions: Optional
  - ***allDiff(i*, *homes*({roads}), i.city != NY, i.name)**

italic represents a PRB reserved word

bin@il.ibm.com

Fields operators: examples

- carry
- concat
- subField
- extend
- maskField
- setField
- overflow
- pullOutSubField
- sameLsb
- numLsbBits
- Bitwise operations

bin@il.ibm.com

Square Parentheses [ ] operators

- Direct access to a field of a known register
  - resources.MSR[TR]
    is equivalent to
    subField(resources.MSR, 4, 6)
  - The application informs PRB about all the known register fields

- The indirect operator
  - vec[x+3] = y
    both x and y are CSP variables

bin@il.ibm.com

The triple operator

- x = (condExp ? thenExp : elseExp)

  This operator was found essential.

- x = (cond1 ? then1,
  cond2 ? then2,
  cond3 ? then3,
  ….
  condN ? thenN : else)

Boolean Operators

- memberOf
- table
- positive
- negative
- zero
- find

```
b = table(x, y, z,
        {
          ( << 0b0 >>, UBool, UBool) : false,
          ( << 0b1 >>, UBool, false) : false,
        //( << 0b1 >>, false, true ) : illegal
          ( << 0b1 >>, true,  true)  : true
        });
```

The tupels of the table can be generated in run time.

bin@il.ibm.com

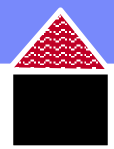
Restrictors Operators

- choose
- maxValue
- minValue
- randomBool
- randomMSBValues
- randomWeightedNumber
- randomWeightedValue
- randomNumber
- randomValue

These operators are legal just in non-deterministic constraints

bin@il.ibm.com

Homes: background



- An application's class. Inherits from PRB_Home.

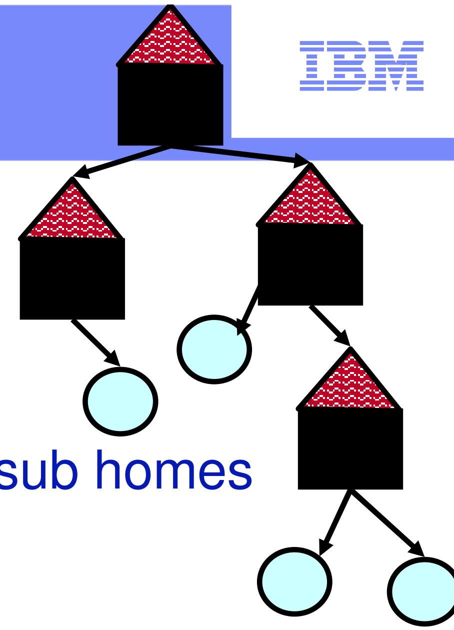
- Includes (optionally):
  - Variables (inherits from PRB_Variable)
  - Constraints
  - Sub homes

- The application can add any data members / methods

- The home serves PRB during constraint hatching:
  - FindVar()
  - GetImmediateValue()
  - GetHomesGivenType()

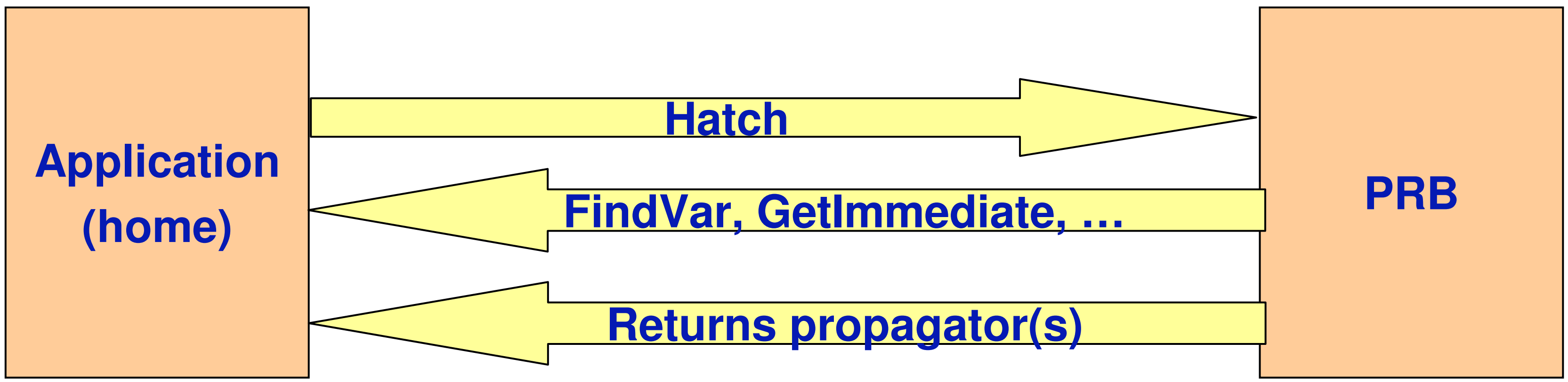
bin@il.ibm.com

Interface: PRB <-> Application
Propagators creation



- The application creates a tree of 'home's
  - Each home holds CSP variables, Constraints and sub homes

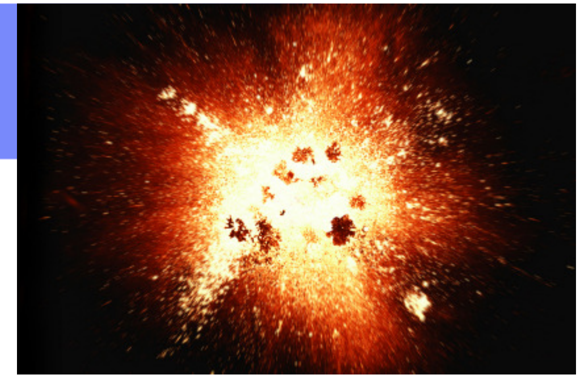
- Propagators creation
  - This interface enables sending expressions with unknown number of variables



bin@il.ibm.com

Interface: PRB <-> Application
Configuration

- Reserved words
- Max number of masks per variable
- Register fields
- Table's tuples
- Macros
- Depth of conflict detection
- … and many more



Over approximation

- PRB over approximates the variable's content
- Requirements:
  - **Reduce** the number of masks to the requested level
  - Do not insert values that were not in the variable's domain when entering the propagator
  - Insert as few values as can

- Partial solution
  - While the number of masks is too many
    - Find two similar masks (heuristics)
    - Combine the masks
    - Reduce other masks that contained in the new one

0bxxxx1
0bxxx1x
↓
0bxxxxx

bin@il.ibm.com

Conflict Detection

- **a > b**
- **a < b**

- Constraints contradiction should be handle specifically since regular MAC with large domains does not cope with it efficiently.

- Our solution: instrumentation
  - Insert an auxiliary variable v
  - convert the constraints

Conflict Detection: examples

Original:

$(a > b)$ and $(b > a)$

Instrumented:

$(v_{\underline{a,b}} > 0)$ and $(v_{a,b} < 0)$ and $(v_{a,b} > 0 \leftrightarrow a > b)$ and $(v_{\underline{a,b}} < 0 \leftrightarrow a < b)$

Original:

$(a > b)$ and $((x=1) \rightarrow (a < b))$

Instrumented:

$(v_{\underline{a,b}} > 0)$ and $((x=1) \text{ -> } (v_{\underline{a,b}} < 0))$ and $(v_{\underline{a,b}} > 0 \leftrightarrow a > b)$ and $(v_{\underline{a,b}} < 0 \leftrightarrow a < b)$

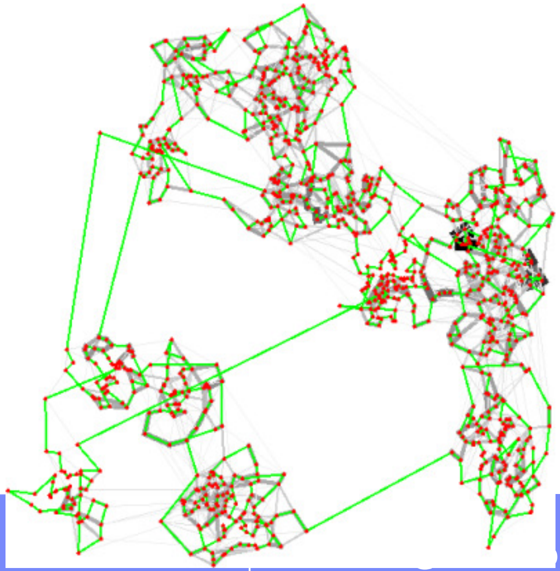
bin@il.ibm.com

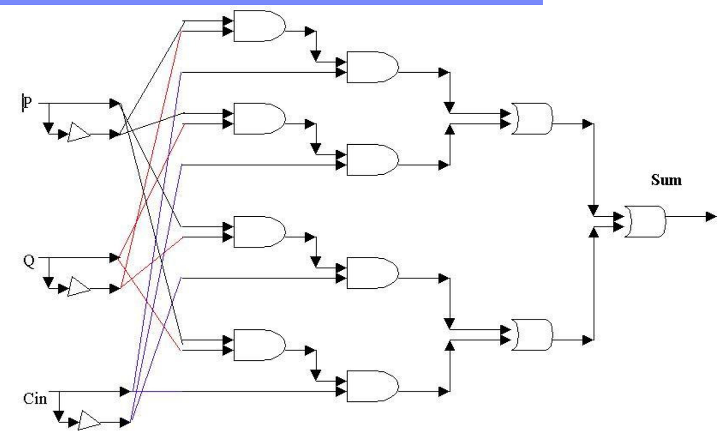
Semantics Variable and Value ordering (heuristics)

- When the semantics of the constraint is not a black box, it can be used for variable and value ordering

- Two methods:
  - Static – partial ordering is defined before solving starts
  - Dynamic – ordering is defined during solving time

- Both methods neither use the number of values in a domain nor the constraints graph.





Static Semantics Variable Ordering

1. Variable V is selected randomly as a candidate to be instantiated
2. If all the variables Vs that V depends on have a single value, return V otherwise, choose randomly a variable from Vs and go to 2.

Comments:

1. If variables' cycle is exposed, the variables in the cycle do not returned.
2. Work on fields granularity.

- Characteristics:
  - Automatic
  - Sensitive to the way the user writes the constraints
  - Works in causal CSP networks

bin@il.ibm.com

Static Semantics Variable Ordering: examples

- Equal operator at the constraint's tree root:
  a = b + c   a depends on b, c

- 'imply' operator at the constraint's tree root:
  (a>7) -> (b > c)   b, c depends on a

- Fields granularity
  subField(a, 2, 3) = …   just the two bits of a are depended

bin@il.ibm.com

Dynamic Semantics Variable and Value Ordering Motivation

3	12	56	83	14	94	22	22	2	92

The domain of v[i] is [0, 100]

numOf(i, 0, 9, v[i] = 0) > 0

numOf(i, 0, 9, v[i] = 1) > 0

numOf(i, 0, 9, v[i] = 2) > 0

numOf(i, 0, 9, v[i] = 3) > 0

numOf(i, 0, 9, v[i] = 4) > 0

numOf(i, 0, 9, v[i] = 5) > 0

bin@il.ibm.com

Dynamic Semantics Variable and Value Ordering

- During regular propagation, when a propagator has multiple ways to be satisfied, it registers itself
- During variable ordering:
  - Choose one of the registered propagators
    - Last one
    - Random one
  - Invoke the propagator in ‘ordering’ mode
  - When the propagator has multiple ways, it chooses one of them and satisfies it.
    - Last way
    - Random one
  - A variable does not change the real domain, but works on a copy
  - The variables that were copied (and their new domain) are the suggestion.

bin@il.ibm.com

Wrap up

- Simulation is still the main platform for hardware verification
- Biased random test generation is widely used in the industry
- CSP is the major technique used for generating tests
- Architectures and micro-architectures enforce new CSP techniques
  - Modeling languages
  - Domain representation
  - Variable and value ordering
  - CSP debug methods

bin@il.ibm.com

Thank you

bin@il.ibm.com