System Design
Taking Informed Decisions

KTH Information and Communication Technology

Christian Schulte
School of Information and Communication Technology
KTH – Royal Institute of Technology
Sweden

Designing a Constraint Programming System

- Model
  - entities, properties, description, reasoning, terminology, …
  - expressiveness, simplicity, purpose, …
- Architecture
  - components, interfaces, interaction, …
  - simplicity, expressiveness, invariants, …
- Implementation
  - data structures, algorithms, services, …
  - efficiency (space time), simplicity, robustness, …

Designing a Constraint Programming System

- Model
  - entities, properties, de… terminology, …
  - expressiveness, simpli…
- Architecture
  - components, interfaces, …eraction, …
  - simplicity, expressiveness, invariants, …
- Implementation
  - data structures, algorithms, services, …
  - efficiency (space time), simplicity, …

all require taking **design decisions**

Design Decisions

- Goal: what does one want to achieve?
  - generally, multiple criteria
- Decision space
  - how many alternatives
  - their properties (cost, efficiency, simplicity …)
- Dependencies among decisions
  - independent? causal? interrelated?

Here: Design Decisions for Finite Domain CP Systems

- Scope of this talk
  - what are important decisions?
  - how informed are common decisions?
  - what are good decisions?
  - what are bad or uninformed decisions?
- Restriction to rather basic decisions
  - finite domains and tree search
  - level: architecture and implementation
  - many are folklore and the thing to do (?)

Approach

- Design decisions discussed
  - search: state restoration
  - propagators: which one to run next
  - propagators: combining filter algorithms
  - variables: events and dependencies
  - variables: domain representation
- Decisions for Gecode
  - how did we reason (if we did at all…)

Gecode

`www.gecode.org`

- Current research platform
- Used for example decisions in this talk
- More information on Gecode itself
  - later in system presentation

Notation and Background

- Propagator
  - implementation of a constraint
  - example: alldifferent, linear, extensional, …
- Time and space figures
  - from some 20 benchmarks
  - paper in preparation (with Peter Stuckey)
  - results available upon request
- Anything else: interrupt me…

Search

State restoration

State Restoration

- Essential: backtrack to *previous* state
  - variable domains
  - propagators and their variable dependencies
  - propagator state
- Approaches
  - trailing — record and undo changes
  - copying — put complete state aside
  - recomputation — recompute state on need

Trailing

- Trailing stores undo and redo information
  - interleaved with constraint propagation
  - uses trail data structure
  - update: put ⟨location,content⟩
  - undo: write location ← content
  - every choice point: put mark or record top of trail
- Requires
  - all updates trail-aware
  - for example: domain change, change of dependency information, …

Time Stamping

- Problem: multiple change of same location
  - for example: multiple narrowing of domain
  - only original value needs restauration
  - intermediate values not needed
- Solution: local time stamp on modified entity
  - new choice point — increase global time stamp
  - upon modification — trail, if local stamp earlier; update local stamp

[Aggoun & Beldiceanu 90] [Aggoun & Beldiceanu 91]

Semantic Trailing

- Trail abstract operations rather than low-level updates
  - typically: establish equivalent but not same state
  - examples: CLP(R) [Jaffar ea, 1992], CLP(R-lin) [Van Hentenryck & Ramachandran, 1995], MAC [Régin, 2005]

Copying

- Take a complete copy of the system state
  - simple
  - confined to copy routine
  - rest of system orthogonal
- Infeasible due to excessive memory requirements [Schulte 1999]

Recomputation

- Remember path in search tree to recompute nodes
  - requires at least one copy to start from
- Path representation
  - number of alternative [Schulte, 1997]
  - constraints from labeling (batch recomputation [Choi ea, 2001]) (decomposition-based search [Michel & Van Hentenryck, 2004])

Impact of Path Representation

- Naïve: number of alternative
  - propagate, replay alternative, propagate, replay alternative, …
  - O(*n*) fixpoints (full propagations)
- Batch: constraints from labeling
  - add constraint, add constraint, …, propagate
  - O(1) fixpoints
  - more: cost of propagation less than number of constraints added

Properties of Recomputation with Copying

- Simplicity
  - copying orthogonal to propagation
- Expressiveness
  - parallelism, moves in search tree
- Efficiency
  - little space due to recomputation
  - amazingly (truly) efficient due to batch recomputation
  - space versus time configurable

Recomputation Optimizations

- Adaptive recomputation
  - create additional copies in between upon recomputation
- Branch-and-bound
  - naïve: recompute from copy, add constraint from last solution to yield better solution (often repeated)
  - better: add constraint to copy and possibly fail entire subtree

Decision in Gecode

- Batch recomputation and copying
  - not that simple (tricky invariants)
  - but confined to search!
  - very efficient (time and space)
  - rather complicated search engines (but we have good abstractions [Tack, 2004])
- Decision
  - ultimately intrusive: affects everything
  - informed: to some extent (batch to a lesser)

Informed Decision?

- Trailing works
  - it clearly does
  - it clearly does better for Prolog
- Recomputation with copying works better
  - simpler, orthogonal, more expressive
  - key issue for us: parallelism for free!

General Impact of State Restoration Decision

- Influences the entire system
  - data structures and operations
- Trailing
  - small data structures (eg: list)
  - operations with local effect
- Copying
  - compact data structures (eg: array)
  - operations can do whatever they want

Propagators

Which one to run next

Pending Propagators

- When variable *x* modified during propagation
  - all propagators depending on *x* must be run eventually
- Possibilities
  - immediately: stack
  - as late as possible: queue
  - decide on cost: priorities (cost)

Queue ⇆ Stack

- Stack shows pathological behavior in some cases
  - can increase runtime by 3 orders
  - in average: almost twice the runtime
- Pathological behavior
  - cheap, expensive global, cheap, expensive global, …
  - can that fixed by cost: no (jumping ahead)

Propagator Costs

- Define cost metric
  - unary, binary, ternary, linear, quadratic, cubic, crazy
  - fine metric: low and high variants
  - coarse metric: collapse some cost values
- Organize according to cost
  - one queue for each cost category
  - pick always from cheapest queue first

Using Costs

- Impact of cost metric on runtime
  - fine -7.4%
  - medium -6.3%
  - coarse -5.6%
- Variations
  - fine + stack +23.9%
  - inverted +107.5%

Using Costs

- Number of propagators executed increases
  - from 3.8% to 7.0%
- Reason: iterated fixpoints
  - cheap fixpoint
  - single more expensive propagator
  - cheap fixpoint
  - …

Decision in Gecode

- Medium metric, queue, dynamic cost
  - protection from pathological behavior
  - efficiency (to a lesser extent)
  - quite complicated: minimize number of cost computations (due to dynamic cost)
  - enabler of more optimizations (next)
- Informed decision
  - for costs: yes
  - for dynamic costs: not really
  - for priorities: not at all, why not impact?

Propagators

How to combine them

Combining Filter Algorithms

- Consider alldifferent(x)
  - naïve variable becomes assigned
remove value from other variables
cheap
  - domain find and prune Hall sets [Régin, 1994]
expensive
- Common approach
  - first naïve, then domain
  - applicable to many global constraints
  - but how?

Possible Decisions

- Nothing
  - only do expensive but strong
- Immediate
  - do cheap immediately followed by expensive
- Multiple propagators
  - create propagators for cheap and expensive
  - with according costs
  - other cheap propagators before expensive

Better: Staging

- Single propagator [Schulte & Stuckey, 2004]
  - idle and must be run: set stage one
  - stage one: do cheap
set stage two
  - stage two: do expensive
set idle
- Optimizations
  - stage one finds stage two not needed: idle
  - more stages (possibly)

Comparison

- Relative to nothing: time memory
  - immediate -1.6% 0.0%
  - multiple -4.8% +3.0%
  - staging -6.5% 0.0%
- Examples with costly global constraints
  - immediate just around -2%
  - staging often -16% up to -40% time

Decision in Gecode

- Staging
  - simple (but for every propagator)
  - efficient
  - no memory cost
  - requires sufficiently fine cost metric
  - requires ability to change propagator cost
- Informed decision
  - principle: yes
  - implementation choice (based on modification events): no

Constraint Variables

Events and dependencies

Dependency Management

- Propagator knows variables
  - perform propagation on them
- Variables know propagators
  - propagators depending on variable
  - using events to avoid useless execution
  - time and space critical

Events

- Which domain changes affect propagator
- Events when computing new domain
  - fix(*x*) variable becomes assigned
  - bnd(*x*) bound changes
  - dom(*x*) domain changes
  - events clearly overlap
  - some systems distinguish lbc and ubc

Events: Really?

- Events combine two aspects
  - how a variable domain changes
  - when a propagator needs to be run
- Better: separate concerns
  - modification event (ME): how is a domain modified
  - propagation condition (PC): for which modification events propagator must run

Modification Events Propagation Conditions

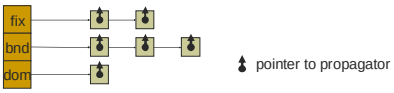
- Modification events can be rich
  - fix, bnd, dom, ubc, lbc, …
  - but also: ubc and new upper bound smaller than stated (hole in domain), …
- Propagation conditions can differ
  - just fix and dom
  - fix, bnd, dom
  - later: the more conditions the more memory
  - good for set variables

Relation ME ⇆ PC

- Give relation between ME and PC
  - relation holds: propagator must be run
  - **ME: fix, bnd, dom**
  - **PC: fix, bnd, dom**

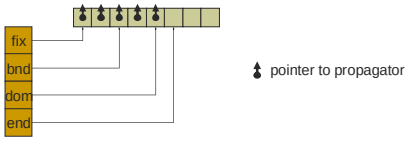
(fix,fix) **(fix,bnd)** **(fix,dom)**
(bnd,bnd) **(bnd,dom)**
(dom,dom)

Implementing Dependencies Suspension Lists



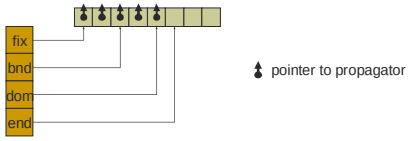
- When modification event occurs
  - process all elements from list of related propagation conditions
  - traversal reasonably efficient
  - takes two memory cells per dependency

Gecode: Dependency Arrays



- When modification event occurs
  - process elements in array areas for related propagation conditions
  - traversal very efficient
  - takes one memory cell per dependency

Dependency Arrays: Adding



- When adding new dependency
  - make one free entry (possibly resize)
  - by moving pointers into array and single element for propagation condition
  - requires time in number of propagation conditions

Dependency Arrays ⇆ Trailing

- Semantic trailing
  - when adding or removing dependency just trail that fact
  - undo operation is easy
- Low-level trailing
  - when adding or removing: several pointers into array and array elements change
- Dependent design decision

How Many Propagation Conditions?

Change	time	space
fix, dmc	-7.8%	+3.9%
plus bnd	-7.8%	+9.9%
plus ubc, lbc	-6.3%	+15.5%

- Propagation steps decrease drastically
  - but: do nothing is rather cheap…
  - but: maybe only in some situations and for some systems…

Discussion

- Gecode decision: fix, bnd, dmc
  - efficiency
  - applicability
  - memory usage
  - protection from pathological behavior
  - separation of concerns
- Informed decision
  - yes for which conditions: choices explored
  - not really for dependency arrays
  - clearly for events versus conditions

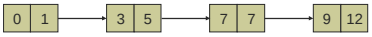
Finite Domain Variables

Domain representations

Variable Domains

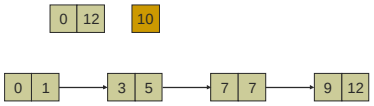
- Example: finite domain variables
  - similar situation for other domains
- What are good implementations
  - which data structures
  - which operations

Data Structures for Variables



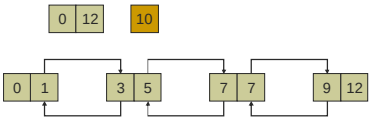
- Sorted simple-linked list of intervals
  - simple
  - commonly used
  - good for iteration (constant time)
  - bad for test and update (linear time)

Data Structures for Variables



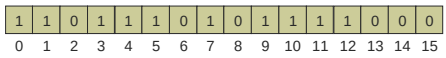
- Sorted simple-linked list of intervals
  - with bound information
  - with cardinality information
  - good for accessing bounds information

Data Structures for Variables



- Sorted double-linked list of intervals
  - provides quick access to both ends
  - decreases runtime by 50%
  - commonly used

Data Structures for Variables



- Bitvector
  - quick test and set (constant time)
  - not easy to iterate (non-constant time)
  - difficult to scale to large domains

Design Decision in Gecode

- Compromise between time and space
  - doubly linked list
  - single pointer for both links
  - uniform presentation
- Optimize common cases
  - keep bounds information
  - keep size of holes in domain (unchanged when only bounds information changes)

Single-Pointer Doubly-Linked

- Store in element $p \oplus n$ (bitwise xor)
  - p is previous, n is next
- For forward iteration
  - previous p known: next is $p \oplus (p \oplus n) = n$
- Restricted functionality
  - allows iteration but not random access
  - no time overhead
- Saves memory (32 bit integers)
  - 25% for 32bit, 33% for 64bit

Discussion

- Criteria
  - efficient (bidirectional iteration)
  - memory usage
  - simple
  - uniform (no hybrid)
- Informed
  - not really (does okay in comparison)
  - recent attempt by Minion: choose representation at compile time

Set Variables

- Way more intricate design space
- Different expressiveness
  - lub and glb [Puget, 1992] [Gervet, 1997]
  - full domain [Hawkins & Lagoon & Stuckey, 2005]
  - length-lex domains [Gervet & Van Hentenryck, 2006]
- How to implement them
  - implementations known, but variations?

Summary

Summary Overview

- Design decisions in systems
- Design decisions in papers
- Area maturity
- Fundamental decisions
- System research

Design Decisions in Systems

- Disciplined and informed reasoning mandatory
  - well-understood systems
  - flexible systems: revising decisions
  - efficient systems: design-space explored
  - interesting systems: you can say something about the system (not only that it “works” or that it is “fast”)

Design Decisions in Papers

- The easiest way to write a good paper
  - discuss a selection of decisions…
  - present your informed decisions…
  - choose abstraction level for decisions not for underlying system…
  - … paper accepted
- Good reasoning about decisions
  - others can judge merits for their systems
  - makes impact likely

Area Maturity

- Only basic decisions discussed here
  - but many decisions not that well-informed
- General
  - lack of design alternatives known and published
  - poor culture
  - poor visibility

Area Maturity

- Only basic decisions discussed here
  - but many decisi… d
- General
  - lack of design… published
  - poor culture
  - poor visibility

Fundamental Decisions

- Some decisions are assumptions
  - compute fixpoints before search
  - only communicate via variables
  - choice of propagator rather than choice of constraint
  - …
  - typically on level of model
  - very interesting, very challenging

System Research

- Not only about implementations
- Even more important: models and architectures
- Common models and architectures lead to a coherent and thriving community!
- Or, briefly: system research is just like any other type of research
  - that is: should be!