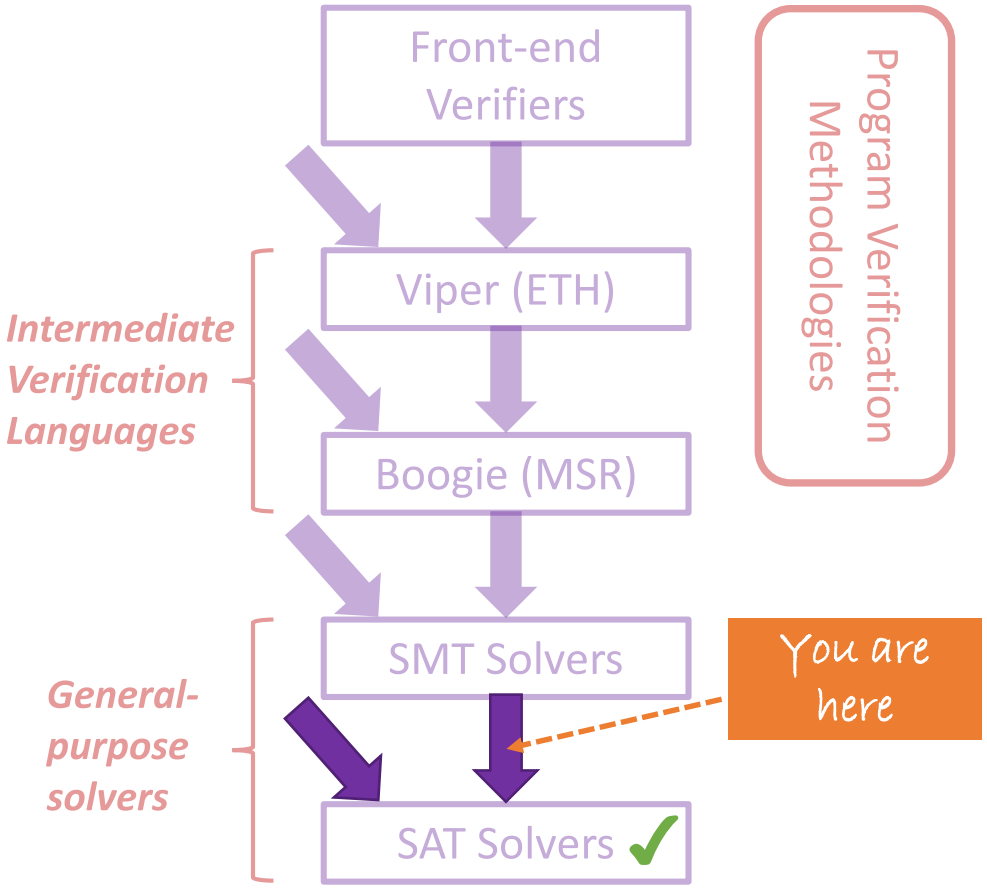
2. Encoding Problems to SAT

Program Verification

ETH Zurich, Spring Semester 2017

Alexander J. Summers

Next up: Encoding Problems to SAT



Beyond Propositional Satisfiability?

- Recall the SAT problem:

  Given a propositional formula with free variables, can we find a way to assign the free variables to make the formula true?

- Many applications naturally need features *beyond propositional logic*
  - *first-order terms*: variables, constants, function symbols $\mathrm{f(a)=b} \Rightarrow \mathrm{g(b)=a}$
  - *theories*: e.g. integer arithmetic, sets, floating points $\mathrm{n<5} \Rightarrow \mathrm{m \cdot n \leq 42}$
  - *quantifiers*: first-order "forall" ($\forall$) and "exists" ($\exists$) $\forall \mathrm{x \in S.}\ \exists \mathrm{y \in S.}\ \mathrm{y=f(x)}$

- In most cases, such features are best handled natively (SMT), but:
  - in some cases, these features *can* be encoded into propositional logic
  - we can then *directly use a propositional SAT solver*

Sorted First-Order Logic - Syntax

- Fix a set of *sorts* (types), T_1, T_2, … (typically includes Bool)
- For each sort, fix an alphabet of *term variables* $\mathrm{x},\mathrm{y},\mathrm{z},\mathrm{x}_1,\ldots$
- Fix a set of *function symbols* $\mathrm{f},\mathrm{g},\mathrm{h},\mathrm{f}_1,\ldots$ each with a *function signature*
  - a *function signature* defines an *arity* (≥ 0), sort for each argument, return sort
  - nullary functions are also referred to as *constant symbols*
- Then we can define *first-order terms* $\mathrm{t},\mathrm{s} ::= \mathrm{x} \mid \mathrm{f}(\mathrm{t}_1,\mathrm{t}_2,\ldots)$
  - we assume all terms to be type-correct (*well-sorted*) and to respect arities
- A *signature* is a set of sorts and function symbols (over those sorts)
- For a given signature, *first-order assertions* A are defined by $\mathrm{A},\mathrm{B} ::= \mathrm{t} \mid \neg\mathrm{A} \mid \mathrm{A}\wedge\mathrm{B} \mid \mathrm{A}\vee\mathrm{B} \mid \mathrm{A}\Rightarrow\mathrm{B} \mid \mathrm{A}\Leftrightarrow\mathrm{B} \mid \forall\mathrm{x}{:}\mathrm{T}.\mathrm{A} \mid \exists\mathrm{x}{:}\mathrm{T}.\mathrm{A}$
  - Here, t must be a term of (interpreted) sort Bool (assertions also Bool terms)
  - Variables / constant symbols of sort Bool subsume propositional variables

Sorted First-Order Logic - Semantics

- A *first-order model* M maps
  - sorts to non-empty sets of values: the *interpretation of the sort in* M
  - term variables to elements of the interpretations of their sorts
  - function symbols to (total) *mathematical functions* with appropriate arity, domain/range according to interpretations of function arguments/return sort
- The *value of a term* t *in a model* M, written $\ulcorner \mathrm{t} \lrcorner_{\mathrm{M}}$, is defined by:
  $\ulcorner \mathrm{x} \lrcorner_{\mathrm{M}} = \mathrm{M}(\mathrm{x})$ and $\ulcorner \mathrm{f}(\mathrm{t}_1,\mathrm{t}_2,\ldots) \lrcorner_{\mathrm{M}} = \mathrm{M}(\mathrm{f})(\ulcorner \mathrm{t}_1 \lrcorner_{\mathrm{M}}, \ulcorner \mathrm{t}_2 \lrcorner_{\mathrm{M}}, \ldots)$
- A formula A is *satisfied by a model* M, written $\mathrm{M} \models \mathrm{A}$, defined by:

$\mathrm{M} \models \mathrm{t}$ iff $\ulcorner \mathrm{t} \lrcorner_{\mathrm{M}} = \mathrm{true}$. $\mathrm{M} \models \neg \mathrm{A}$ iff $\mathrm{M} \not\models \mathrm{A}$ …(usual propositional cases)

$\mathrm{M} \models \forall \mathrm{x}{:}\mathrm{T}.\mathrm{A}$ iff for all values $\mathrm{v} \in \mathrm{M}(\mathrm{T})$, $\mathrm{M}[\mathrm{x} \mapsto \mathrm{v}] \models \mathrm{A}$

$\mathrm{M} \models \exists \mathrm{x}{:}\mathrm{T}.\mathrm{A}$ iff for some value $\mathrm{v} \in \mathrm{M}(\mathrm{T})$, $\mathrm{M}[\mathrm{x} \mapsto \mathrm{v}] \models \mathrm{A}$

- here $\mathrm{M}[\mathrm{x} \mapsto \mathrm{v}]$ denotes a new model mapping x to v and otherwise unchanged

Interpreted vs. Uninterpreted Symbols

- Sorts and function symbols are either *interpreted* or *uninterpreted*
  - e.g., Bool is *interpreted*: should always be mapped to mathematical booleans
  - boolean equality $=$ ($\mathrm{Bool} \times \mathrm{Bool} \rightarrow \mathrm{Bool}$) is an *interpreted function symbol*
  - we will usually write interpreted binary functions (operators) infix, i.e. $\mathrm{b} = \mathrm{b}$
- Interpreted symbols impose *constraints* on the set of possible models
  - the interpretations of these symbols are fixed, *independently of the model*
  - similarly, the semantics of *bound variables* is independent of the model
  - models specify the interpretations only of the *free, uninterpreted* symbols
- Notions of *satisfiability*, *validity*, *entailment* all generalise directly
  - for example, the formula $\neg(\mathrm{b} = \mathrm{b})$ is *unsatisfiable*
  - however, the formula $\neg \mathrm{f}(\mathrm{b},\mathrm{b})$ is *satisfiable* if f is an uninterpreted function
  - the formula $\mathrm{a}{=}\mathrm{b} \Rightarrow \mathrm{f}(\mathrm{a}){=}\mathrm{f}(\mathrm{b})$ is *valid*; f must represent *some* function

Extension: Propositional Logic plus Equality

- Consider first a signature consisting of *only*:
  - a single uninterpreted sort T, plus the interpreted sort Bool
  - interpreted Bool constant symbols $\top$ and $\bot$ with the usual meaning
  - interpreted equality on T, written $=$ with the usual meaning ($\mathrm{T} \times \mathrm{T} \rightarrow \mathrm{Bool}$)
- Consider only *quantifier-free* formulas over this signature
  - e.g. we can write formulas such as $\mathrm{x}{=}\mathrm{y} \wedge \mathrm{y}{=}\mathrm{z} \Rightarrow \neg(\mathrm{x}{=}\mathrm{z})$ but not $\forall \mathrm{x}{:}\mathrm{T}.\mathrm{x}{=}\mathrm{x}$
- We can encode SAT problems in this syntax into propositional SAT
- Key observation: formulas can only concern finitely-many T elements
- With only equality, the only *relevant* information in a model is:
  - are a particular pair of these elements equal or not equal to each other?

Ackermannization (Eliminating Equality) I

- The trick employed is a special case of so-called *Ackermannization*
- For each application of equality $\mathrm{x}{=}\mathrm{y}$ in the original formula:
  - introduce a fresh variable $\mathrm{eq}_{\mathrm{x,y}}$ of sort Bool
  - replace all occurrences of $\mathrm{x}{=}\mathrm{y}$ in the original formula, with $\mathrm{eq}_{\mathrm{x,y}}$
- For example, given the formula $\mathrm{x}{=}\mathrm{y} \wedge \mathrm{y}{=}\mathrm{z} \Rightarrow \neg(\mathrm{x}{=}\mathrm{z})$ we obtain $\mathrm{eq}_{\mathrm{x,y}} \wedge \mathrm{eq}_{\mathrm{y,z}} \Rightarrow \neg\mathrm{eq}_{\mathrm{x,z}}$ where $\mathrm{eq}_{\mathrm{x,y}}$, $\mathrm{eq}_{\mathrm{y,z}}$, $\mathrm{eq}_{\mathrm{x,z}}$ are Bool variables
- Is the result equisatisfiable with the original formula?
- *No*: we are missing the properties of the *interpreted function* $=$
- We can fix this by *conjoining* reflexivity, symmetry, transitivity facts
  - this might require extra Bool variables (e.g. we need $\mathrm{eq}_{\mathrm{y,x}}$ for $\mathrm{eq}_{\mathrm{x,y}} \Leftrightarrow \mathrm{eq}_{\mathrm{y,x}}$)
  - simplest approach is to add these properties per pair of original variables

Ackermannization (Eliminating Equality) II

- This encoding of the formula $\mathrm{x}{=}\mathrm{y} \wedge \mathrm{y}{=}\mathrm{z} \Rightarrow \neg(\mathrm{x}{=}\mathrm{z})$ gives:

$$\mathrm{eq}_{\mathrm{x,x}} \wedge \mathrm{eq}_{\mathrm{y,y}} \wedge \mathrm{eq}_{\mathrm{z,z}} \wedge$$
$$(\mathrm{eq}_{\mathrm{x,y}} \Leftrightarrow \mathrm{eq}_{\mathrm{y,x}}) \wedge (\mathrm{eq}_{\mathrm{x,z}} \Leftrightarrow \mathrm{eq}_{\mathrm{z,x}}) \wedge (\mathrm{eq}_{\mathrm{y,z}} \Leftrightarrow \mathrm{eq}_{\mathrm{z,y}}) \wedge$$
$$(\mathrm{eq}_{\mathrm{x,y}} \wedge \mathrm{eq}_{\mathrm{y,z}} \Rightarrow \mathrm{eq}_{\mathrm{x,z}}) \wedge (\mathrm{eq}_{\mathrm{y,z}} \wedge \mathrm{eq}_{\mathrm{z,x}} \Rightarrow \mathrm{eq}_{\mathrm{y,x}}) \wedge (\mathrm{eq}_{\mathrm{z,x}} \wedge \mathrm{eq}_{\mathrm{x,y}} \Rightarrow \mathrm{eq}_{\mathrm{z,y}})$$
$$\wedge (\mathrm{eq}_{\mathrm{x,y}} \wedge \mathrm{eq}_{\mathrm{y,z}} \Rightarrow \neg \mathrm{eq}_{\mathrm{x,z}})$$

Is the result equisatisfiable with the original formula?

- *Yes* (both are now unsatisfiable); in fact, this works in general
  - We can map between models for the original and resulting formulas (how?)
- In the above, we skipped some transitivity cases (why is this OK?)
- Can you see further potential optimisations to this encoding?
  - think about what the SAT solver will do next; also see the exercise sheet

Extension: Equality and Uninterpreted Functions

- Consider now a signature consisting of:
  - any number of uninterpreted sort, plus the interpreted sort Bool
  - interpreted Bool constant symbols $\top$ and $\bot$ with the usual meaning
  - any number of uninterpreted function symbols over these sorts
  - equality functions on each uninterpreted sort, written (overloaded) $=$
- Consider again *quantifier-free* formulas over this signature
  - for example, $\mathrm{y}{=}\mathrm{f}(\mathrm{z}) \wedge \mathrm{x}{=}\mathrm{f}(\mathrm{f}(\mathrm{z})) \wedge \neg(\mathrm{x}{=}\mathrm{f}(\mathrm{y}))$
- We can also encode SAT problems in this syntax into propositional SAT
- We will again use an Ackermann encoding to eliminate functions

Ackermannization (Uninterpreted Functions) I

- This time, we eliminate applications of *uninterpreted functions*
  - for simplicity, we'll deal with unary function symbols on this slide
- For each function application $\mathrm{f}(\mathrm{x})$ in the original formula with only variables as parameters:
  - introduce a fresh variable $\mathrm{f_x}$ of the same sort as the return sort of f
  - replace all occurrences of $\mathrm{f}(\mathrm{x})$ in the original formula, with $\mathrm{f_x}$
  - repeat this process until the original formula contains no function symbols
- For example, given the formula $\mathrm{y}{=}\mathrm{f}(\mathrm{z}) \wedge \mathrm{x}{=}\mathrm{f}(\mathrm{f}(\mathrm{z})) \wedge \neg(\mathrm{x}{=}\mathrm{f}(\mathrm{y}))$ we obtain $\mathrm{y}{=}\mathrm{f_z} \wedge \mathrm{x}{=}\mathrm{f_{fz}} \wedge \neg(\mathrm{x}{=}\mathrm{f_y})$ where $\mathrm{f_z}$, $\mathrm{f_{fz}}$, $\mathrm{f_y}$ are fresh variables
- Is the result equisatisfiable with the original formula?
- *No*: we are missing the fact that the original functions *were functions*
  - e.g. $\mathrm{y}{=}\mathrm{f}(\mathrm{z}) \Rightarrow \mathrm{f}(\mathrm{y}){=}\mathrm{f}(\mathrm{f}(\mathrm{z}))$

Ackermannization (Uninterpreted Functions) II

- Idea: each time we introduce a variable f_x to replace application $f(x)$
  - for each variable f_y *already introduced* to replace an application $f(y)$ of f: conjoin the formula $(x{=}y \Rightarrow f_x = f_y)$ to the overall formula
  - (for functions of higher arity, add extra equalities on the left-hand-side)
  - extra clauses in the resulting formula: quadratic in original applications of f
- For example, we rewrite $y{=}f(z) \wedge x{=}f(f(z)) \wedge \neg(x{=}f(y))$ to:

$$(y{=}z \Rightarrow f_y = f_z) \wedge (y{=}f_z \Rightarrow f_y = f_{fz}) \wedge (z{=}f_z \Rightarrow f_z = f_{fz}) \wedge$$
$$y{=}f_z \wedge x{=}f_{fz} \wedge \neg(x{=}f_y)$$

- The resulting formula is equisatisfiable with the original
  - We can map between models for the original and resulting formulas (how?)
- We can subsequently eliminate equalities via the previous technique

Finitely-Bounded Quantifiers

- In general, reasoning about quantifiers needs more than a SAT solver
  - note that propositional SAT is decidable, while first-order logic SAT is not
- However, *finitely-bounded quantifiers* can be (easily) handled
  - allow $\exists$ and $\forall$ quantification over known, finite sets
- Essentially, this works because of the following "equivalence":
  - $\forall x{:}T.A \equiv A[t_1/x] \wedge A[t_2/x] \wedge \ldots$ where $t_1, t_2, \ldots$ enumerate sort T
  - $\exists x{:}T.A \equiv A[t_1/x] \vee A[t_2/x] \vee \ldots$ where $t_1, t_2, \ldots$ enumerate sort T
  - if T is infinite, this doesn't lead to an effective algorithm in general
- If only *finitely many* instantiations are relevant, we can unroll this way
- For example, integers in constant ranges: $\forall x{:}\mathrm{Int}.\ 0 \leq x \wedge x \leq 4 \Rightarrow A$
  - outer-quantified variables: $\forall x{:}\mathrm{Int}.\ 0 \leq x \wedge x \leq 4 \Rightarrow (\forall y{:}\mathrm{Int}.\ x \leq y \wedge y \leq 4 \Rightarrow A)$
  - similarly, selection from other finite sets: $\forall x{:}\mathrm{Int}.\ x \in \{2,4,6,8\} \Rightarrow A$

Sudoku Solving as a SAT Problem

- How can we encode this (and other instances) as a SAT problem?

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Need to represent the rules and initial constraints, and get a solution

Sudoku Rules

- How do we encode the state of a cell? … the entire board?
  - easiest to design in terms of an integer variable $x_{(i,j)}$ for the value at cell (i,j)
- We define what it means for each value to occur in a set of nine:
  - $\texttt{eachValue}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) =$

    $\forall d{:}\text{Int}.\ d \in [1..9] \Rightarrow \exists x{:}\text{Int}.\ x \in \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\} \wedge x = d$
  - by expanding quantifiers, this is equivalently: $\bigwedge_{d=1}^{9} \bigvee_{i=1}^{9} x_i = d$
- Now define that each row, column and 3x3 grid contains each value:

$$\bigwedge_{i=1}^{9} \texttt{eachValue}(x_{(i,1)}, x_{(i,2)}, x_{(i,3)}, x_{(i,4)}, x_{(i,5)}, x_{(i,6)}, x_{(i,7)}, x_{(i,8)}, x_{(i,9)}) \wedge$$
$$\bigwedge_{j=1}^{9} \texttt{eachValue}(x_{(1,j)}, x_{(2,j)}, x_{(3,j)}, x_{(4,j)}, x_{(5,j)}, x_{(6,j)}, x_{(7,j)}, x_{(8,j)}, x_{(9,j)}) \wedge$$
$$\bigwedge_{i,j \in \{1,4,7\}} \texttt{eachValue}(x_{(i,j)}, x_{(i,j+1)}, x_{(i,j+2)}, x_{(i+1,j)}, x_{(i+1,j+1)}, x_{(i+1,j+2)},$$
$$x_{(i+2,j)}, x_{(i+2,j+1)}, x_{(i+2,j+2)})$$

Encoding Sudoku Rules to SAT

- Simplest formulas in this representation are of the form $x_{(i,j)} = d$
- We encode these using a Bool-sorted variable to represent each
  - i.e. 9 variables for each cell, each representing a "cell (i,j) has value d" fact
  - similar to Ackermannization, but we exploit special shape of equalities here
- With this representation, we should add constraints expressing that:
  - each cell has at least one value (how?)
  - each cell has at most one value (how?)
- A resulting model returned by the SAT solver will determine a solution
- Alternative definition: $\texttt{eachValue}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) = \bigwedge_{1 \le i < j \le 9} \bigwedge_{d=1}^{9} \neg(x_i = d) \vee \neg(x_j = d)$
- Is this definition likely to yield better or worse performance?

Incorporating (Bounded) Integers

- Ackermannization works for eliminating uninterpreted functions
- What about general *interpreted functions*, such as integer $+$?
  - Ackermannization loses many properties about general interpreted functions
- One approach is to support theory reasoning *natively* (next lecture)
- For many integer problems, *bounded-size integers* may be suitable
  - In this case, an alternative approach is available to us, known as *bit-blasting*
  - Successfully used in industrial hardware and software analysis tools
- idea: a 32-bit integer n is really just a sequence of 32 individual bits
  - A *bit-vector* is such a sequence of Bool-sorted variables (e.g. $\mathrm{n}_{[0]},\mathrm{n}_{[1]},\ldots,\mathrm{n}_{[31]}$)
- What about *interpreted bitwise/arithmetic functions* on integers?
  - e.g. $(\mathrm{m}\ \&\ \mathrm{n})=0$, $(\mathrm{m} >> 1)=1$, $(\mathrm{m} + \mathrm{n} < 1024)$

Bit-Blasting I

- We use a *bit-vector representation* to encode an integer sort Int32
  - each integer-typed *term* in the original formula which is not an interpreted constant (e.g. 0) is associated with a fresh bit-vector $\mathrm{n}_{[0]},\mathrm{n}_{[1]},\ldots,\mathrm{n}_{[31]}$
  - e.g. $(\mathrm{m}\ \&\ \mathrm{n})=0$ needs a bit-vector for each of m, n, $(\mathrm{m}\ \&\ \mathrm{n})$ (but not 0)
  - assume these are $\mathrm{m}_{[0]},\mathrm{m}_{[1]},\ldots,\mathrm{m}_{[31]}$, $\mathrm{n}_{[0]},\mathrm{n}_{[1]},\ldots,\mathrm{n}_{[31]}$, $\mathrm{a}_{[0]},\mathrm{a}_{[1]},\ldots,\mathrm{a}_{[31]}$ respectively
- For each function application of return sort Int32, generate constraint relating its bit-vector to the bit-representations of the arguments
  - for interpreted constant arguments, use their translation as bit values directly
- For example, for the $(\mathrm{m}\ \&\ \mathrm{n})$ function application, generate:
  $(\mathrm{a}_{[0]}\Leftrightarrow\mathrm{m}_{[0]}\wedge\mathrm{n}_{[0]})\ \wedge\ (\mathrm{a}_{[1]}\Leftrightarrow\mathrm{m}_{[1]}\wedge\mathrm{n}_{[1]})\ \wedge\ \ldots\ \wedge\ (\mathrm{a}_{[31]}\Leftrightarrow\mathrm{m}_{[31]}\wedge\mathrm{n}_{[31]})$

Bit-Blasting II

- For functions of return sort Bool, generate constraints defining them in terms of (the bit-representations of) their arguments.
- For example, for the $(\mathrm{m}\ \&\ \mathrm{n})=0$ function application, generate:
  $(\mathrm{a}_{[0]}\Leftrightarrow\bot)\ \wedge\ (\mathrm{a}_{[1]}\Leftrightarrow\bot)\ \wedge\ \ldots\ \wedge\ (\mathrm{a}_{[31]}\Leftrightarrow\bot)$
- In the original formula, replace these Bool-sorted function applications with these defining constraints
- Then, to the resulting formula, conjoin all constraints generated from processing the Int32-sorted function applications (cf. previous slide)
- Overall, starting from our formula $(\mathrm{m}\ \&\ \mathrm{n})=0$ we obtain:
  $(\mathrm{a}_{[0]}\Leftrightarrow\bot)\ \wedge\ (\mathrm{a}_{[1]}\Leftrightarrow\bot)\ \wedge\ \ldots\ \wedge\ (\mathrm{a}_{[31]}\Leftrightarrow\bot)\ \wedge$
  $(\mathrm{a}_{[0]}\Leftrightarrow\mathrm{m}_{[0]}\wedge\mathrm{n}_{[0]})\ \wedge\ (\mathrm{a}_{[1]}\Leftrightarrow\mathrm{m}_{[1]}\wedge\mathrm{n}_{[1]})\ \wedge\ \ldots\ \wedge\ (\mathrm{a}_{[31]}\Leftrightarrow\mathrm{m}_{[31]}\wedge\mathrm{n}_{[31]})$

Bit-Blasting Arithmetic Functions I

- For bitwise operators, the generated constraints are fairly simple
  - inequalities and equalities are also straightforward to define
- What about *arithmetic operators*, such as addition or multiplication?
  - these can be defined by translating a *circuit definition* of the operation
  - e.g. take a 32-bit adder circuit, and encode its input/output functionality
- e.g. for a simple adder, with operands m,n and result a (bit-vectors)
- helper "carry" definition: $C(m,n,0) = \bot$, and for any $0<i<32$,
  $C(m,n,i) = m_{[i-1]} \wedge n_{[i-1]} \vee (\neg(m_{[i-1]} \Leftrightarrow n_{[i-1]}) \wedge (C(m,n,i-1)))$
- Now we can define the addition result, using (expanding $C(...)$):

$$(a_{[0]} \Leftrightarrow \neg(\neg(m_{[0]} \Leftrightarrow n_{[0]}) \Leftrightarrow C(m,n,0))) \wedge$$
$$(a_{[1]} \Leftrightarrow \neg(\neg(m_{[1]} \Leftrightarrow n_{[1]}) \Leftrightarrow C(m,n,1))) \wedge \ldots$$

Bit-Blasting Arithmetic Functions II

- In principle, the same approach adapts to other operators
  - subtraction is easy to define – how?
- However, encoding a multiplier is expensive (also division, modulo)
  - each step of the recursion employs further (simpler) bit-operations
  - number of variables and clauses rapidly increases
- A 32-bit multiplier, encoded reasonably efficiently, requires:
  - 5089 variables and 17057 clauses (in CNF) [Kroening and Strichman 2008]
  - for 64-bits, this becomes 20417 variables and 68929 clauses
  - problems involving multiplication, division etc. can easily become intractable
- The formula $a \cdot b = c \wedge \neg(b \cdot a = c) \wedge x < y \wedge y < x$ is
  - unsat for two independent reasons (each easy to see as an integer problem)
  - bit-blasted first terms explode, “tricking” SAT solver into focusing efforts there

Incremental Bit-Blasting

- To improve performance, we can vary the bit-blasting approach
- Introduce all new variables, rewrite original formula, but:
- Leave out some of the "expensive" constraints
  - for example, do not provide definitions from multiplications, initially
- The resulting formula is strictly weaker than the original
  - if the SAT solver manages to return `unsat`, this result is known to be correct
  - if we get a `sat` and a model, check whether it satisfies the *missing constraints*
  - If so, then we can indeed return this result
  - If not, add the unsatisfied constraints to the formula, and repeat the process
- Cheaper "reasons" for a `sat` or `unsat` result can be found
- e.g. the easier `unsat` reason for $\mathrm{a}\cdot\mathrm{b}=\mathrm{c} \wedge \neg(\mathrm{b}\cdot\mathrm{a}=\mathrm{c}) \wedge \mathrm{x}<\mathrm{y} \wedge \mathrm{y}<\mathrm{x}$

Encoding Problems to SAT - Summary

- We have seen techniques for encoding non-propositional problems
- Equality and uninterpreted functions via Ackermannization
- Finitely-bounded quantification via expansion of cases
- Bounded integer arithmetic via bit-blasting
- These techniques are all employed in practice
  - In some domains, they are very successful (e.g. hardware verification)
  - However, limitations in expressiveness and performance can be easily hit
- General theory reasoning requires custom, native support
- In the next section: incorporating this support, via SMT solving

Encoding Problems to SAT – Some References

- *Handbook of Satisfiability.* Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (2009)
- Ackermannization:
  - *Solvable Cases of the Decision Problem.* W. Ackermann (1955)
  - *Model Based Theory Combination.* Leonardo de Moura and Nikolaj Bjørner (2008)
- Soduko: *A SAT-based Sudoku Solver.* Tjark Weber (2005)
- Bit Vectors: *Decision Procedures: An Algorithmic Point of View.* Daniel Kroening and Ofer Strichman (2008)
  - *See Chapter 6 "Bit Vectors" or search online for "Bit-Vector Arithmetic"*
- General SAT Developments:
  - *SAT-solving in practice.* Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson (2008)
  - *Successful SAT Encoding Techniques.* Magnus Björk (2009)