User-Propagators for Custom Theories in SMT Solving

– In a Nutshell –

Nikolaj Bjørner[1], **Clemens Eisenhofer**[2], and Laura Kovács[2]

[1] Microsoft Research, USA [2] TU Wien, Austria

Recap

Satisfiability Modulo Theories (SMT) solvers support reasoning in (fragments of) first-order logic with native support for a wide range of theories (integers, arrays, strings, bit-vectors, ADTs, ...).

Recap

Satisfiability Modulo Theories (SMT) solvers support reasoning in (fragments of) first-order logic with native support for a wide range of theories (integers, arrays, strings, bit-vectors, ADTs, ...).
$\Rightarrow$ Very important for program verification. e.g.,

```
int32 i1, i2;
...
assume(i1 > 0);
arr[0] = 1;
arr[i1 + i2] = 2;
assert(arr[0] = 1);
```

Recap

Satisfiability Modulo Theories (SMT) solvers support reasoning in (fragments of) first-order logic with native support for a wide range of theories (integers, arrays, strings, bit-vectors, ADTs, ...).

$\Rightarrow$ Very important for program verification. e.g.,

```
int32 i1, i2;
...
assume(i1 > 0);
arr[0] = 1;
arr[i1 + i2] = 2;
assert(arr[0] = 1);
```

$$
\begin{aligned}
&\ldots \wedge \\
&i1 > 0 \wedge \\
\Rightarrow\ &arr_1 = store(arr_0, 0, 1) \wedge \\
&arr_2 = store(arr_1, i1 + i2, 2) \wedge \\
&select(arr_2, 0) \neq 1
\end{aligned}
$$

Recap

Satisfiability Modulo Theories (SMT) solvers support reasoning in (fragments of) first-order logic with native support for a wide range of theories (integers, arrays, strings, bit-vectors, ADTs, ...).
$\Rightarrow$ Very important for program verification. e.g.,

```
int32 i1, i2;
...
assume(i1 > 0);
arr[0] = 1;
arr[i1 + i2] = 2;
assert(arr[0] = 1);
```

$\Rightarrow$

$$
\begin{aligned}
&\ldots \wedge \\
&i1 > 0 \wedge \\
&arr_1 = store(arr_0, 0, 1) \wedge \\
&arr_2 = store(arr_1, i1 + i2, 2) \wedge \\
&select(arr_2, 0) \neq 1
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
&array_0 \mapsto \langle 0, ..., 0 \rangle, \\
&array_1 \mapsto \langle 1, ..., 0 \rangle, \\
&array_2 \mapsto \langle 2, ..., 0 \rangle, \\
&i1 \mapsto 2^{31}, \\
&i2 \mapsto 2^{31}
\end{aligned}
$$

Relevancy

My talk might be of *special interest* for you if ...

- You need some "crazy" theory that is not directly supported by your SMT solver
- You encounter bad performance on solving some big problem in SMT
- You use an SMT-solver as a core for your own solver
- You are not happy with how SMT solvers do certain things internally (partially)

Relevancy

My talk might be of *special interest* for you if ...

- You need some "crazy" theory that is not directly supported by your SMT solver
- You encounter bad performance on solving some big problem in SMT
- You use an SMT-solver as a core for your own solver
- You are not happy with how SMT solvers do certain things internally (partially)

Note: This talk should only convey the general idea; just relax and check if you might utilize the user-propagator for your own needs. We are looking for use-cases and can add new features if required. I am happy to discuss things in more detail individually after the talk :-)!

Topics

More precisely, we can use user-propagation for (incomplete list):

- Defining *custom theories* (finite or infinite)
  - e.g., graphs, posets, (new) strings
- *Lazily instantiating* potentially unused axioms (less required memory - more speed)
  - e.g., We will see a simple example
- Easy and high performance *constraint satisfaction problem* solving
  - e.g., Encoding the *n*-queens problem (vastly increases performance/decreases memory
- Implementing custom variable selection/assignment *heuristics*
- Efficiently/Easily expressing *aggregates and scheduling constraints*
  - e.g.,
    $$3 \cdot |\{(i,j) \mid i < j \wedge x_i + x_j = 42 \wedge (x_i > 30 \vee x_j > 30)\}| + |\{(i,j) \mid i < j \wedge x_i + x_j = 42 \wedge \neg(x_i > 30 \vee x_j > 30)\}|$$
- Speeding-up propositional *AllSAT*
- Speeding-up *quantifier checking* / Improving quantifier *instantiation*
- Ease *integration* of Z3 in other tools (use Z3 as a core for other solvers)
- Extend Z3 with *additional semantics* (partially)
  - e.g., Modal Logic (might be a stupid idea, but why not ...)
- *Logging/Debugging* (potentially replaying proofs – let's see) intern Z3 actions
- ...

Topics

More precisely, we can use user-propagation for (incomplete list):

- Defining *custom theories* (finite or infinite)
  - e.g., graphs, posets, (new) strings
- *Lazily instantiating* potentially unused axioms (less required memory - more speed)
  - e.g., We will see a simple example
- Easy and high performance *constraint satisfaction problem* solving
  - e.g., Encoding the *n*-queens problem (vastly increases performance/decreases memory
- Implementing custom variable selection/assignment *heuristics*
- Efficiently/Easily expressing *aggregates and scheduling constraints*
  - e.g.,
    $3 \cdot |\{(i,j) \mid i < j \wedge x_i + x_j = 42 \wedge (x_i > 30 \vee x_j > 30)\}| + |\{(i,j) \mid i < j \wedge x_i + x_j = 42 \wedge \neg(x_i > 30 \vee x_j > 30)\}|$
- Speeding-up propositional *AllSAT*
- Speeding-up *quantifier checking* / Improving quantifier *instantiation*
- Ease *integration* of Z3 in other tools (use Z3 as a core for other solvers)
- Extend Z3 with *additional semantics* (partially)
  - e.g., Modal Logic (might be a stupid idea, but why not ...)
- *Logging/Debugging* (potentially replaying proofs – let's see) intern Z3 actions
- ...
- *Educating* SMT-Solving
- *Drafting* new decision procedures in a rather easy "framework"

Q: How does it work?

Q: How does it work?

A:

1. Build formula via the solver's API.

$$P(s1 \mathbin{+\!\!+} s2) \wedge (z + 3 > c \vee c - 5 = x)$$

Q: How does it work?

A:

1. Build formula via the solver's API.
2. Register all (ground) terms, that are relevant for the custom theory.

$$P(s1 \mathbin{+\!\!+} s2) \wedge (z + 3 > c \vee c - 5 = x)$$

Q: How does it work?

A:

1. Build formula via the solver's API.
2. Register all (ground) terms, that are relevant for the custom theory.
3. Attach callbacks (= functions) to desired events.

$$\{\ \mathit{Fixed} = \text{Some-Code-In-C++},\ \mathit{Eq} = \text{Or-Some-Code-In-Python},\ \ldots\ \}$$

Q: How does it work?

A:

1. Build formula via the solver's API.
2. Register all (ground) terms, that are relevant for the custom theory.
3. Attach callbacks (= functions) to desired events.
4. Checking satisfiability $\Rightarrow$ Triggers attached callbacks in case the reasoner processes a registered term.

Q: How does it work?

A:

1. Build formula via the solver's API.
2. Register all (ground) terms, that are relevant for the custom theory.
3. Attach callbacks (= functions) to desired events.
4. Checking satisfiability $\Rightarrow$ Triggers attached callbacks in case the reasoner processes a registered term.
5. The callbacks may interfere if required.

How can the callbacks interfere?

How can the callbacks interfere?

A:

- Adding conflicts:
  e.g., "You (the solver) assigned $\langle z \mapsto 3, x \mapsto 7, `c - 5 = x` \mapsto \bot \rangle$ but I don't want that. Please backtrack!"

How can the callbacks interfere?

A:

- Adding conflicts
- Adding further constraints ("propagating lemmas"):
  e.g., "(Some parts of) Your current assignments enforces, that z must be greater than c. Please keep that in mind."

How can the callbacks interfere?

A:

- Adding conflicts
- Adding further constraints ("propagating lemmas")
- Interfere in variable selection heuristics:
  e.g., "If you cannot derive anything any more without guessing then try $`c - 5 = x` \mapsto \bot$ next."

Q: To which events can we attach callbacks?

Q: To which events can we attach callbacks?

A:

- fixed: "I just guessed/deduced that x must be assigned to $\top$ (resp., 5). Is that fine for you?"

Q: To which events can we attach callbacks?

A:

- fixed: "I just guessed/deduced that x must be assigned to $\top$ (resp., 5). Is that fine for you?"
- eq: "x and y have to be equal given the current assignment. Do you agree?"

Q: To which events can we attach callbacks?

A:

- fixed: "I just guessed/deduced that x must be assigned to $\top$ (resp., 5). Is that fine for you?"
- eq: "x and y have to be equal given the current assignment. Do you agree?"
- final: "I am done assigning truth-values. Please check once more if this is fine with the theory you have in mind?"

Q: To which events can we attach callbacks?

A:

- fixed: "I just guessed/deduced that x must be assigned to $\top$ (resp., 5). Is that fine for you?"
- eq: "x and y have to be equal given the current assignment. Do you agree?"
- final: "I am done assigning truth-values. Please check once more if this is fine with the theory you have in mind?"
- Further: push, pop, fresh, decide, diseq, and created

A CS flavoured example:

Consider 1000 bit-vectors x_i each with 32 bits. All of them are disjoint.

A CS flavoured example:

Consider 1000 bit-vectors x_i each with 32 bits. All of them are disjoint.

Adding $\bigwedge_{1 \leq i \leq j \leq 1000} x_i \neq x_j$ adds 499500 disequalities although the chance of a collision with a random assignment is far below 1%!

A CS flavoured example:

Consider 1000 bit-vectors x_i each with 32 bits. All of them are disjoint.

Adding $\bigwedge_{1 \leq i \leq j \leq 1000} x_i \neq x_j$ adds 499500 disequalities although the chance of a collision with a random assignment is far below 1%!

$\Rightarrow$ Add $x_i \neq x_j$ only if x_i and x_j are assigned to equal values.

A CS flavoured example:

Consider 1000 bit-vectors x_i each with 32 bits. All of them are disjoint.

Adding $\bigwedge_{1 \leq i \leq j \leq 1000} x_i \neq x_j$ adds 499500 disequalities although the chance of a collision with a random assignment is far below 1%!

$\Rightarrow$ Add $x_i \neq x_j$ only if x_i and x_j are assigned to equal values.

Nice, however it is called Alpine **Verification** Meeting, *not* Alpine **Constraint Satisfaction** Meeting ...

However, . . .

consider the variant: We have disjoint intervals $[addr_i, addr_i + size_i]$:

$$\bigwedge_{1 \leq i < j \leq n} addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i$$

However, . . .

consider the variant: We have disjoint intervals $[addr_i, addr_i + size_i]$:

$$\bigwedge_{1 \leq i < j \leq n} addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i$$

$\Rightarrow$ Add $addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i$ only if interval i and j are intersecting.

fixed: "Hey, I just assigned the interval I_1 to $[1, 4]$ and the interval I_2 to $[2, 5]$. Is this fine for you?" >> "No, either I_1 has to be strictly for I_2 or vice-versa"

However, . . .

consider the variant: We have disjoint intervals $[addr_i, addr_i + size_i]$:

$$\bigwedge_{1 \leq i < j \leq n} addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i$$

$\Rightarrow$ Add $addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i$ only if interval i and j are intersecting.

fixed: "Hey, I just assigned the interval I_1 to $[1, 4]$ and the interval I_2 to $[2, 5]$. Is this fine for you?" >> "No, either I_1 has to be strictly for I_2 or vice-versa"
This is an axiom in the verification tool alive2 to ensure that allocated memory blocks are disjoint.

- Quadratic memory requirement.
- Axiom probably not violated $\Rightarrow$ Solver should not focus on it.
- Time consuming to generate the axiom.

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

$$A \wedge \forall addr_1, \ldots size_n : ((\bigwedge_{1 \leq i \leq j \leq n} addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i) \Rightarrow B)$$

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

$$A \wedge \forall addr_1, \ldots size_n : ((\bigwedge_{1 \le i \le j \le n} addr_i + size_i \le addr_j \vee addr_j + size_j \le addr_i) \Rightarrow B)$$

We transform it to $A \wedge \forall addr_1, \ldots size_n : (disjoint(addr_1, \ldots, size_n) \Rightarrow B)$
and observe also the truth value of $disjoint(\ldots)$ and distinguish:

- $disjoint(addr_1, \ldots, size_n) \mapsto \top$ but at least two intervals i and j intersect

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

$$A \wedge \forall addr_1, \ldots size_n : ((\bigwedge_{1 \le i \le j \le n} addr_i + size_i \le addr_j \vee addr_j + size_j \le addr_i) \Rightarrow B)$$

We transform it to $A \wedge \forall addr_1, \ldots size_n : (disjoint(addr_1, \ldots, size_n) \Rightarrow B)$
and observe also the truth value of $disjoint(\ldots)$ and distinguish:

- $disjoint(addr_1, \ldots, size_n) \mapsto \top$ but at least two intervals i and j intersect:
  Propagate (overapproximation):

$$disjoint(addr_1, \ldots, size_n) \Rightarrow (addr_i + size_i \le addr_j \vee addr_j + size_j \le addr_i)$$

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

$$A \wedge \forall addr_1, \ldots size_n : ((\bigwedge_{1 \le i \le j \le n} addr_i + size_i \le addr_j \vee addr_j + size_j \le addr_i) \Rightarrow B)$$

We transform it to $A \wedge \forall addr_1, \ldots size_n : (disjoint(addr_1, \ldots, size_n) \Rightarrow B)$
and observe also the truth value of $disjoint(\ldots)$ and distinguish:

- $disjoint(addr_1, \ldots, size_n) \mapsto \top$ but at least two intervals i and j intersect:
  Propagate (overapproximation):

  $$disjoint(addr_1, \ldots, size_n) \Rightarrow (addr_i + size_i \le addr_j \vee addr_j + size_j \le addr_i)$$

- $disjoint(addr_1, \ldots, size_n) \mapsto \bot$ but at no intervals intersect

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

$$A \wedge \forall addr_1, \ldots size_n : ((\bigwedge_{1 \leq i \leq j \leq n} addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i) \Rightarrow B)$$

We transform it to $A \wedge \forall addr_1, \ldots size_n : (disjoint(addr_1, \ldots, size_n) \Rightarrow B)$
and observe also the truth value of $disjoint(\ldots)$ and distinguish:

- $disjoint(addr_1, \ldots, size_n) \mapsto \top$ but at least two intervals i and j intersect:
  Propagate (overapproximation):

  $$disjoint(addr_1, \ldots, size_n) \Rightarrow (addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i)$$

- $disjoint(addr_1, \ldots, size_n) \mapsto \bot$ but at no intervals intersect:
  Propagate (underapproximation):

  $$(addr_{\pi(1)} + size_{\pi(1)} \leq \ldots \leq addr_{\pi(n)}) \Rightarrow disjoint(addr_1, \ldots, size_n)$$

  where π is the current ordering of the addresses.

. . . but what if the part that should be lazily instantiated is not a ground formula occurring in a top-level conjunction?

$$A \wedge \forall addr_1, \ldots size_n : ((\bigwedge_{1 \leq i \leq j \leq n} addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i) \Rightarrow B)$$

We transform it to $A \wedge \forall addr_1, \ldots size_n : (disjoint(addr_1, \ldots, size_n) \Rightarrow B)$
and observe also the truth value of $disjoint(\ldots)$ and distinguish:

- $disjoint(addr_1, \ldots, size_n) \mapsto \top$ but at least two intervals i and j intersect:
  Propagate (overapproximation):

  $$disjoint(addr_1, \ldots, size_n) \Rightarrow (addr_i + size_i \leq addr_j \vee addr_j + size_j \leq addr_i)$$

- $disjoint(addr_1, \ldots, size_n) \mapsto \bot$ but at no intervals intersect:
  Propagate (underapproximation):

  $$(addr_{\pi(1)} + size_{\pi(1)} \leq \ldots \leq addr_{\pi(n)}) \Rightarrow disjoint(addr_1, \ldots, size_n)$$

  where π is the current ordering of the addresses.
- Else we are fine

Topics (Again)

- Defining *custom theories* (finite or infinite)
  - e.g., graphs, posets, (new) strings
- *Lazily instantiating* potentially unused axioms (less required memory - more speed)
  - e.g., We will see a simple example
- Easy and high performance *constraint satisfaction problem* solving
  - e.g., Encoding the *n*-queens problem (vastly increases performance/decreases memory
- Implementing custom variable selection/assignment *heuristics*
- Efficiently/Easily expressing *aggregates and scheduling constraints*
  - e.g.,
    $$3 \cdot |\{(i,j) \mid i < j \wedge x_i + x_j = 42 \wedge (x_i > 30 \vee x_j > 30)\}| + |\{(i,j) \mid i < j \wedge x_i + x_j = 42 \wedge \neg(x_i > 30 \vee x_j > 30)\}|$$
- Speeding-up propositional *AllSAT*
- Speeding-up *quantifier checking* / Improving quantifier *instantiation*
- Ease *integration* of Z3 in other tools (use Z3 as a core for other solvers)
- Extend Z3 with *additional semantics* (partially)
  - e.g., Modal Logic (might be a stupid idea, but why not ...)
- *Logging/Debugging* (potentially replaying proofs – let's see) intern Z3 actions
- ...
- *Educating* SMT-Solving
- *Drafting* new decision procedures in a rather easy "framework"

Thank you for attending!

Questions?