Generalized and Efficient Array Decision Procedures
FMCAD, Austin, 2009

Leonardo de Moura and Nikolaj Bjørner
Microsoft Research
Verification/Analysis tools need some form of Symbolic Reasoning
Verification/Analysis tools need some form of **Symbolic Reasoning**

Many Flavors:

- SAT Solvers
- SMT Solvers
- First-order Theorem Provers
- Computer Algebra Systems
Is formula $F$ satisfiable modulo theory $T$?
Is formula $F$ satisfiable modulo theory $T$?

Arithmetic, Bit-vectors, Arrays, Inductive data-types, ....
Example:

\[ 1 > 2 \]

Satisfiable if the symbols 1, 2 and > are uninterpreted.

\[ M = \{ \bullet \} \]

\[ M(1) = M(2) = \bullet \]

\[ M(>) = \{ (\bullet, \bullet) \} \]

Unsatisfiable modulo the theory arithmetic
\[ b + 2 = c \quad \text{and} \quad f(\text{select}(\text{store}(a,b,3), c-2)) \neq f(c-b+1) \]
\[ b + 2 = c \quad \text{and} \quad f(\text{select(} \text{store}(a,b,3), c-2)) \neq f(c-b+1) \]
$b + 2 = c \text{ and } f(\text{select}(\text{store}(a,b,3), c-2) \neq f(c-b+1)$
Satisfiability Modulo Theories (SMT)

\[ b + 2 = c \quad \text{and} \quad f(\text{select}(\text{store}(a,b,3), c-2)) \neq f(c-b+1) \]
Applications

- Test case generation
- Verifying Compilers
- Predicate Abstraction
- Invariant Generation
- Type Checking
- Model Based Testing
Some Applications @ Microsoft

- Spec# Programming System
- HAVOC
- Hyper-V Virtualization
- Terminator T-2
- VCC
- NModel
- SLAM
- SpecExplorer
- SAGE
- Prefix
- F7
- Vigilante
What is a Theory?

A theory $T$ is a set of first-order sentences.

$F$ is satisfiable modulo $T$ iff $T \cup F$ is satisfiable.
\( \forall a, i, v. \ select(store(a, i, v), i) = v \)

\( \forall a, i, j, v: \ i = j \lor \ select(store(a, i, v), j) = select(a, j) \)
We say store is a combinator.
Array Theory: a more familiar notation

∀a, i, v. select(store(a, i, v), i) = v

∀a, i, j, v: i = j ∨ select(store(a, i, v), j) = select(a, j)

∀a, i, v. store(a, i, v)[i] = v

∀a, i, j, v: i = j ∨ store(a, i, v)[j] = a[i]
Why array theory is useful?

It is used to model the memory in Hardware/Software verification/analysis tools.
\( \forall a, b: (\forall i: a[i] = b[i]) \Rightarrow a = b \)
Arrays are actually “maps”

We have arrays from $T_1$ to $T_2$

$T_1$ does not need to be the Integers
Models for arrays as “finite graphs”

\[ a = \text{store}(b, 0, 5), \quad b = \text{store}(c, 1, 10), \quad c[0] = 2 \]

\[ M(a) = \{ 0 \rightarrow 5, \ 1 \rightarrow 10, \ \text{else} \rightarrow 0 \} \]
\[ M(b) = \{ 0 \rightarrow 2, \ 1 \rightarrow 10, \ \text{else} \rightarrow 0 \} \]
\[ M(c) = \{ 0 \rightarrow 2, \ \text{else} \rightarrow 0 \} \]
1962 - McCarthy proposes the Basic Array Theory.

1968 - Kaplan solves the satisfiability problem.


2001 - Stump, Barrett, Dill and Levitt propose a procedure for extentional arrays.

2005 - Lazy instantiation is used in Yices (it wins all array divisions in SMT-COMP from 2005 - 2007).

2005 - Kapur and Zarba propose the reduction approach (many array-like theories are described).

2006 - Bradley, Manna and Sipma propose a procedure for a rich decidable array fragment.
2008 - Goel, Krstic and Fuchs formalize the lazy instantiation approach.

2008 - Bofill, Nieuwenhuis, Oliveras, Rodriguez-Carbonell and Rubio propose the store-reduction approach

“Model-Based” approaches:

2007 - Ganesh and Dill, “a decision procedure for bitvectors and arrays”, CAV’07

2008 - Brummayer and Biere, “lemmas on demand for the extensional theory of arrays”, SMT’08
A “Timeline” (Related Work)

“Rewrite-Based” approaches:

2002 - Lynch and Morawska, “Automatic Decidability”, LICS
2005 - Armando, Bonacina, Ranise and Schulz propose the rewrite based approach.

Arrays in hardware verification:

1994 - Burch and Dill, “Automatic Verification of pipelined microprocessor control”, CAV
2006 - Manolios, Srinivasan, Vroon, “Automatic memory reductions for RTL model verification”, ICCAD

More relevant work can be found in our paper...
Recipe: Given a formula $F$

1) Collect all array terms in $F$
2) Collect all indices in $F$
3) Instantiate array axioms using 1 and 2
   $$F' = F \cup \text{Instances}$$
4) Execute EUF solver on $F'$

Array theory is a local theory extension.
$a = \text{store}(b, i, v), a[j] \neq v, c[k] = v, i = j$

array terms: $a, b, \text{store}(b, i, v), c$

indices: $i, j, k$
Naïve instantiation: Example

\[ a = store(b, i, v), a[j] \neq v, c[k] = v, i = j \]

array terms: \( a, b, store(b, i, v), c \)
indices: \( i, j, k \)

Instances:

\[ store(a, i, v)[i] = v, store(a, j, v)[j] = v, \ldots \]
\[ i = j \lor store(a, i, v)[j] = a[i], \ldots \]

Problem: Many useless instances!
Naïve instantiation: Example

\[ a = \text{store}(b, i, v), \ a[j] \neq v, \ c[k] = v, \ i = j \]

Array terms: \( a, b, \text{store}(b, i, v), c \)

Indices: \( i, j, k \)

Instances:

\[ \text{store}(a, i, v)[i] = v, \ \text{store}(a, j, v)[j] = v, \ldots \]

\[ i = j \lor \text{store}(a, i, v)[j] = a[i], \ldots \]

Problem: Many useless instances!

Lazy instantiation: select a small subset of instances.
(more later)
Our contributions

A generalization of the Array theory
CAL: Combinatory Array Logic

New filters for minimizing the number of instances

A simple architecture for non-stably infinite theories

We want arrays of bit-vectors.
∀ v, i: K(v)[i] = v
∀ a_1, ..., a_n, i: map_f(a_1, ..., a_n)[i] = f(a_1[i], ..., a_n[i])
∀v,i: \( K(v)[i] = v \)

∀\( a_1, \ldots, a_n, i \): \( map_f(a_1, \ldots, a_n)[i] = f(a_1[i], \ldots, a_n[i]) \)

Suggested by Stump, Barrett, Dill, Levitt
Their procedure works for infinite-domain satisfiability.
∀ v, i: \( K(v)[i] = v \)

∀ \( a_1, \ldots, a_n, i \): \( \text{map}_f(a_1, \ldots, a_n)[i] = f(a_1[i], \ldots, a_n[i]) \)

“Family” of combinators.
We can instantiate it with any \( f \).
\( \text{map}_f \) is the pointwise function application

\[
\text{map}_f(\ldots v_1 v_2 v_3 v_4 v_5 \ldots, \ldots w_1 w_2 w_3 w_4 w_5 \ldots) = \ldots f(v_1, w_1) f(v_2, w_2) f(v_3, w_3) f(v_4, w_4) f(v_5, w_5) \ldots
\]
CAL is powerful: Sets as arrays

Set of T as an Array from T to Boolean

\[ \emptyset \equiv K(false) \]
\[ \{a\} \equiv store(\emptyset, a, true) \]
\[ a \in S \equiv S[a] \]
\[ S_1 \cup S_2 \equiv map_\vee(S_1, S_2) \]
\[ S_1 \cap S_2 \equiv map_\wedge(S_1, S_2) \]
Set of $T$ as an Array from $T$ to Boolean

\[
\emptyset \equiv K(\text{false})
\]
\[
\{a\} \equiv \text{store}(\emptyset, a, \text{true})
\]
\[
a \in S \equiv S[a]
\]
\[
S_1 \cup S_2 \equiv \text{map}_\lor(S_1, S_2)
\]
\[
S_1 \cap S_2 \equiv \text{map}_\land(S_1, S_2)
\]

But not cardinality $|S|$, power-set, ...
CAL is powerful: Bags as arrays

Bag of T as an Array from T to Integer

\[
\emptyset \equiv K(0) \\
\{a\} \equiv \text{store}(\emptyset, a, 1) \\
mult(a, B) \equiv B[a] \\
B_1 \oplus B_2 \equiv \text{map}_+(B_1, B_2) \\
B_1 \prod B_2 \equiv \text{map}_{\text{min}}(B_1, B_2)
\]
CAL is powerful: a multiplexer

\[ \text{map}_{ite}(\ldots, T, F, T, T, F, \ldots), \]

\[ \ldots, v_1, v_2, v_3, v_4, v_5, \ldots, \]

\[ \ldots, w_1, w_2, w_3, w_4, w_5, \ldots \]

\[ = \]

\[ \ldots, v_1, w_2, v_3, v_4, w_5, \ldots \]
Support for equality and uninterpreted functions (EUF)
Set of strongly disjoint theories (more later)
Clauses and literals
Boolean terms

\[ a \equiv t \quad -\quad a \text{ is a name for the term } t \]
\[ a : \sigma \quad -\quad a \text{ has sort } \sigma \]
\[ a \sim b \quad -\quad a \text{ and } b \text{ are equal in the current context} \]

\[
\begin{align*}
w_1 & \equiv f(v_1, \ldots, v_n), \quad w_2 \equiv f(v_1', \ldots, v_n') \quad v_1 \sim v_1', \ldots, v_n \sim v_n' \\
\therefore w_1 & \sim w_2
\end{align*}
\]
Array Saturation Rules
(this is not new)

\[
\begin{align*}
\text{idx} & \quad \frac{a \equiv \text{store}(b, i, v)}{a[i] \simeq v} \\
\downarrow & \quad \frac{a \equiv \text{store}(b, i, v), \quad w \equiv a'[j], \quad a \sim a'}{i \simeq j \lor a[j] \simeq b[j]} \\
\uparrow & \quad \frac{a \equiv \text{store}(b, i, v), \quad w \equiv b'[j], \quad b \sim b'}{i \simeq j \lor a[j] \simeq b[j]}
\end{align*}
\]

\[
\begin{align*}
\text{ext} & \quad \frac{a: (\sigma \Rightarrow \tau), \quad b: (\sigma \Rightarrow \tau)}{a \simeq b \lor a[k_{a,b}] \not\simeq b[k_{a,b}]}
\end{align*}
\]

\(\quad a \sim b \quad \) – \( a \) and \( b \) are equal in the current context
\(\quad a \equiv t \quad \) – \( a \) is a name for the term \( t \)
\(\quad a: (\sigma \Rightarrow \tau) \quad \) – \( a \) is an array from \( \sigma \) to \( \tau \)
Extensionality is applied to every pair of array constants.

Upwards propagation distributes index over all modifications of same array.
Extensionality is applied to every pair of array constants.

\[
\frac{a : (\sigma \Rightarrow \tau), \quad b : (\sigma \Rightarrow \tau)}{a \simeq b \lor a[k_{a,b}] \not\simeq b[k_{a,b}]}
\]

Upwards propagation distributes index over all modifications of same array.

\[
\frac{a \equiv \text{store}(b, i, v), \quad w \equiv b'[j], \quad b \sim b'}{i \simeq j \lor a[j] \simeq b[j]}
\]

Delay the application of ext and \( \uparrow \).

Only works for unsatisfiable instances.
Ignore “congruent” axiom instances

\[ i \simeq j \lor a[j] \simeq b[j] \]

\[ i' \simeq j' \lor a'[j'] \simeq b'[j'] \]

\[ a \sim a', \ b \sim b', \ i \sim i', \ \text{and} \ j \sim j' \]
Extensionality is applied to every pair of array constants. Restrict to constants asserted to be different or foreign.

We say \(a\) is foreign if there is \(b\) s.t. \(a \sim b\) and \(b\) is the argument of an uninterpreted function symbol.
Why do we need $\text{ext}_r$?

Example:

$$a = \text{store}(b, i, v), \ b[i] = v, \ f(a) \neq f(b)$$
We do not need to add the extensionality axiom for \((a,b)\) if they are already known to be disequal.

**Definition 9 (Already Disequal)** Given a state \(\Gamma\), \((a,b) \in \text{already-diseq}\) iff there are two definitions \(v_1 \equiv a_1[i_1]\) and \(v_2 \equiv a_2[i_2]\) in \(\Gamma\) such that \(v_1 \not\approx v_2\), \(a \sim a_1\), \(b \sim b_1\), and \(i_1 \sim i_2\).
We do not need to add the extensionality axiom for \((a,b)\) if they are already known to be disequal.

**Definition 9 (Already Disequal)** Given a state \(\Gamma\), \((a,b) \in\) already-diseq iff there are two definitions \(v_1 \equiv a_1[i_1]\) and \(v_2 \equiv a_2[i_2]\) in \(\Gamma\) such that \(v_1 \not\sim v_2\), \(a \sim a_1\), \(b \sim b_1\), and \(i_1 \sim i_2\).

*Typo in the paper! Should be \(b_1\)*
Scenario from software verification
Bunch of facts about the initial state of the heap
\( a_0[i_0] = v_0, \ a_0[i_1] = v_1, \ a_0[i_2] = v_2, \ldots \)

Perform a series of updates
\( a_1 = \text{store}(a_0, j_1, w_1), \ a_2 = \text{store}(a_1, j_2, w_2), \ldots \)

Check some property on the final heap
\( a_n[k] \neq v \)
store(a, i, v_1) = store(b, i, v_2), i \neq k, a[k] \neq b[k]

Definition 10 (Linearity) Given a state \( \Gamma \), the set non-linear of non-linear constants is the least set such that:
1. \( a_1 \equiv \text{store}(b_1, i_1, v_1), a_2 \equiv \text{store}(b_2, i_2, v_2), a_1 \) is not \( a_2 \) and \( a_1 \sim a_2 \) implies \( \{a_1, a_2\} \subseteq \text{non-linear} \),
2. \( a \equiv \text{store}(b, i, v) \) and \( a \in \text{non-linear} \) implies \( b \in \text{non-linear} \),
3. \( a \in \text{non-linear} \) and \( a \sim b \) implies \( b \in \text{non-linear} \).

We say \( a \) is linear if \( a \notin \text{non-linear} \).
\[ a \equiv \text{store}(b, i, v), \quad w \equiv b'[j], \quad b \sim b' \]

\[ \frac{i \approx j \land a[j] \approx b[j]}{i \approx j \lor a[j] \approx b'[j]} \]

\[ a \equiv \text{store}(b, i, v), \quad w \equiv b'[j], \quad b \sim b', \quad \boxed{b \in \text{non-linear}} \]

\[ \frac{i \approx j \lor a[j] \approx b[j]}{i \approx j \lor a[j] \approx b'[j]} \]
Effect on Benchmarks
Saturating CAL

\[ K \downarrow \quad a \equiv K(v), \quad w \equiv a'[j], \quad a \sim a' \]
\[ a[j] \simeq v \]

\[ \text{map} \downarrow \quad a \equiv \text{map}_f(b_1, \ldots, b_n), \quad w \equiv a'[j], \quad a \sim a' \]
\[ a[j] \simeq f(b_1[j], \ldots, b_n[j]) \]

\[ a \equiv \text{map}_f(b_1, \ldots, b_n), \quad w \equiv b'_k[j], \quad b_k \sim b'_k, \quad \text{for some} \quad k \in \{1, \ldots, n\} \]
\[ a[j] \simeq f(b_1[j], \ldots, b_n[j]) \]

\[ \epsilon \neq \quad v \equiv a[i], \quad i : \sigma, \quad i \text{ is not } \epsilon_\sigma \quad \epsilon_\sigma \neq i \]
\[ \epsilon_\sigma \neq i \]
\[ \epsilon \delta \quad \frac{a : (\sigma \Rightarrow \tau)}{a[\epsilon_\sigma] \simeq \delta_a} \]
Potentially unsound if $F$ only has models $M$ where $M(\sigma)$ is finite.
We also have a restricted version of map using linear stratification (see paper for details).

\[ a \simeq \text{map}_{ite}(a, b, c) \land b[j] \simeq \bot \land c[j] \simeq \top \]

Default-value extension (new theory symbol \( \delta \)), and alternative for \( \varepsilon \neq \) and \( \varepsilon \delta \)

\[
\begin{align*}
U\delta & \quad \text{if } a \equiv \text{store}(b, i, v) \\
& \quad \frac{}{\delta(a) \simeq \delta(b)} \\
K\delta & \quad \text{if } a \equiv K(v) \\
& \quad \frac{}{\delta(a) \simeq v} \\
\text{map}\delta & \quad \text{if } a \equiv \text{map}(b_1, \ldots, b_n) \\
& \quad \frac{}{\delta(a) \simeq f(\delta(b_1), \ldots, \delta(b_n))}
\end{align*}
\]
Theory combination in Z3

Efficient Core

Strongly disjoint theories + Uninterpreted functions

Strongly disjoint theory ≡ Sort disjoint
Examples: Arithmetic, Bitvectors and Booleans

\[ f(\top) \simeq w \land f(\bot) \simeq w \land f(v) \not\equiv w \]

All other theories are reduced to this core.
Not covered today: inductive datatypes.
**Conclusion**

Arrays are useful in practice.

They are used in many verification tools at Microsoft.

CAL is a useful extension of the array theory.

Simple combination architecture.
Efficient and easy to implement.
Arrays are useful in practice.

They are used in many verification tools at Microsoft.

CAL is a useful extension of the array theory.

Simple combination architecture.
Efficient and easy to implement.

Thank You!