Lemmas on Demand for the Extensional Theory of Arrays

Robert Brummayer Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

ABSTRACT

Deciding satisfiability in the theory of arrays, particularly in combination with bit-vectors, is essential for software and hardware verification. We precisely describe how the lemmas on demand approach can be applied to this decision problem. In particular, we show how our new propagation based algorithm can be generalized to the extensional theory of arrays. Our implementation achieves competitive performance.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving; B.5.2 [**Register-Transfer-Level Implementation**]: Verification; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Decision Procedure, Satisfiability, SAT, Arrays, SMT

1. INTRODUCTION

Reasoning about bit-vectors and arrays is an active area of research and is successfully applied to hardware and software verification. Particularly, deciding satisfiability of first-order formulas, with respect to theories, known as Satisfiability Modulo Theories (SMT) [15, 9, 16, 2, 11, 18], increasingly gains importance.

SMT approaches can be roughly divided into *eager* and *lazy* [19]. In the lazy *lemmas on demand* [12, 14, 1, 13, 10] approach, theory constraints are added on demand, rather than up front. The SAT solver is used as a black box and generates concrete assignments for the abstracted formula. If an assignment violates the theory, then a lemma that rules out this assignment is added as refinement. An important aspect of this approach is that the theory solver is used as consistency checker for concrete assignments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SMT '08, July 7-8, 2008, Princeton, New Jersey, USA
Copyright 2009 ACM 1-60558-440-9/09/02 ...$5.00.

Previous work [12, 14, 1, 13, 10] lacks a precise description on how lemmas on demand can be generated for arrays, particularly, if equalities on arrays are involved. Introducing array equalities, particularly inequalities, makes the theory extensional. Prior work on arrays typically maintains congruence on arrays, but not necessarily extensionality. Our approach handles both.

Extensionality has many applications in hardware and software verification. A typical example in hardware is conformance of a pipelined machine to a sequential implementation of its instruction set architecture, in particular with memory. On the software side, extensionality makes it easy to specify that algorithms have the same memory semantics. In both cases, the algorithms respectively machines are symbolically executed on one big array, which models memory. Finally, the memories are compared with the help of extensionality.

Previous approaches to decide extensionality use eager rewriting [20], which blows up in space, as our experiments confirm. Our main contribution is a novel lemmas on demand algorithm for the extensional theory of arrays. We focus on bit-vectors, but our approach can be easily generalized to other theories. We precisely describe our new propagation based algorithm for the non-extensional case, and generalize it to the extensional case. Finally, we report on experiments.

2. BACKGROUND

The theory of bit-vectors allows precise modelling of actual computation in hardware and software. It models modular arithmetic, e.g. addition is modulo 2^{32}, using two's complement representation. Other operations include comparison and logical operations, shifting, concatenation, and bit extraction.

Arrays are one of the most important data structures in computer science [17]. The basic operations on arrays are *read*, *write* and equality on array *elements*. With $read(a, i)$ we denote the value of array a at index i. With $write(a, i, e)$ we denote array a, overwritten at index i with element e. All other array elements remain the same. From the theory of uninterpreted functions $\mathcal{EUF}$ we inherit the array congruence axiom:

$$\text{(A1)} \quad a = b \wedge i = j \quad \Rightarrow \quad read(a, i) = read(b, j)$$

Our approach does not need explicit congruence closure. The remaining non-extensional axioms of the theory of arrays are [5, 4]:

$$\text{(A2)} \quad i = j \quad \Rightarrow \quad read(write(a,i,e),j) = e$$
$$\text{(A3)} \quad i \neq j \quad \Rightarrow \quad read(write(a,i,e),j) = read(a,j)$$

Variables in these axioms are assumed to be universally quantified. Axiom (A2) asserts that the value read at an index is the same as the last value written to this index. Axiom (A3) asserts that writing to an index, does not change the values at other indices. In principle, axioms (A2) and (A3) can be used as rewrite rules, which allow to replace every read following a write by a *conditional* expression:

$$read(write(a,i,e),j) \quad \text{is repl. by} \quad cond(i = j, e, read(a,j))$$

The expression $cond(c, s, t)$ returns s if the condition c is *true*, and t otherwise. The benefit of this *eager* approach is that writes are completely eliminated. The drawback is a quadratic blowup [15]. In addition to the array operations discussed so far, SMT solvers support *conditionals* on arrays, e.g. $cond(c, a, b)$, where a and b are arrays. In principle, using a fresh array variable d,

$$cond(c,a,b) \quad \text{can be repl. by} \quad (c \rightarrow a = d) \wedge (\bar{c} \rightarrow b = d),$$

but this requires two more array equalities, which might be expensive to handle.

Extensionality allows array comparisons [5, 4]:

$$\text{(A4)} \quad a = b \quad \Leftarrow \quad \forall i\, (read(a,i) = read(b,i))$$

The other direction of the implication is already covered by (A1). The extensionality axiom (A4) is required for reasoning about array *inequalities*:

$$\text{(A4}'\text{)} \quad a \neq b \quad \Rightarrow \quad \exists \lambda\, (read(a,\lambda) \neq read(b,\lambda))$$

3. ALGORITHM

We use lemmas on demand [12, 14, 1, 13, 10] for bit-vectors and one-dimensional[1] arrays, similar to [15], but for the extensional theory of arrays. We also describe the algorithm on a much more precise level.

Bit-vector variables and operations are eagerly encoded into SAT, including fresh variables for array equalities and read operations. The array operations, write and conditional, are not encoded, as they return arrays. The resulting SAT instance can produce spurious satisfying assignments, invalid in the extensional theory of arrays. Therefore, if the SAT solver returns a satisfying assignment σ, then we still have to check all array axioms, before concluding satisfiability. If an array axiom is violated, we generate a symbolic bit-vector lemma that rules out this and similar assignments. In our implementation this lemma is incrementally added on the CNF level. No additional expressions are generated.

We show that the lemmas on demand approach [12] can be generalized to the extensional theory of arrays. Adding bit-vector lemmas on demand is sufficient. This algorithm [12] always terminates, as there are only finitely many assignments to bit-vector variables in ϕ. In every iteration the algorithm can terminate concluding unsatisfiability, or satisfiability if there is no theory inconsistency. However, if an inconsistency is detected, this and similar inconsistent assignments are ruled out, and the algorithm continues with the next refinement iteration.

[1] Multi-dimensional arrays can be easily be represented by one-dimensional arrays.

```
procedure lemmas-on-demand (φ)
    encode-to-sat (φ)
    loop
        (r, σ) ← sat(φ)
        if (r = unsatisfiable) return unsatisfiable
        if (consistent (φ, σ)) return satisfiable
        add-lemma (φ, σ)
```

Figure 1: Algorithm.

4. CONSISTENCY CHECKER

Let ϕ denote the formula for which satisfiability is checked, and let σ denote a concrete assignment, generated by the SAT solver. We represent ϕ as an expression DAG. Expressions are either arrays or bit-vectors.

4.1 Read

If the only array operations are reads, the only array expressions in ϕ are array *variables*. The consistency checker iterates over all array variables and checks congruence. If $\sigma(i) = \sigma(j)$, but $\sigma(read(a,i)) \neq \sigma(read(a,j))$, e.g. (A1) is violated, then a read-read conflict occurs, and the symbolic bit-vector lemma is added:

$$i = j \quad \Rightarrow \quad read(a,i) = read(a,j)$$

All lemmas are encoded directly on the CNF level. No additional expressions are generated. Due to space constraints, encoding details have to be skipped.

4.2 Write

If we also consider write expressions in ϕ, then an array expression is either an array variable or a write operation. Note that ϕ can contain *nested* writes. If array axiom (A2) or (A3) is violated, a read-write conflict occurs. In the first phase, the consistency checker iterates over array expressions and propagates reads:

1. Map all a to its set of reads $\rho(a)$, initialized as $\rho(a) = \{read(a,j) \text{ in } \phi\}$.
2. For all $read(b,i) \in \rho(write(a,j,e))$: if $\sigma(i) \neq \sigma(j)$, add $read(b,i)$ to $\rho(a)$.
3. Repeat step 2 until fix-point is reached, i.e. ρ does not change anymore.

In the second phase consistency is checked:

4. For all $read(b,i)$, $read(c,k)$ in $\rho(a)$: check *adapted* congruence axiom.
5. For all $read(b,i) \in \rho(write(a,j,e))$, $\sigma(i) = \sigma(j)$: check $\sigma(read(b,i)) = \sigma(e)$.

In step 4 the original array congruence axiom (A1) can not be applied, as $\rho(a)$ can contain propagated reads, which do not read on a directly. An adapted axiom is violated if $\sigma(i) = \sigma(k)$, but $\sigma(read(b,i)) \neq \sigma(read(c,k))$. We collect all indices $j_1^i \ldots j_m^i$, which have been used as j in the update rule for ρ (step 2) while propagating $read(b,i)$. Similarly, we collect all indices $j_1^k \ldots j_n^k$ used as j in the update rule for ρ

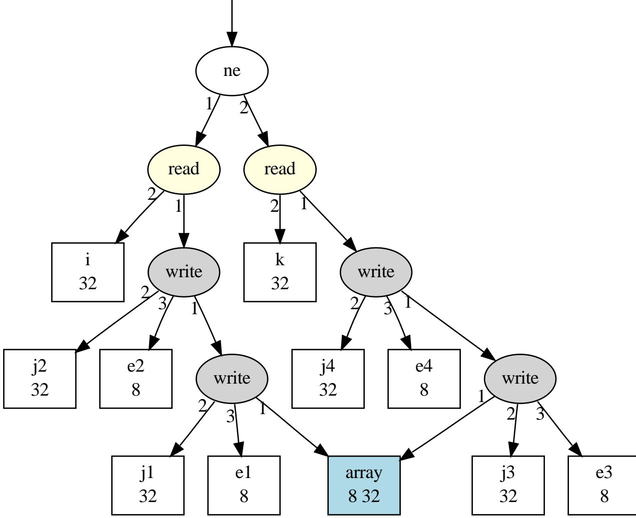

Figure 2: ϕ for the examples 1 and 2.

while propagating $read(c, k)$. We add the following symbolic bit-vector lemma:

$$i = k \;\wedge \bigwedge_{l=1}^{m} i \neq j_l^i \;\wedge \bigwedge_{l=1}^{n} k \neq j_l^k \quad \Rightarrow \quad read(b, i) = read(c, k)$$

The first big conjunction encodes that $\sigma(i)$ is different from all the assignments to the write indices on the propagation path of $read(b, i)$, the second conjunction that $\sigma(k)$ is different from all the write indices on the propagation path of $read(c, k)$. Adding this symbolic lemma makes sure that in the consistency checks of the following refinement iterations, the two reads can not produce the same conflict: either the propagation paths change, or the reads are congruent.

EXAMPLE 1. *Consider the reads*

$$r_1 := read(write(write(a, j_1, e_1), j_2, e_2), i)$$

and

$$r_2 := read(write(write(a, j_3, e_3), j_4, e_4), k),$$

as shown in Fig. 2. Let ϕ be $r_1 = r_2$. We assume that the SAT solver has generated the following assignments: $\sigma(i) = 0$, $\sigma(k) = 0$, $\sigma(j_1) = 1$, $\sigma(j_2) = 5$, $\sigma(j_3) = 1$, $\sigma(j_4) = 2$, $\sigma(r_1) = 3$ and $\sigma(r_2) = 4$, i.e. all the write indices are different from the two identical read indices, and the read values differ. Read r_1 is propagated down to a, as $\sigma(i) \neq \sigma(j_2)$ and $\sigma(i) \neq \sigma(j_1)$. Read r_2 is propagated down in the same way. We check $\rho(a)$, find an inconsistency according to step 4, as $\sigma(i) = \sigma(k)$, but $\sigma(r_1) \neq \sigma(r_2)$, and add the following lemma:

$$i = k \wedge i \neq j_1 \wedge i \neq j_2 \wedge k \neq j_3 \wedge k \neq j_4 \quad \Rightarrow \quad r_1 = r_2$$

If the check in step 5 fails, i.e. axiom (A2) is violated, then we generate a similar lemma. In this case one of the propagation paths is empty.

EXAMPLE 2. *Let ϕ be as in example 1, and $\sigma(i) = 3$, $\sigma(r_1) = 1$, $\sigma(j_2) = 0$, $\sigma(j_1) = 3$ and $\sigma(e_1) = 4$. We propagate r_1 down to $write(a, j_1, e_1)$, as $\sigma(i) \neq \sigma(j_2)$. We check $\rho(write(a, j_1, e_1))$, find an inconsistency according to step 5, as $\sigma(i) = \sigma(j_1)$, but $\sigma(r_1) \neq \sigma(e_1)$, and add the following lemma:*

$$i = j_1 \wedge i \neq j_2 \quad \Rightarrow \quad r_1 = e_1$$

Fix-point computation can be implemented as post-fix DFS traversal on the array expression sub graph of ϕ interleaved with on-the-fly consistency checking. Without array conditionals nor equality the sub graph is actually a tree, which simplifies this traversal further.

4.3 Conditional

As soon as we add conditionals on arrays, our array expressions become real DAGs. We add the following rule after step 2:

2*a*. For all $read(b, i) \in \rho(cond(c, t, e))$:
if $\sigma(c) = 1$, add $read(b, i)$ to $\rho(t)$,
else add $read(b, i)$ to $\rho(e)$.

The current assignment of the SAT solver determines which array is selected. If the condition is set to 1, reads are propagated down to t, otherwise down to e.

The lemma, generated upon inconsistency detection, is extended in the following way. We additionally collect all conditions $c_1^i, \ldots c_o^i$ used as c in step 2*a* while propagating down $read(b, i)$. Similarly, we collect all conditions $c_1^k, \ldots c_p^k$ used as c in step 2*a* while propagating down $read(c, k)$. Two more conjunctions are added:

$$\ldots \bigwedge_{l=1}^{o} c_l^i = \sigma(c_l^i) \;\wedge \bigwedge_{l=1}^{p} c_l^k = \sigma(c_l^k) \;\Rightarrow\; read(b, i) = read(c, k)$$

Note that on the CNF level, $c_k = \sigma(c_k)$ can be represented as one literal.

4.4 Equality

Finally, we also consider extensionality and introduce array equalities. For every array equality $a = b$ we generate a fresh boolean variable $e_{a,b}$, as in the Tseitin Transformation [21]. Two *virtual* reads, $read(a, \lambda)$ and $read(b, \lambda)$, over a fresh index variable λ, are added. Then we add as constraint:

$$\bar{e}_{a,b} \quad \Rightarrow \quad read(a, \lambda) \neq read(b, \lambda)$$

The idea of this constraint is that virtual reads are used as witness for array *inequality*. The SAT solver is only allowed to set $e_{a,b}$ to *false* if there is an assignment to λ, such that $read(a, \lambda) \neq read(b, \lambda)$. A similar usage of λ can be found in [4], and as k in rule *ext* [20]. To propagate reads over array equalities, we add rule 2*b*:

2*b*. For all $a = b$ where $\sigma(e_{a,b}) = 1$ and $read(c, i) \in \rho(a)$:
add $read(c, i)$ to $\rho(b)$ and vice versa.

According to Rule 2*b* reads have to be propagated over an array equality, but only if the SAT solver assigns it to *true*.

However, the changes made so far are not sufficient. Consider the formula:

$$write(a, i_1, e_1) = write(b, i_2, e_2) \;\wedge\; i_1 = i_2 \quad \Rightarrow \quad e_1 = e_2$$

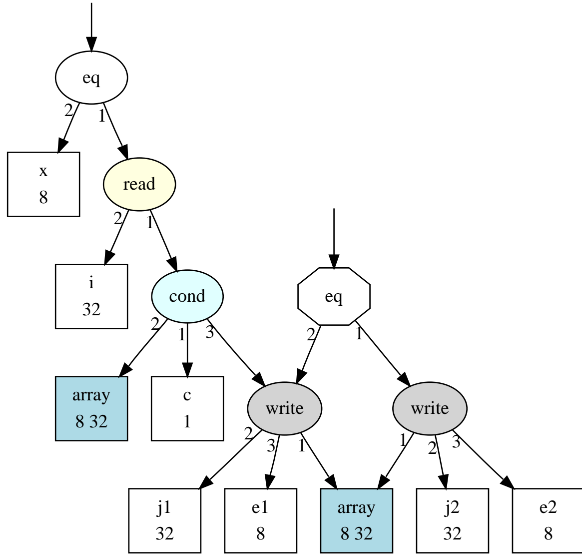

Figure 3: ϕ for the multi-rooted examples 3 and 4.

So far this formula can not be shown to hold, although it is a theorem. To solve this problem, we interpret writes as reads and propagate them in the same way. As a result of this, we enforce that the write values e_1 and e_2 have to be equal, as writes interpreted as reads have to be congruent according to axiom (A1). Note that write propagation is only necessary as we do not handle $\mathcal{EUF}$ explicitly, i.e. congruence of *write*.

We use a polymorphic expression *access* in our implementation. An access expression can be a read or write. This simplifies the implementation, as we do not have to distinguish between read or write during propagation.

As soon as array equalities are introduced, fix-point computation is needed, as propagations may be cyclic. A simple example is the transitivity of array equality. Recall that we introduce two virtual reads for every array equality, which in this example are propagated in a cyclic way. Hence, post-order traversal is no longer sufficient.

The lemma, generated upon inconsistency detection, is extended in the following way. We additionally collect all array equalities $e^i_1, \dots e^i_q$ used as e in step $2b$ while propagating $read(b, i)$. Similarly, we collect all array equalities $e^k_1, \dots e^k_r$ used as e in step $2b$ while propagating $read(c, k)$. Two more conjunctions are added:

$$\dots \wedge \bigwedge_{l=1}^{q} e^i_l \wedge \bigwedge_{l=1}^{r} e^k_l \quad \Rightarrow \quad read(b, i) = read(c, k)$$

EXAMPLE 3. *Now consider $w_1 := write(a, j_1, e_1)$, $r := read(cond(c, b, w_1), i)$ and $w_2 := write(a, j_2, e_2)$, as shown in Fig. 3. Let ϕ be $r = x \wedge w_1 = w_2$. We assume that the SAT solver has generated the following assignments: $\sigma(j_1) = 0$, $\sigma(j_2) = 0$, $\sigma(e_1) = 0$, $\sigma(e_2) = 3$ and $\sigma(e_{w_1,w_2}) = 1$. We propagate w_2 as read over the array equality $w_1 = w_2$ to w_1. We find an inconsistency for $\rho(w_1)$, according to step 4, as $\sigma(j_1) = \sigma(j_2)$, but $\sigma(e_1) \neq \sigma(e_2)$, and add the lemma:*

$$j_1 = j_2 \wedge e_{w_1,w_2} \quad \Rightarrow \quad e_1 = e_2$$

EXAMPLE 4. *Now let ϕ be as in example 3, and $\sigma(i) = 0$, $\sigma(r) = 1$, $\sigma(c) = 0$, $\sigma(j_1) = 1$, $\sigma(j_2) = 0$, $\sigma(e_2) = 5$ and $\sigma(e_{w_1,w_2}) = 1$. We propagate r down to a, as $\sigma(c) = 0$ and $\sigma(i) \neq \sigma(j_1)$. We propagate w_2 as read over e_{w_1,w_2} to w_1 and down to a, as $\sigma(e_{w_1,w_2}) = 1$ and $\sigma(j_1) \neq \sigma(j_2)$. We find an inconsistency for $\rho(a)$, according to step 4, as $\sigma(i) = \sigma(j_2)$, but $\sigma(r) \neq \sigma(e_2)$, and add the following lemma:*

$$i = j_2 \wedge i \neq j_1 \wedge j_2 \neq j_1 \wedge c = 0 \wedge e_{w_1,w_2} \quad \Rightarrow \quad r = e_2$$

The rules presented so far, do not propagate upwards, e.g. axiom (A3) is only applied from left to right. Consequently, there is one class of formulas, where the consistency checker fails to find inconsistencies. Consider the following example:

$$\begin{aligned} & i \neq k \ \wedge \ j \neq k \ \wedge \\ & write(a, i, e_1) = write(b, j, e_2) \ \wedge \\ & read(a, k) \neq read(b, k) \end{aligned}$$

This formula is unsatisfiable, which can be derived by applying axiom (A1) and (A3). The access objects are not propagated upwards, therefore the reads are never propagated to the same array expression. The inconsistency can not be detected by the set of rules discussed so far. To solve this problem, we finally complete our consistency checking algorithm as follows:

2c. For all $read(b, i) \in \rho(a)$:
if $\sigma(i) \neq \sigma(j)$, add $read(b, i)$ to $\rho(write(a, j, e))$.

2d. For all $read(b, i) \in \rho(t)$:
if $\sigma(c) = 1$, add $read(b, i)$ to $\rho(cond(c, t, e))$.

2e. For all $read(b, i) \in \rho(e)$:
if $\sigma(c) = 0$, add $read(b, i)$ to $\rho(cond(c, t, e))$.

4.5 Complexity, Soundness and Completeness

The complexity of the consistency checking algorithm is quadratic in the worst case. More specifically the number of expression nodes $|\phi|$ is an upper bound on the number of read expression nodes and also an upper bound on the number of array expression nodes. Therefore there are at most $O(|\phi|^2)$ updates to ρ in one run of the consistency checker.

Soundness follows from the fact, that all our rules respect axioms. Thus, if the consistency checker finds a conflict, then the current assignment does not satisfy the extensional theory of arrays. Our rules simply model all ways of applying axioms (A1) to (A4). Thus, if our consistency checker determines consistency, the model is also consistent in the extensional theory of arrays.

After the workshop, we have been working on a detailed complexity analysis, and proofs of soundness and completeness, which are contained in a full version of this paper [6].

5. EXPERIMENTAL EVALUATION

We implemented our lemmas on demand algorithm for the extensional theory of arrays in our new SMT solver Boolector including the previous state of the art decision procedure [20]. Our implementation of [20] eliminitates equalities on arrays by eager rewriting. Side conditions of rewrite rules are handled symbolically to avoid case splitting. We used Boolector version 0.0 for our experiments. Boolector can be downloaded from `http://fmv.jku.at/boolector`.

Boolector uses a functional AIG encoding with two level AIG rewriting [7], as in [8]. Picosat [3] is used as SAT solver.

We ran our benchmarks on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu. We set a time limit of 900 seconds and a memory limit of 1500 MB.

Our set of benchmarks, see Table 1, contains extensional examples, where computer memory is modelled as one big array of 2^{32} bytes. Benchmark `swapmem` swaps with a combination of XOR operations two byte sequences in memory twice. Extensionality is used to show that the final memory is equal to the initial. An instance is satisfiable if the sequences can overlap, and unsatisfiable otherwise. Benchmark `dubreva` reverses multiple byte sequences in memory twice. Again, extensionality is used to show that the final memory is equal to the initial. The instances are all unsatisfiable. In benchmarks `wchains` we check the following. Given P 32 bit pointers, the 32 bit word a pointer points to is overwritten with its pointer address. Then, the order in which the pointers are written does not matter as long the pointers are 4 byte aligned.

6. CONCLUSION

We showed that lemmas on demand can also handle the extensional theory of arrays. Our key contributions are a *precise* formulation of a propagation based approach and its *generalization* to extensionality. Our implementation shows very competitive performance. Future optimizations include a linear equation solver [15] and a faster rewriting engine.

After the workshop, we have been working on a full version [6] of this article which describes complexity, soundness and completeness in more detail.

Finally, we want to thank Nikolaj Bjørner and Leonardo De Moura for their helpful comments on an earlier version of this paper.

7. REFERENCES

[1] C. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proc. CAV*, 2002.

[2] C. Barrett and C. Tinelli. CVC3. In *Proc. CAV*, 2007.

[3] A. Biere. PicoSAT essentials. *JSAT*, 4, 2008.

[4] A. Bradley and Z. Manna. *The Calculus of Computation - Decision Procedures with Applications to Verification*. Springer, 2007.

[5] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In *Proc. VMCAI*, 2006.

[6] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. submitted.

[7] R. Brummayer and A. Biere. Local two-level and-inverter graph minimization without blowup. In *Proc. MEMICS*, 2006.

[8] R. Brummayer and A. Biere. C32SAT: Checking C expressions. In *Proc. CAV*, 2007.

[9] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In *Proc. CAV*, 2007.

[10] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. TACAS*, 2007.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACACS*, 2008.

[12] L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *Proc. SAT*, 2002.

[13] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. In *Proc. BMC*, 2003.

[14] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Proc. CAV*, 2003.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV*, 2007.

[16] P. Manolios, S. Srinivasan, and D. Vroon. BAT: The bit-level analysis tool. In *Proc. CAV*, 2007.

[17] J. McCarthy. Towards a mathematical science of computation. In *Proc. IFIP Congress*, 1962.

[18] R. Nieuwenhuis and A. Oliveras. Decision procedures for SAT, SAT modulo theories and beyond. the BarcelogicTools. In *Proc. LPAR*, 2005.

[19] R. Sebastiani. Lazy satisfiability modulo theories. *JSAT*, 3, 2007.

[20] A. Stump, C. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *Proc. LICS*, 2001.

[21] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, 1968.

Table 1: Benchmarking Extensional Examples

			lemmas			eager		z3		cvc3	
benchmark	S	P	R	T	M	T	M	T	M	T	M
swapmem	s	2	16	0.1	0.6	**0.0**	**0.0**	0.1	5.0	554.1	18.2
swapmem	s	6	53	**0.4**	**1.2**	0.6	3.3	547.6	101.6	90.0	71.9
swapmem	s	8	37	**0.3**	**1.4**	1.5	5.4	out of time	out of time	333.8	175.8
swapmem	s	10	34	**0.5**	**1.7**	1.0	8.3	out of time	out of time	740.2	305.3
swapmem	s	12	64	**1.0**	**2.2**	3.5	11.2	out of time	out of time	out of time	out of time
swapmem	s	14	57	**1.6**	**2.4**	1.8	15.7	out of time	out of time	out of time	out of time
wchains	s	6	14	**0.0**	**0.0**	0.8	7.0	46.3	11.6	8.5	60.1
wchains	s	8	16	**0.0**	**0.0**	1.9	14.5	572.2	32.5	23.5	117.6
wchains	s	10	18	**0.1**	**1.3**	3.1	20.3	out of time	out of time	34.2	193.6
wchains	s	12	20	**0.1**	**1.4**	3.5	29.8	out of time	out of time	65.4	294.6
wchains	s	15	21	**0.1**	**1.5**	4.2	45.8	out of time	out of time	126.3	472.6
wchains	s	20	28	**0.3**	**2.0**	5.7	80.2	out of time	out of time	267.8	897.3
wchains	s	30	36	**0.6**	**2.8**	10.9	179.9	out of time	out of time	out of memory	out of memory
wchains	s	60	70	**2.3**	**4.8**	41.8	718.5	out of time	out of time	out of memory	out of memory
wchains	s	90	96	**4.8**	**6.8**	out of memory	out of memory	out of time	out of time	out of memory	out of memory
dubreva	u	2	19	0.1	0.6	**0.0**	**0.0**	0.5	5.0	out of time	out of time
dubreva	u	3	41	0.3	**0.7**	**0.1**	1.2	0.7	5.6	out of time	out of time
dubreva	u	4	239	**12.5**	**2.2**	**12.5**	6.5	out of time	out of time	out of time	out of time
dubreva	u	5	379	**27.4**	**3.7**	348.3	13.1	out of time	out of time	out of time	out of time
dubreva	u	6	1187	**322.2**	**12.4**	out of time	out of time	out of time	out of time	out of time	out of time
dubreva	u	7	1637	**663.1**	**18.2**	out of time	out of time	out of time	out of time	out of memory	out of memory
swapmem	u	2	13	**0.1**	**0.7**	**0.1**	0.8	1.2	5.5	6.1	9.0
swapmem	u	4	25	**1.1**	**1.0**	3.0	2.0	5.2	7.1	159.6	49.8
swapmem	u	6	37	**3.2**	**1.2**	27.9	4.2	16.1	8.6	out of time	out of time
swapmem	u	8	49	**9.0**	**1.5**	129.0	8.3	20.7	10.1	out of time	out of time
swapmem	u	10	61	**18.3**	**2.0**	411.2	15.4	30.9	11.0	out of time	out of time
swapmem	u	12	73	**34.7**	**2.4**	out of time	out of time	62.1	13.3	out of time	out of time
swapmem	u	14	85	**63.4**	**2.8**	out of time	out of time	102.7	19.4	out of time	out of time
wchains	u	2	17	**0.0**	**0.0**	**0.0**	**0.0**	out of time	out of time	4.1	5.8
wchains	u	4	33	**0.1**	**0.9**	1.7	3.8	10.5	16.4	out of time	out of time
wchains	u	6	49	**0.4**	**1.1**	13.5	8.2	1.9	5.5	out of memory	out of memory
wchains	u	8	65	**0.6**	**1.3**	54.8	15.0	848.0	725.1	out of memory	out of memory
wchains	u	10	81	**1.1**	**1.5**	158.2	23.3	59.7	14.6	out of memory	out of memory
wchains	u	12	97	**2.4**	**1.8**	333.5	34.3	93.7	22.4	out of memory	out of memory
wchains	u	14	113	**2.9**	**1.9**	588.2	46.7	19.3	6.9	out of memory	out of memory
wchains	u	16	129	**4.8**	**2.1**	out of time	out of time	35.7	8.1	out of memory	out of memory
wchains	u	18	145	**6.6**	**2.3**	out of time	out of time	209.8	26.2	out of memory	out of memory

We compare our lemmas on demand approach with (i) the eager approach [20], implemented as rewrite system, (ii) Z3 1.2 [11], and (iii) CVC3 1.2.1 [2]. The first column gives the benchmark name, the second the satisfiability S, followed by the benchmark parameter P. In the fourth column, labelled R, the number of refinements (lemmas on demand) is shown. Columns T and M show solving time in seconds and memory usage in MB. Our approach clearly outperforms all the others.