A Flexible Schema for Generating Explanations in Lazy Theory Propagation

Roberto Bruttomesso* Edgar Pek† Natasha Sharygina*
*Università della Svizzera Italiana, Lugano, Switzerland
†University of Urbana-Champaign, Illinois, USA

Abstract—**Theory propagation in Satisfiability Modulo Theories is crucial for the solver's performance. It is important, however, to pay particular care to the amount of deductions to perform. The risk is in fact to clog the SAT-Solver with too many (and potentially useless clauses). In this paper we review some techniques for generating and communicating clauses to the SAT-Solver. In addition we propose a generic and flexible schema for theory propagation in which explanations for entailed facts are generated by re-using the consistency check procedure that is normally available in a theory solver. We argue that our schema can simplify the design of a theory solver, and allow a flexible form of theory propagation even for inherently hard theories (such as bit-vectors).**

I. Introduction

Satisfiability Modulo Theories (SMT) [4] can be seen as a generalization of Boolean satisfiability: besides Boolean variables SMT-Solvers also accept atomic constraints in some background decidable first-order theory $\mathcal{T}$.

A well-studied architecture for SMT-Solvers uses a DPLL SAT-Solver in cooperation with a decision procedure for a background theory $\mathcal{T}$ ($\mathcal{T}$-solver). In this schema the search is driven by the SAT-Solver, while the $\mathcal{T}$-solver is called on partial assignments to check the consistency in $\mathcal{T}$ of the set of enumerated constraints. If the set of constraints is found to be $\mathcal{T}$-inconsistent a conflict clause is produced by the $\mathcal{T}$-solver and it is added to the SAT-Solver's database. This approach is currently implemented in many SMT-Solvers such as BARCELOGIC [6], CLSAT, CVC3 [5], MATHSAT [9], OPENSMT [10], SATEEN [16], VERIT [7], YICES [14], and Z3 [12]. For a detailed introduction on SMT we refer the reader to [4], [23], [20].

Theory propagation consists of discovering and activating constraints that are a logical consequence in $\mathcal{T}$ ($\mathcal{T}$-consequences) of a partial assignment. Theory propagation can be used in combination with standard Boolean Constraint Propagation (BCP) to achieve a higher degree of deduction power. *Explanations*[1] for $\mathcal{T}$-consequences are clauses that encode logical implications in $\mathcal{T}$. For instance, if $x = y$ and $y = z$ belong to the partial assignment, then $x = z$ is a $\mathcal{T}$-consequence of the partial assignment. The explanation of the consequence is: $\neg(x = y) \vee \neg(y = z) \vee (x = z)$.

Explanations are *potentially* necessary during the conflict analysis in the SAT-Solver. However (as pointed out in [20]) only a small ratio of all explanations for $\mathcal{T}$-consequences is required during conflict analysis (for some benchmarks no explanation at all is needed). For efficiency it is important, therefore, to *generate and communicate* only those explanations that are strictly necessary.

[1]Also called reasons, justifications, or lemmata in the literature.

In this paper we propose a method for computing explanations on demand by reusing the consistency check algorithm for a generic theory $\mathcal{T}$. We argue that our approach simplifies the design of a $\mathcal{T}$-solver, as it does not have to provide the implementation of any additional procedure for generating explanations of $\mathcal{T}$-consequences.

Moreover, we show mechanisms to detect $\mathcal{T}$-consequences for the theory of equality with uninterpreted functions ($\mathcal{EUF}$) and the theory of bit-vectors ($\mathcal{BV}$). The mechanisms are based on simple modifications of the original consistency checking algorithms.

The paper is structured as follows. In §II we recall some preliminary notions and previous approaches to theory propagation. In §III we describe our approach. In §IV we show how to efficiently detect $\mathcal{T}$-consequences for an $\mathcal{EUF}$ and a $\mathcal{BV}$ theory solvers. In §V we discuss the quality of explanation for a $\mathcal{DL}$ solver w.r.t. different theory propagation methods. We conclude in §VI.

II. Preliminaries and Background

A. Notation

In the following we shall use the letters $\{a, b, \ldots, h\}$ (possibly with subscripts) to denote Boolean variables or the Boolean abstraction of a theory atom, l for a literal, $\{v, w, x, y, z\}$ to denote variables in a theory, F to denote an uninterpreted function, and $\{s, t\}$ for generic terms in a theory. η is a (partial) Boolean assignment, i.e., the set of literals (Boolean variables or negated variables) assigned to $true$ by the SAT-Solver at some point in time during the search. Sometimes we shall refer to η as to the *trail*.

B. Theory Propagation

In the context of SMT, theory propagation (TP) can be seen as a form of Boolean Constraint Propagation (BCP) driven by the background theory [2], [3], [13], [19], [8], [11], [20]. There is however a crucial difference between BCP and TP. In BCP any propagated literal l is associated with an explicit *explanation* clause in the database (i.e., the clause that caused the propagation); in theory propagation, instead, the explanation is *generated* by the $\mathcal{T}$-solver, and *communicated* to the SAT-Solver's clause database.

There are two main approaches to theory propagation in the SMT literature, with respect to the way explanations are generated and communicated.

Figure 1 depicts the approach as described and used in the MATHSAT solver [8]. The loop marked with the dashed line represents the interaction between BCP and TP (*BCP-TP loop* TP runs after BCP if the partial assignment η is consistent. The loop closes after TP eagerly adds (to the clause database) an explanation $\eta_l \subseteq \eta$ for each $\mathcal{T}$-consequence l.

Cotton and Maler [11] describe two variations of the BCP-TP loop. In the first variation, explanations for TPs are communicated all-at-once; in the second (which was shown to be faster), each TP run is interleaved with a BCP run. In both variations explanations are *eagerly generated communicated* inside the BCP-TP loop.

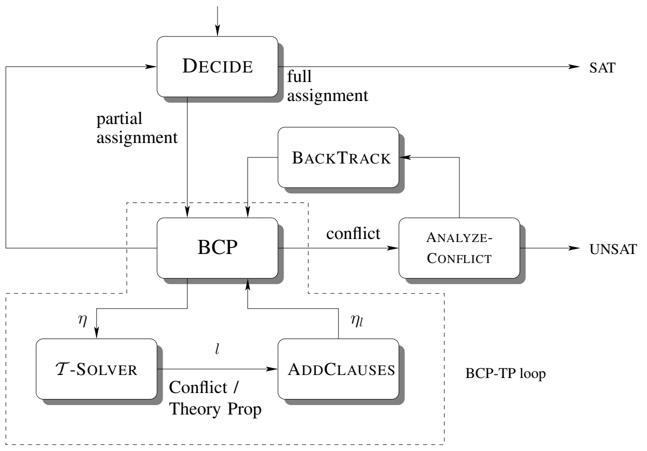

Fig. 1. Eager communication schema (figure adapted from [17]). The BCP-TP loop is marked with the dashed line. BCP runs first. Then the partial assignment η is given to the $\mathcal{T}$-solver. If η is $\mathcal{T}$-consistent a $\mathcal{T}$-consequence l may be derived, and the explanation η_l is eagerly added to the clause database. The loop is repeated until no more propagations are derived or a conflict is detected.

Nieuwenhuis and Oliveras [19] proposed the approach depicted in Figure 2 for the theory of difference logic. In this approach the BCP-TP loop does not eagerly add explanations to the database: explanations are generated and communicated during conflict analysis (*Expl. loop* in Figure 2). In other words, explanations are *lazily generated and communicated.*

This theory propagation schema was presented in [19] with a method of generating explanations restricted to the difference logic theory. [20] (§5) reports a more detailed presentation of eager vs. lazy strategies for theory propagation, and discusses the problem of avoiding the generation of "too new explanations". Basically, when generating an explanation, one has to make sure that all the literals in the explanation of a literal l have been assigned before l. The absence of "too new explanations" is a sufficient condition to guarantee the absence of loops in the implication graph of the SAT-Solver.

In this paper we present another technique for generating explanations lazily. Our approach is generic (i.e., it is not restricted to a particular theory) as it reuses the consistency check procedure available in the theory solver, and it is guaranteed not to generate "too new explanations". The technique is the core theory propagation strategy implemented in OPENSMT solver [10]. According to the results of the SMT-COMP'09 [1], despite not being as competitive as other industrial solvers, OPENSMT was the fastest open-source solver for the categories QF_UF, QF_IDL, QF_RDL, QF_LRA.

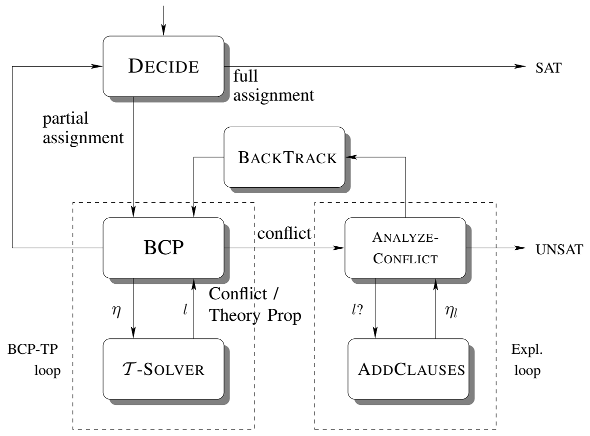

Fig. 2. Lazy communication schema was first used in [19] for the case of SMT($\mathcal{DL}$). In the BCP-TP loop a $\mathcal{T}$-consequence l is passed without an explanation. Explanations can be requested during conflict analysis. If an explanation for a $\mathcal{T}$-consequence is requested (l?) it is generated and communicated on demand.

C. Lazy explanations: a running example

Before we describe our method, we briefly recall the lazy theory propagation schema by means of a running example.

In this example we focus on the execution of an SMT-Solver for the theory of equality with uninterpreted functions and predicates ($\mathcal{EUF}$). We assume that the input formula contains the following set of theory atoms and Boolean atoms (each atom corresponds to Boolean abstraction (BA) in the SAT-Solver):

Atom	BA	Atom	BA
$y = z$	a	$v = w$	e
$F(x) = w$	b	$F(x) = v$	f
$x = y$	c	g	g
$x = z$	d	$v = F(z)$	h

Let's now consider a possible branch of the search. Suppose that at the current decision level, which we assume to be 2, the partial assignment in the SAT-Solver is $\{a, b\}$ ($\{y = z, F(x) = w\}$). Also suppose that the remaining unsatisfied clauses are

$$\begin{array}{l} \neg(x = z) \vee (v = w) \\ \neg(F(x) = v) \vee \neg g \\ g \vee \neg(x = z) \vee \neg(F(z) = v) \end{array}$$

At the current decision level no further propagation can be done (neither BCP nor TP). So, the SAT-Solver decides to increase the decision level to 3 by assuming c. The new assumption does not cause any BCP, but it triggers a TP. The propagated theory atom is $x = z$ because $\{y = z, F(x) = w, x = y\}$ is $\mathcal{EUF}$-consistent and $x = y \wedge y = z \rightarrow x = z$. Also, $x = y$ is added to the trail at the decision level 3.

The previous TP causes BCP to assign $v = w$ in the clause $\neg(x = z) \vee (v = w)$ ($x = z$ is added to the trail). The trail is still $\mathcal{EUF}$-consistent, so TP assigns $F(x) = v$ because $F(x) = w \wedge v = w \rightarrow F(x) = v$. Finally, $\neg g$ is Boolean propagated by the clause $\neg(F(x) = v) \vee \neg g$, and $\neg(v = F(z))$ is Boolean propagated by the clause $g \vee \neg(x = z) \vee \neg(v = F(z))$. Now, the call to the $\mathcal{EUF}$-solver returns $unsat$ because the set $\{x = z, F(x) = v, \neg(F(z) = v)\}$ is inconsistent.

Figure 3 shows the mixed Boolean-theory implication graph before the theory conflict is detected. The graph includes both BCPs (solid lines) and TPs (dashed lines). The following is the complete list of clauses involved in the described run:

1) $\neg(x = y) \vee \neg(y = z) \vee (x = z)$
2) $\neg(x = z) \vee (s = w)$
3) $\neg(F(x) = w) \vee \neg(v = w) \vee (F(x) = v)$
4) $\neg(F(x) = v) \vee \neg g$
5) $g \vee \neg(x = z) \vee \neg(v = F(z))$

where clauses 1 and 3 are the *explanations* for the two TPs.

The crucial point to observe is that the SAT-Solver behaved *as if* the explanation clauses for TPs (1 and 3) were already part of the original problem. The $\mathcal{T}$-solver can "substitute" the missing clauses (1 and 3) by just *detecting* the deduced literals that would have been propagated if 1 and 3 were part of the original problem.

D. Theory Propagation and Conflict Analysis

Explanations for propagations (both Boolean and theory) are needed by the SAT-Solver during conflict analysis to compute a conflict clause, and therefore to choose the right decision level for backtracking. In this paper we use the common conflict analysis technique [18], in the variant implemented in the MINISAT solver [15], that we briefly recall as follows.

Conflict analysis is performed with a number of resolution steps: the clause that causes the conflict (also called the *conflicting clause*) is iteratively resolved with the clauses that caused the literals involved in the conflict to be propagated. The resolution steps are driven by the trail: the pivot literal for each resolution step is the one in the conflicting clause that has been last assigned on the trail and not yet resolved. In other words, the resolution steps are performed in *reverse* chronological order w.r.t. trail assignments. Figure 4 shows the conflict analysis steps for our running example.

The resolution process begins with the conflicting clause $g \vee \neg d \vee \neg h$, generated by the $\mathcal{T}$-solver during the last consistency check. The pivots for resolution are chosen among the literals occurring in the candidate clause, and in reversed assignment order. The conflict analysis stops whenever the candidate clause contains exactly one literal implied by the latest assumption.

As observed in [18], not all the clauses used to perform a sequence of BCPs are needed during conflict analysis. Only the explanations of the literals appearing within the *implication cone* of the conflict will be used by the resolution-based analysis (as depicted in Figure 5).

This observation is also true for theory propagations. In other words, from the SAT-Solver point of view, the computation of the conflict clause can be performed *without knowing all the explanations for the TPs performed*: only the explanations for the atoms involved in the resolution steps have to be known.

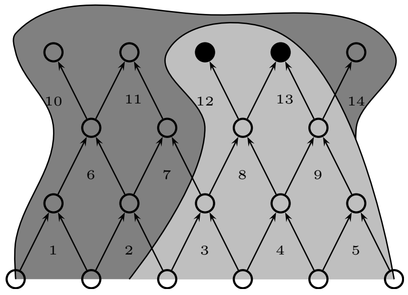

Fig. 5. Example of an implication cone for two conflicting literals (black-filled nodes); only clauses $\{5, 4, 3, 8, 9, 12, 13\}$ are used in conflict analysis.

To give a glimpse of the number of redundant explanations for TP, we ran experiments on a set of benchmarks from the SMT-LIB [22]. The results are reported in Table I. Data was computed using OPENSMT. This data seem to be consistent with the observations contained in [20].

Logic	Bench. Suite	TP done	E. need	% Ratio
QF_UF	SEQ	110.3 M	2.1 M	1.9
	PEQ	78.7 M	2.2 M	2.8
	NEQ	16.4 M	3.2 M	2.0
	loops6	23.0 M	2.3 M	9.9
	qg6	195.3 M	18.0 M	9.3
	qg7	180.1 M	1.7 M	9.3
QF_BV	tacas07	13.8 K	0	0
	brummayerbiere	109	0	0
	brummayerbiere2	93.0 K	57.0 K	61.1
QF_IDL	job_shop	60.2 M	1.6 M	2.7
	schedulingIDL	142.5 M	9.4 M	6.6
	parity	201.5 K	4.4 K	2.2
	mathsat	10.2 K	0.5 K	4.5
	qlock	27.4 M	0.5 M	1.8
	queens_bench	412.3 M	1.8 M	0.4

TABLE I
NUMBER OF TPS, NUMBER EXPLANATIONS NEEDED BY THE SAT-SOLVER DURING CONFLICT ANALYSIS, AND PERCENTAGE OF EXPLANATIONS NEEDED, DIVIDED BY CATEGORIES AND BENCHMARKS.

In Table I third column shows the sum of the number of TPs done, fourth column shows the number of explanations needed in conflict analysis, and the last column shows the ratio between the two values. For most benchmarks the number of explanations to be computed is a very small fraction of the total amount of TPs done (in some benchmarks no explanations at all is required).

III. LAZY EXPLANATIONS FOR $\mathcal{T}$-CONSEQUENCES BY CONSISTENCY CHECK

In our approach we follow the lazy theory propagation schema, i.e., we do not compute any explanation *a priori* in the $\mathcal{T}$-solver. The unassigned $\mathcal{T}$-atoms that are deduced by

Trail	
a	$y = z$
b	$F(x) = w$
c	$x = y$
d	$x = z$
e	$v = w$
f	$F(x) = v$
$\neg g$	g
$\neg h$	$\neg(v = F(z))$

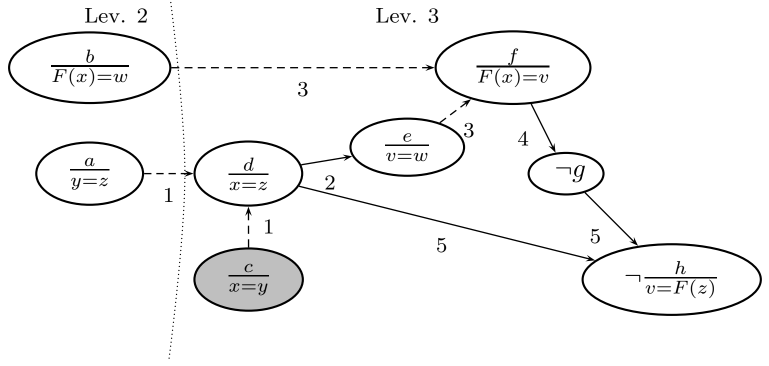

Fig. 3. Mixed Boolean-Theory Implication Graph for the running example of §II-C. Solid lines correspond to BCPs, while dashed lines correspond to TPs. The edges are labelled with the number of the clause that is responsible of the propagation. The gray-colored node is the latest assumption.

$$\dfrac{\neg d \lor e \qquad \dfrac{\neg b \lor \neg e \lor f \qquad \dfrac{\neg f \lor \neg g \qquad \dfrac{\neg d \lor \neg f \lor h \qquad g \lor \neg d \lor \neg h}{g \lor \neg d \lor \neg f}\,(h)}{\neg d \lor \neg f}\,(g)}{\neg b \lor \neg e \lor \neg d}\,(f)}{\neg b \lor \neg d}\,(e)$$

Fig. 4. Conflict Analysis for our running example. Each resolution step is labelled with the pivot variable.

the $\mathcal{T}$-solver are simply propagated to the SAT-Solver with a *dummy* explanation, a placeholder for a real explanation. The minimal requirements for the $\mathcal{T}$-solvers are shown in the interface of Figure 6a: $pushBckPoint$ sets a backtrack point in the $\mathcal{T}$-solver, in such a way that its current state can be restored later with a call to $popBckPoint$. $assertLit(l)$ communicates that the literal l has been assigned to $true$. $check$ is the consistency check procedure: if the set of asserted literals η is inconsistent it returns a (possibly minimal) explanation η_l, or an empty set if η is consistent.

```
class TSolver
    ...
    // Assert a literal
    bool assertLit( T-atom );

    // Set a backtrack point
    void pushBckPoint( );

    // Restore last backtrack point
    void popBckPoint( );

    // Consistency check
    set<T-atom > check( );

(a)
```

```
class SAT-Solver
    ...
    TSolver t;
    ...
    // l is the top-literal of the trail
    Clause getExplanation(l)
        t.popBckPoint();
        t.pushBckPoint();
        t.assertLit(¬l);
        η_l = t.check();
        t.popBckPoint();
        // Computes expl. clause
        Clause R = false
        for e ∈ η_l
            R = R ∨ ¬e
        return R

(b)
```

Fig. 6. (a): the $\mathcal{T}$-solver interface of OpenSMT. (b): the explicit code for $getExplanation$, for a generic $\mathcal{T}$-solver.

The main modification occurs inside the conflict analysis procedure of the SAT-Solver, and it is based on the following well-known observation[2].

Remark 1: Let η be a partial assignment such that $\eta \not\models_{\mathcal{T}} \bot$. Suppose that $\eta \models_{\mathcal{T}} l$. Therefore $\eta \cup \{\neg l\} \models_{\mathcal{T}} \bot$.

The modified conflict analysis procedure works as follows. When a conflict is found, either during BCP or a theory consistency check, we let the SAT-Solver proceed with the resolution steps of conflict analysis as usual; at the same time we backtrack the $\mathcal{T}$-solver of the $\mathcal{T}$-atoms already resolved. Whenever a dummy explanation for an atom l is about to be considered, we demand the $\mathcal{T}$-solver to return an explanation for its previous deduction with the procedure $getExplanation$ (see Figure 6b for the explicit code): we backtrack l from the $\mathcal{T}$-solver, and we temporarily push back $\neg l$. For Remark 1 the $\mathcal{T}$-solver is now in an inconsistent state. We can therefore ask the $\mathcal{T}$-solver for an explanation of the inconsistent state. Since $\eta \not\models_{\mathcal{T}} \bot$, but $\eta \cup \{\neg l\} \models_{\mathcal{T}} \bot$, the explanation $\eta_l \subseteq \eta \cup \{\neg l\}$ is guaranteed to contain l. The equivalent clausal form of η_l can be used as an explanation for the interrupted resolution step.

Notice that η considered at the moment of getting the explanation, might be different from the partial assignment that caused the $\mathcal{T}$-detection of l; this fact does not affect the soundness of the approach, as it is sufficient that the explanations for l always contain literals that have been assigned before l, in order to avoid *circular dependencies* in the implication graph [18] (i.e., we do not produce "too new explanations").

[2]Remark 1 is also used in [2], [13] to detect $\mathcal{T}$-consequences, and it is often called *plunging*. Plunging detects $\mathcal{T}$-consequences by means of consistency check calls on all unassigned literals. It is therefore computationally very expensive. In this respect our method an be seen as a lazy form of plunging.

The whole method is based on a double deception: the SAT-Solver behaves *as if* the explanations for TP were already part of the initial problem, while the $\mathcal{T}$-solver is used to compute explanations on demand *as if* it were performing a normal consistency check.

We argue that our variant of lazy theory propagation may greatly simplify the design of a $\mathcal{T}$-solver because it is only required to detect $\mathcal{T}$-consequences without providing an additional infrastructure for computing explanations. Moreover, in our schema $\mathcal{T}$-consequences detection does not have to be exhaustive. This is the key-point that enables the possibility of lazy communication for inherently hard theories such as $\mathcal{BV}$. Notice that detecting $\mathcal{T}$-consequences is generally an easy task compared to that of generating an explanation.

IV. Detecting $\mathcal{T}$-consequences for $\mathcal{EUF}$ and $\mathcal{BV}$

In this section we show how to detect $\mathcal{T}$-consequences for $\mathcal{EUF}$ and $\mathcal{BV}$ by modifying a congruence closure algorithm for $\mathcal{EUF}$ and a solver based on bit-blasting for $\mathcal{BV}$. To the best of our knowledge the following is the first complete description of a lazy theory propagation schema for $\mathcal{EUF}$ and $\mathcal{BV}$.

A. Equality and Uninterpreted Functions

We describe how $\mathcal{EUF}$-consequences can be efficiently detected in a congruence closure algorithm with a minor modification of the consistency check procedure.

Congruence closure is usually computed with an extension of the union-find algorithm. The procedure keeps track of the set of equivalence classes induced by the axioms of equality (reflexive, symmetric, and transitive) and by the congruence axiom ($x_1 = y_1, \ldots, x_n = y_n \rightarrow F(x_1, \ldots, x_n) = F(y_1, \ldots, y_n)$, for any functional symbol F). Each term t is initially associated with an individual equivalence class (t is the class *representant*).

Equivalence classes may merge for two reasons. First, classes merge when terms in different classes become equal. For example, let t, s, $F(t)$, and $F(s)$ be in four different classes. Then, equality $t = s$ merges classes of t and s into one. Second, classes merge because of the congruence axiom. In the example, $t = s$ implies $F(s) = F(t)$. Hence, classes of $F(s)$ and $F(t)$ merge into one.

We can consider the equality symbol $=$ as a normal functional symbol $=\!(\cdot,\cdot)$ with co-domain $\{true, false\}$. So, the congruence algorithm can treat equality as any other functional symbol. We achieve this by introducing the two special constants $true$ and $false$ (each constant represents an unmergable equivalence class). During the search, we merge $true$ with the positive $\mathcal{EUF}$-atoms (e.g., $t = s$) and $false$ with the negative $\mathcal{EUF}$-atoms (e.g., $t \neq s$).

When an $\mathcal{EUF}$-atom $t = s$ or $t \neq s$ is assigned, the congruence closure algorithm runs as usual. Additionally, we maintain two sets of unassigned $\mathcal{EUF}$-atoms that are merged by congruence with the equivalence classes of $true$ and $false$. The atoms in the set P are merged with the equivalence class of $true$, the atoms in the set N are merged with the equivalence class of $false$. The two sets represent $\mathcal{EUF}$-consequences to be propagated: atoms in P positively, atoms in N negatively. We can detect $\mathcal{EUF}$-consequences for other generic predicates in the same way.

Example 1: Consider again the running example of §II-C. We focus only on variables x, y, and z, and we assume that initially no $\mathcal{EUF}$-atom is assigned. Figure 7a shows the initial state of equivalence classes. When the atom $y = z$ is asserted to true, y and z are merged into the same class; $=\!(y, z)$ is merged with $true$ (Figure 7b). When the atom $x = z$ is asserted to true, x is merged with the class of z, and $=\!(x, z)$ is merged with $true$. As a consequence of the congruence axiom and the fact that x, y, and z are equivalent, $=\!(x, y)$ is merged with the class of $=\!(y, z)$, $=\!(x, z)$, and $true$ (Figure 7c). Since $=\!(x, y)$ is unassigned and merged with $true$, we have $P = \{ =\!(x, y) \}$, i.e., $x = y$ is an $\mathcal{EUF}$-consequence of the current assignment.

To evaluate our theory propagation mechanism for $\mathcal{EUF}$ we ran a set of experiments using OpenSMT on the most challenging benchmarks from the SMT-LIB for $\mathcal{EUF}$. Table II shows the results with theory propagation disabled (NOTP) and enabled (TP). Tests were run with a timeout of 600 s on an Intel Xeon 3.4 GHz. Results show that with our theory propagation schema we solve more benchmarks or solve them in less time.

Bench. Suite	# of timeouts		accumulated time (s)	
	NOTP	TP	NOTP	TP
`SEQ`	5	**4**	2243	1331
`PEQ`	16	**14**	1588	1849
`NEQ`	10	**6**	2827	2587
`loops6`	0	0	257	**159**
`gq6`	0	0	6913	**4567**
`gq7`	0	0	4901	**3205**

TABLE II
Number of timeouts and accumulated time for theory propagation disabled (NOTP) and enabled (TP). The best performance is highlighted with the bold-faced font.

B. Bit-Vectors

The $\mathcal{BV}$-solver of OpenSMT is based on an incremental reduction to SAT that can also be efficiently backtracked to a previous state. The main engine is based on another instance of the SAT-Solver used as a global enumerator, that we shall call *bit-blaster* from now on, to avoid confusion.

Each bit-vector term $t_{[n]}$ can be represented as an array of Boolean formulæ $[\mathtt{t_n}, \ldots, \mathtt{t_1}]$ by means of a well-known technique, that we briefly recall as follows. Each bit-vector variable $x_{[n]}$ is represented by n Boolean variables $[\mathtt{x_n}, \ldots, \mathtt{x_1}]$ inside the bit-blaster, while bit-vector operators are encoded into arrays of Boolean formulæ that encode the semantic of the operators: for instance $t_{[2]} + s_{[2]}$ is encoded as $[\mathtt{t_2} \oplus \mathtt{s_2} \oplus (\mathtt{t_1} \wedge \mathtt{s_1}), \ \mathtt{t_1} \oplus \mathtt{s_1}]$, where $[\mathtt{t_2}, \mathtt{t_1}]$ and $[\mathtt{s_2}, \mathtt{s_1}]$ are, recursively, the encodings for $t_{[2]}$ and $s_{[2]}$ respectively. Predicates, such as equalities, are instead encoded into a

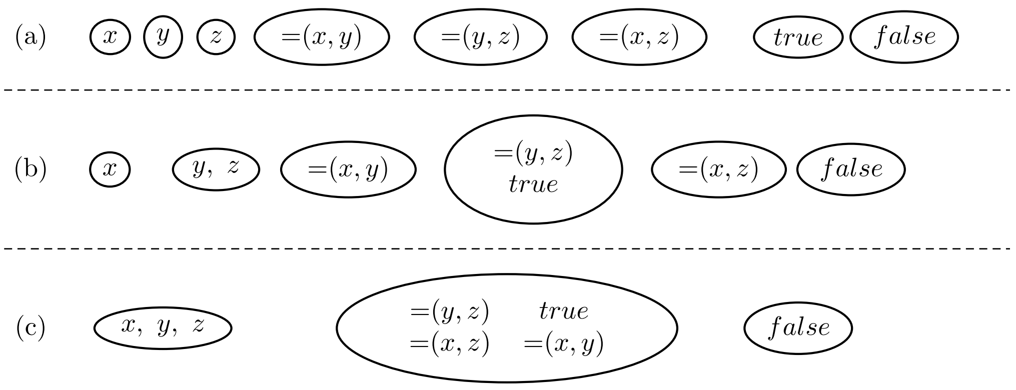

Fig. 7. Congruence classes for Example 1.

single Boolean formula: for instance $t_{[2]} = s_{[2]}$ results into $[(\mathtt{t}_2 \leftrightarrow \mathtt{s}_2) \wedge (\mathtt{t}_1 \leftrightarrow \mathtt{s}_1)]$.

In our approach we collect, at the initialization phase, all the $\mathcal{BV}$-atoms that might be assigned during the search process, and we bit-blast them upfront. Each $\mathcal{BV}$-atom a, with encoding $[\mathtt{a}]$, is associated with an *activation variable* $\mathtt{v_a}$ and given to the bit-blaster as $\mathtt{v_a} \leftrightarrow \mathtt{a}$. The activation variable controls the activation of the encoding of an entire $\mathcal{BV}$-atom inside the bit-blaster.

Whenever, during the search, an $assertLit(l)$ is called, and $var(l) = a$, we retrieve the activation variable $\mathtt{v_a}$ associated with a: $\mathtt{v_a}$ is then assigned inside the bit-blaster with the same polarity as l. The bit-blaster can be then queried for satisfiability. In case of a satisfiable call, the detection of $\mathcal{BV}$-consequences can be performed in a straightforward manner: it is sufficient to track the sets P and N of $\mathcal{BV}$-atoms corresponding to those activation variables that have been forced (i.e., propagated at decision-level 0) to $true$ or $false$ by the bit-blaster during the satisfiability check. As for $\mathcal{EUF}$, P and N are the sets of $\mathcal{BV}$-atoms that can be propagated to $true$ and $false$ respectively.

In practice we observed that it is beneficial to call the $check$ method of the $\mathcal{BV}$-solver only when a complete Boolean assignment is enumerated, given the complexity of $check$ itself. On partial models (in particular, on each call to $assertLit$), instead, only Boolean constraint propagation is performed inside the bit-blaster: $\mathcal{BV}$-consequences can be still detected as a result of the mere propagation process (this is the method we use for computing statistics of Table I). Our bit-blaster supports efficient backtracking. Theory conflicts can be detected by tracking the activation variables involved in the conflict analysis procedure of the bit-blaster.

To demonstrate the effectiveness of our theory propagation mechanism for $\mathcal{BV}$ we ran a set of experiments using OpenSMT on a set of benchmarks from the SMT-LIB (these benchmarks have a non-trivial Boolean structure compared to others), with theory propagation disabled (NOTP) and enabled (TP). Results are shown in Table III. Tests were run with a timeout of 600 s on an Intel Xeon 3.4 GHz. Using our theory propagation schema for $\mathcal{BV}$ results in a better performance, either in the number of timeout or in the accumulated solving time.

Bench. Suite	# of time/mem-outs		accumulated time (s)	
	NOTP	TP	NOTP	TP
`tacas07`	0	0	71	**60**
`brummayerbiere`	7	**6**	274	715
`brummayerbiere2`	35	**34**	1398	309

TABLE III
Number of timeouts and accumulated time for theory propagation disabled (NOTP) and enabled (TP). The best performance is highlighted with the bold-faced font.

V. Quality of explanations

The discussion in [20] (§5.1) about the advantages of using lazy instead of eager explanations mostly focuses on the aspect that certain explanations are not required during conflict analysis. Adding the explanation to the SAT-Solver may result in an useless burden for the solver's data structures. Here we suggest an additional reason why one should prefer lazy w.r.t. eager explanations: the quality of the generated explanations.

We have implemented both the eager generation schema (E) and the lazy generation schema (L) by consistency check (described in §III) in our difference logic ($\mathcal{DL}$) solver. The solver is based on the algorithm described in [11]: the consistency check algorithm reduces to an incremental negative cycle detection procedure. Eager theory propagation is performed via two single source shortest path (SSSP) computations (as described in [11]). Table IV shows a comparison of the size of the explanations obtained with the eager and lazy methods.

The reason why this happens is outlined as follows. Let A be the current set of asserted literals, and suppose that a literal l_a, not in A, is now asserted in the $\mathcal{DL}$ solver. l_a might cause another unassigned literal l_d to be deduced. However, since BCP may run after l_d is deduced, a number of literals $l_{a_1}, \ldots, l_{a_n}$ may be asserted in the $\mathcal{DL}$ solver *before* l_d is asserted. Consider now the way E and L generate explanations for l_d. In E the explanation is generated eagerly based on the

Bench. Suite	avg expl. size		max expl. size		min expl. size	
	E	L	E	L	E	L
`job_shop`	14.7	**14.1**	32.3	**19.2**	3.0	**2.9**
`schedIDL`	15.6	**14.2**	35.6	**20.1**	2.9	**2.7**
`parity`	**15.0**	15.1	26.0	**25.8**	7.1	7.1
`mathsat`	**6.5**	6.6	12.5	**12.4**	2.6	2.6
`qlock`	8.1	**7.3**	29.7	**25.8**	2.2	**2.1**
`queens_bench`	7.8	**6.7**	25.7	**18.2**	2.3	2.3
Total	13.1	**12.1**	30.6	**20.9**	3.5	**3.4**

TABLE IV
AVERAGE VALUES OF AVERAGE EXPLANATION SIZE, MAX EXPLANATION SIZE, AND MIN EXPLANATION SIZE.

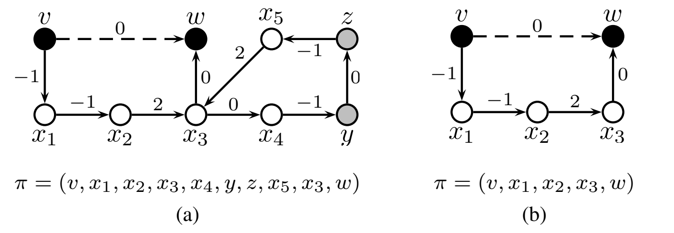

Fig. 8. Explanation path (π) for the deduced edge $v \xrightarrow{0} w$: (a) generated with approach E with respect to the edge $y \xrightarrow{0} z$ that caused the deduced edge; (b) generated with approach L.

information $A \cup \{l_a\}$. In L instead the explanation is generated lazily based on the information $A \cup \{l_a, l_{a_1}, \ldots, l_{a_n}\}$. Notice in fact that the explanation for l_d to be used in conflict analysis is just required to contain literals that have been pushed before l_d. This richer set of information generally leads to producing shorter explanations for L.

An additional reason why L creates shorter explanations than E lies in the different *way* they generate explanations. We explain the reason in the context of the $\mathcal{DL}$ theory solver. Note that the $\mathcal{DL}$ theory constraints can be interpreted as a graph: a theory literal $x - y \leq c$ corresponds to an edge $x \xrightarrow{c} y$ in the graph.

Figure 8 shows a comparison of explanations generated with E and L. In E the explanation for a deduced edge ($v \xrightarrow{0} w$) is computed eagerly when the edge causing the deduction ($y \xrightarrow{0} z$) is asserted, by traversing two shortest path trees. The two shortest path trees are created from the asserted edge endpoints (y, z) as the source vertices. Consequently, the shortest path for $v \xrightarrow{0} w$ is necessarily a shortest path *local* to ($y \xrightarrow{0} x$) as in Figure 8a. In general this is not the *global* shortest path from w to v.

In L, instead, the explanation is computed lazily using the consistency check procedure for $\mathcal{DL}$. The path explaining the deduction is, therefore, the *global* shortest path w to v, and it does not necessarily include $y \xrightarrow{0} z$ (see example in Figure 8b).

VI. CONCLUSION

In this paper we have studied a method for generating explanations on demand in the context of lazy theory propagation for Satisfiability Modulo Theories. Our approach is generic (i.e., not restricted to a particular theory) because it only requires a detection mechanism for $\mathcal{T}$-consequences and consistency check procedure to generate explanations.

This may simplify the design of a theory solver as it has only to detect $\mathcal{T}$-consequences, without explicitly supporting a procedure for generating explanations. Moreover we have shown how to effectively perform the detection of $\mathcal{T}$-consequences in the context of $\mathcal{EUF}$ and $\mathcal{BV}$ theories.

As a further reason for preferring lazy over eager theory propagation, we have shown, by means of experiments, that the explanations generated on demand are usually shorter than the ones produced eagerly with a state-of-the-art $\mathcal{DL}$ solver. The explanation generation schema described is available in the open-source SMT-Solver OPENSMT [21].

REFERENCES

[1] SMT-COMP'09. http://www.smtcomp.org.
[2] A. Armando, C. Castellini, and E. Giunchiglia. SAT-Based Decision Procedures for Temporal Reasoning. In *ECP*, pages 97–108, 2000.
[3] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. of CADE'02*, 2002.
[4] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Handbook on Satisfiability*, volume 185, chapter Satisfiability Modulo Theories. IO Press, 2009.
[5] C. Barrett and C. Tinelli. CVC3. In *CAV'07*, 2007.
[6] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez Carbonell, and A. Rubio. The Barcelogic SMT Solver. In A. Gupta and S. Malik, editors, *CAV'08*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer, 2008.
[7] T. Bouton, D. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *CADE*, 2009.
[8] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Van Rossum, S. Schulz, and R. Sebastiani. An Incremental and Layered Procedure for Satisfiability of Linear Arithmetic Logic. In *TACAS*, 2005.
[9] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*, pages 299–303, 2008.
[10] R. Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT Solver. In *TACAS*, 2010.
[11] S. Cotton and O. Maler. Fast and Flexible Difference Constraint Propagation for DPLL(T). In *SAT'06*, pages 170–183, 2006.
[12] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, pages 337–340, 2008.
[13] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.
[14] B. Dutertre and L. de Moura. The Yices SMT Solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf.
[15] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *Theory and Applications of Satisfiability Testing (SAT2003)*, pages 502–518, 2003.
[16] H. Kim and F. Somenzi. Finite Instantiations for Integer Difference Logic. In *FMCAD*, pages 31–38, 2006.
[17] D. Kroening and O. Strichman. *Decision procedures an algorithmic point of view*. Theoretical computer science. Springer-Verlag, May 2008.
[18] J. Marques-silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
[19] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In *CAV'05*, pages 321–334, 2005.
[20] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
[21] OPENSMT Web Page. http://verify.inf.usi.ch/opensmt.
[22] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.smtlib.org, 2006.
[23] R. Sebastiani. Lazy Satisfiability Modulo Theories. *JSAT*, 3:144–224, 2007.