Verifying Heap-Manipulating Programs in an SMT Framework *

Zvonimir Rakamarić[2], Roberto Bruttomesso[1], Alan J. Hu[2], and Alessandro Cimatti[1]

[1] ITC-IRST, Povo, Trento, Italy
{bruttomesso,cimatti}@itc.it
[2] Department of Computer Science, University of British Columbia, Canada
{zrakamar,ajh}@cs.ubc.ca

Abstract. Automated software verification has made great progress recently, and a key enabler of this progress has been the advances in efficient, automated decision procedures suitable for verification (Boolean satisfiability solvers and satisfiability-modulo-theories (SMT) solvers). Verifying general software, however, requires reasoning about unbounded, linked, heap-allocated data structures, which in turn motivates the need for a logical theory for such structures that includes unbounded reachability. So far, none of the available SMT solvers supports such a theory. In this paper, we present our integration of a decision procedure that supports unbounded heap reachability into an available SMT solver. Using the extended SMT solver, we can efficiently verify examples of heap-manipulating programs that we could not verify before.

1 Introduction

Automated software verification has made great progress recently, with several successful tools developed in both industry and academia. A key enabling technology for this success has been the advances in automated decision procedures — the software verification tools almost all rely on some form of automatic logical reasoning engine. Some rely on SAT (Boolean satisfiability) or BDDs (binary decision diagrams) to maintain bit-accurate precision (e.g., [16, 22, 2]), whereas others use SMT solvers (satisfiability modulo theories — decision procedures for combinations of decidable theories) in order to capitalize on the natural abstractions present in software verification, such as integer and real linear arithmetic, arrays, and uninterpreted functions (e.g., [4, 20, 18, 5]).

To be broadly applicable, however, software verification tools must be able to verify programs with dynamic memory allocation, i.e., that manipulate potentially unbounded, heap-allocated, linked data structures via pointers. Although verification of such *heap-manipulating programs* (HMPs) is obviously undecidable in general, careful crafting can produce a logic that is expressive enough to verify important properties of programs, yet is still decidable. In particular, a crucial feature for such logics is the ability to specify unbounded reachability (e.g., from node x, is it possible to reach node y by

* Supported by (1) a research grant from the Natural Sciences and Engineering Research Council of Canada, (2) a University of British Columbia Graduate Fellowship, (3) ORCHID, a project sponsored by Provincia Autonoma di Trento, and (4) a research grant from Intel.

following pointers) and related concepts such as betweenness. Slightly more expressive logics, however, are undecidable [21].

Logics for HMP verification have long been a topic of research. Even Nelson's seminal work on software verification with SMT solvers supported a theory of unbounded S-expressions, although without reachability [35, 37], and soon thereafter, Nelson proposed a first-order axiomatization that approximated unbounded reachability [36]. The past few years, however, have seen a blossoming of research in this area, with numerous proposed logics and decision procedures for HMPs, with varying degrees of expressiveness and efficiency, e.g., [3, 6, 9, 15, 21, 24, 27–29, 33, 34, 39–41]. Research progress has been great, with verification examples that were beyond the reach of methods just a few years ago now being verified in seconds. However, the research on HMP verification has focused almost exclusively on the heap-verification aspects, while mainstream software verification research has largely ignored HMP verification — an understandable division, given the difficulty of both problems.

With the logics and decision procedures for HMPs maturing, the time is right to integrate them back into a general SMT solver, to enable verification of more general software. We want to verify software, including software that manipulates heaps, not just software that *only* manipulates heaps! A few researchers have started in this direction. For example, Lahiri and Qadeer have expressed an incomplete axiomatization of unbounded reachability as universally quantified axioms in the Simplify first-order prover [17], allowing verification of heap and non-heap properties and their interactions, but with a substantial performance penalty [27]. Beyer et al. [7] take a different approach, making calls to a specialized HMP verification system (the TVLA system [30]) to handle the heap aspects of the verification from within their non-heap-aware software verification tool. They report excellent performance, but such a loose combination doesn't allow verification of general interactions between heap properties and other program properties. In very recent follow-on work [8], they add a "strengthening" operator to propagate additional information between the heap and non-heap theories, but still not all interactions are captured. Similarly, Charlton and Huth [14] propose a software model checker in which separate analysis plugins (such as for heaps and for other theories) can cooperate, but the communication is ad hoc, so there are no guarantees that all interactions between theories are propagated. Closest to our work is extremely recent work by Lahiri and Qadeer [28]: Instead of their previous first-order axiomatization, they present a decision procedure based on a complete set of rewrite rules, inspired by our previous work [9]. However, they prototype an implementation of the rewrite rules by using the same trick of modeling rewrite rules as universally-quantified first-order axioms inside the theorem prover, as before. Practical implementation of their decision procedure into an SMT solver has not yet been done. The obviously promising next step is a tight integration of an efficient decision procedure for an HMP logic directly into a modern SMT solver, making all of the theories, and their interactions, efficiently available for the verification task. So far, however, nobody has actually done such an integration.

In this paper, we present the theory, methodology, and results of such an integration. In particular, we integrate our recent, efficient decision procedure for an HMP logic that supports unbounded reachability [39] into the established SMT solver MATH-

```
 1: procedure INIT-ADD-FLAG($head, val$)
 2:     assume $\mathsf{reach}(next, head, t) \wedge \mathsf{reach}(next, head, \mathsf{nil}) \wedge \neg t = \mathsf{nil} \wedge oldSum =$
            $\mathsf{data\_int}(sum, t) \wedge oldFlag = \mathsf{data\_bool}(flag, t)$
 3:     $curr := head$;
 4:     while $\neg curr = \mathsf{nil}$ do
 5:         if $\neg(curr \rightarrow flag)$ then
 6:             $curr \rightarrow sum := curr \rightarrow sum + val$;
 7:             $curr \rightarrow flag := \mathsf{true}$;
 8:         end if
 9:         $curr := curr \rightarrow next$;
10:     end while
11:     assert $\mathsf{reach}(next, head, t) \wedge \mathsf{reach}(next, head, \mathsf{nil}) \wedge \neg t = \mathsf{nil} \wedge \mathsf{data\_bool}(flag, t) \wedge$
            $(oldFlag \vee \mathsf{data\_int}(sum, t) = oldSum + val)$
12: end procedure
```

Fig. 1. HMP (Heap-Manipulating Program) Example. The procedure INIT-ADD-FLAG adds the integer variable *val* to integer field *sum* of every node whose boolean field $flag$ is false in an acyclic singly-linked list. Also, boolean field $flag$ of those nodes is set to true. We denote an integer data field named *sum* of a node *x* by $\mathsf{data\_int}(sum, x)$, a boolean data field named $flag$ of a node *x* by $\mathsf{data\_bool}(flag, x)$, and the node pointed to by a pointer field named *next* of node *x* by $\mathsf{next}(next, x)$. Subformulas of the form $\mathsf{reach}(next, x, y)$ express that node y is reachable from node x by following a sequence of any number of *next* pointer fields. We will formally define these predicates in Sect. 3. The fact that nil is reachable from *head* enforces the acyclicity assumption. Variables $oldSum$ and $oldFlag$ are used to store values of fields *sum* and $flag$ of node *t* before the procedure starts, respectively. In the **assume** and **assert** statements, variable *t* represents an arbitrary node (Skolem constant). Since our framework doesn't support quantification, we use the trick of introducing Skolem constants to represent universally quantified variables.

SAT [12].[3] Our results indicate that the integration was fairly straightforward (as was hypothesized in [39] and thanks to the design of MATHSAT [10, 11]), the performance overhead of the integration was reasonable, and the integration enabled verification of many example HMPs that we could not verify before.

2 Motivating HMP Example

In our framework, the *heap* consists of an unbounded number of heap *nodes*. HMPs can have program variables that are pointer variables (pointers) and data variables of different types. Similarly, heap nodes can have any number of pointer fields (i.e. links to other nodes) and data fields of different types.

We'll motivate the work presented in this paper with an illustrative HMP example given in Fig. 1. The procedure INIT-ADD-FLAG adds the value of the integer variable *val* to integer field *sum* of every node whose boolean field $flag$ is false in the non-empty acyclic singly-linked input list *head*. Furthermore, boolean field $flag$ of those nodes is set to true. Necessary assumptions are formalized by the **assume** statement on line 2 of the program. The body of the procedure is simple; it traverses the list, finds

[3] The extended MATHSAT is available at `http://mathsat.itc.it/`.

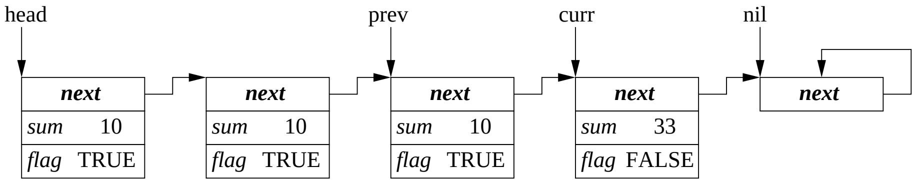

Fig. 2. Heap Structure Example. In this example, each list node has a pointer field *next*, an integer data field *sum*, and a boolean data field *flag*. We model nil as just a node where $\mathsf{next}(f, \mathsf{nil}) = \mathsf{nil}$ for all pointer fields f.

nodes whose field $flag$ is false, and on line 6 adds *val* to the data field *sum* at each iteration. Also, it assigns field $flag$ to true on line 7. The specification is expressed by the **assert** statement on line 11, and indicates that whenever line 11 is reached, *head* points to an acyclic singly-linked list with field *sum* of all nodes whose $flag$ field was false incremented by val. The verification problem we are solving can be stated as follows: given an HMP, determine whether it is the case that all executions that satisfy all **assume** statements also satisfy all **assert** statements. Note that even this simple example is beyond the capability of typical software model-checking tools: it is infinite-state due to both the unbounded integers as well as the unbounded heap. To verify such programs, we employ abstraction, using an SMT framework extended with a suitable logical theory described in the next section.

3 Logic for Verifying Heap-Manipulating Programs

Before we define our logic, we'll intuitively illustrate basic concepts on the example of a heap structure shown in Fig. 2. In this heap structure, *head*, *prev*, *curr*, and *nil* are pointer variables, *next* is a pointer field used to link nodes in the acyclic list, *sum* is an integer data field, and *flag* is a boolean data field. The node to which we get by following the *next* pointer field from the node pointed to by *head* is denoted in our syntax with $\mathsf{next}(next, head)$. The data field *flag* of the node pointed to by *prev* is accessed with $\mathsf{data\_bool}(flag, prev)$. The node pointed to by *curr* is reachable from the node pointed to by *head* by following *next* pointer fields, and that concept of unbounded reachability in our syntax is written as $\mathsf{reach}(next, head, curr)$.

The syntax of our logic is presented in Fig. 3. It is a quantifier-free fragment of first-order logic that contains two equational theories:

1. Theory of data fields with the signature $\{=, \mathsf{data}, \mathsf{update\_dfield}\}$. The theory of data fields can be easily translated into the theory of uninterpreted functions as described in Sect. 4.3. For the simplicity of presentation, in this section we give a single untyped theory of data fields. However, without the loss of generality, we can extend this to a family of theories of data fields whose signatures are parameterized using the respective data types. Currently, we support only boolean and integer data fields with the signatures $\{=, \mathsf{data\_bool}, \mathsf{update\_dfield\_bool}\}$ and $\{=, \mathsf{data\_int}, \mathsf{update\_dfield\_int}\}$, but that can easily be extended to other data types supported by the SMT solver (e.g. reals).

$$
\begin{array}{rcll}
c & \in & \mathit{Constants} & \\
x & \in & \mathit{DataVariables} & v \in \mathit{PointerVariables} \\
d, d' & \in & \mathit{DataFields} & f, f' \in \mathit{PointerFields} \\
\mathit{NodeTerm} & ::= & v \mid \mathsf{next}(f, \mathit{NodeTerm}) & \\
\mathit{DataTerm} & ::= & c \mid x \mid \mathsf{data}(d, \mathit{NodeTerm}) & \\
\mathit{Atom} & ::= & \mathit{NodeTerm} = \mathit{NodeTerm} \mid \mathit{DataTerm} = \mathit{DataTerm} \mid & \\
 & & \mathsf{reach}(f, \mathit{NodeTerm}, \mathit{NodeTerm}) \mid & \\
 & & \mathsf{between}(f, \mathit{NodeTerm}, \mathit{NodeTerm}, \mathit{NodeTerm}) & \\
\mathit{Literal} & ::= & \mathit{Atom} \mid \neg \mathit{Atom} \mid & \\
 & & \mathsf{update\_pfield}(f, \mathit{NodeTerm}, \mathit{NodeTerm}, f') \mid & \\
 & & \mathsf{update\_dfield}(d, \mathit{NodeTerm}, \mathit{DataTerm}, d') & \\
\mathit{Formula} & ::= & \mathit{Literal} \mid \mathit{Formula} \wedge \mathit{Formula} \mid \mathit{Formula} \vee \mathit{Formula} &
\end{array}
$$

Fig. 3. Syntax of the Logic. For brevity, we show the logic with untyped data fields.

2. Theory of unbounded reachability, which is defined below, with the signature $\{=$, next, reach, $\mathsf{between}$, $\mathsf{update\_pfield}\}$.

Clearly, the signatures (other than equality) of these two theories are disjoint, and are also disjoint from the signatures of the various theories MATHSAT currently supports, such as difference logic, linear arithmetic over reals, and linear arithmetic over integers.

3.1 Theory of Unbounded Reachability

The theory of unbounded reachability over heap nodes presented here is essentially the same as in [39], except that reasoning about data fields is now moved into the theory of data fields and handled by the SMT solver (see Sect. 4.3). The theory assumes a finite set of pointer variables *PointerVariables*, which model program variables that point to nodes in the heap, and a finite set of *pointer function* symbols *PointerFields*, which model pointer fields from a heap node to another heap node. Literals of the form $x{=}y$, $\neg x{=}y$, $\mathsf{reach}(f,x,y)$, and $\neg\mathsf{reach}(f,x,y)$ (where x and y are *NodeTerm*) are called *equality*, *disequality*, *reachability*, and *unreachability* literals, respectively. Literals of the form $\mathsf{between}(f,x,y,z)$ or its negation are called *between* literals.

The structures over which the semantics of the theory are defined are called *heap structures*. Formally, a heap structure $H = (N, \Theta)$ consists of a set of *nodes* N and an interpretation function Θ. The interpretation function Θ interprets each symbol σ in $\mathit{PointerVariables} \cup \mathit{PointerFields}$, so that:

- Each pointer variable symbol $\sigma \in \mathit{PointerVariables}$ is interpreted as a node $\Theta(\sigma) \in N$.
- Each pointer function symbol $\sigma \in \mathit{PointerFields}$ is interpreted as a mapping from nodes to nodes $\Theta(\sigma) \in N \rightarrow N$.

The interpretation function Θ extends to interpret any term, atom, or literal of the theory in a straightforward, inductive way. The interpretation of a node term $\tau \in$ *PointerVariables* is defined above, otherwise, τ has the form $\mathsf{next}(f, \tau')$ for some node term τ', and the interpretation is $\Theta(\tau) = \Theta(f)(\Theta(\tau'))$. Atoms are interpreted by Θ as boolean values:

- An equality atom $\tau_1 = \tau_2$ is interpreted as true iff $\Theta(\tau_1) = \Theta(\tau_2)$.
- A reachability atom $\mathsf{reach}(f, \tau_1, \tau_2)$ is interpreted as true iff there exists some $n \geq 0$ such that $\Theta(f)^n(\Theta(\tau_1)) = \Theta(\tau_2)$.[4]
- A between atom $\mathsf{between}(f, \tau_1, \tau_2, \tau_3)$ is interpreted as true iff there exist $n_0, m_0 \geq 0$ such that $\Theta(\tau_2) = \Theta(f)^{n_0}(\Theta(\tau_1))$, $\Theta(\tau_3) = \Theta(f)^{m_0}(\Theta(\tau_1))$, $n_0 \leq m_0$, and for all n, m such that $\Theta(\tau_2) = \Theta(f)^n(\Theta(\tau_1))$, $\Theta(\tau_3) = \Theta(f)^m(\Theta(\tau_1))$, we have $n_0 \leq n$ and $m_0 \leq m$.

The interpretation of a pointer field update literal $\mathsf{update\_pfield}(f, \tau_1, \tau_2, f')$ is defined using the well-known update operator[5] as true iff

$$\Theta(f') \;=\; \mathsf{update}(\Theta(f), \Theta(\tau_1), \Theta(\tau_2)).$$

Finally, the interpretation of a literal that is of the form $\neg\phi$ where ϕ is an atom is simply defined as $\Theta(\neg\phi) = \neg\Theta(\phi)$.

In previous work [9, 39], we described a saturation-based decision procedure for the theory of unbounded reachability. The decision procedure is based on the exhaustive application of a set of inference rules and, as we showed on a number of experiments, is very efficient. Furthermore, we presented some theoretical results behind our logic and decision procedure [38]: our decision procedure is sound and always terminates, and the decision procedure is complete for the fragment of the logic without updates. The experiments showed that in practice completeness was not an issue, as we could verify all examples that we could specify.

3.2 Example

Returning to our example from Fig. 2, we'll illustrate the semantics of our logic extended with the boolean and integer data field types on this heap structure with the interpretation of a few representative literals:

- $\mathsf{reach}(next, head, curr)$ is interpreted as true because the node pointed to by *curr* is reachable from the node pointed to by *head* following *next* pointer fields.
- $\mathsf{reach}(next, head, \mathsf{nil})$ is interpreted as true because the node nil is reachable from the node pointed to by *head* following *next* pointer fields. The fact that nil is reachable from *head* enforces the acyclicity assumption.
- $\mathsf{next}(next, curr) = \mathsf{nil}$ is true because the node to which we get by following one *next* pointer field from *curr* is nil.
- $\mathsf{data\_bool}(flag, prev) \leftrightarrow \mathsf{true}$ is interpreted as true because the boolean field *flag* of the node pointed to by *prev* is set to true.
- $\mathsf{data\_int}(sum, prev) = 10$ is interpreted as true because the integer field *sum* of the node pointed to by *prev* is set to 10.
- $\mathsf{between}(next, head, prev, curr)$ is true because node *prev* is between *head* and *curr*.
- $\mathsf{between}(next, head, \mathsf{nil}, curr)$ is interpreted as false because node nil is not between nodes *head* and *curr*.

[4] Here, function exponentiation represents iterative application: for a function g and an element x in its domain, $g^0(x) = x$, and $g^n(x) = g(g^{n-1}(x))$ for all $n \geq 1$.

[5] If g is a function, a is an element in g's domain, and b is an element in g's codomain, then $\mathsf{update}(g, a, b)$ is defined to be the function $\lambda x.(\text{if } x = a \text{ then } b \text{ else } g(x))$.

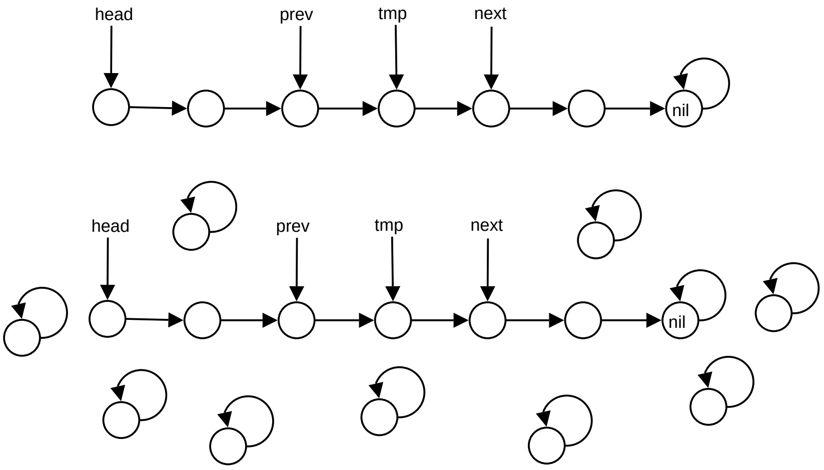

Fig. 4. An example of a heap structure H (top), and a constructed infinite heap structure H' (bottom) which satisfies every quantifier-free formula Ψ that is satisfied by H.

4 Theory Integration into MathSAT

In this section, we briefly recall some recent results concerning theory combination in SMT, and we disclose some details about the integration of the theory of unbounded reachability into MathSAT.

4.1 Efficient and Flexible Nelson-Oppen in SMT

Many verification tasks require the specification of properties at a level of expressiveness that is better captured by a logic that is the result of the combination (or union) of simpler theories T_1 and T_2, defined over signatures Σ_1 and Σ_2, respectively. In many situations, decision procedures $\mathtt{Dec(}T_i\mathtt{)}$ for T_i, $i = 1,2$, are already available to be used.

Nelson and Oppen [37] showed that given two equational theories T_1 and T_2, it is possible to derive a procedure $\mathtt{Dec(}T_1 \cup T_2\mathtt{)}$ for deciding quantifier-free formulae over $T_1 \cup T_2$, provided that:

- T_1 and T_2 are signature-disjoint (i.e. $\Sigma_1 \cap \Sigma_2 = \emptyset$);
- T_1 and T_2 are *stably infinite*[6].

A theory is stably infinite if for every satisfiable quantifier-free formula ϕ, there exists an interpretation satisfying ϕ whose domain is infinite. Many theories of interest are stably infinite, including the theory of integers and the theory of unbounded reachability from Sect. 3.1:

Theorem 1. *The theory of unbounded reachability (Sect. 3.1) is stably infinite.*

[6] This restriction has been relaxed in the recent work by Krstić et al. [25].

Proof. Let Ψ be a satisfiable quantifier-free formula, and let $H = (N, \Theta)$ be a heap structure satisfying Ψ. We'll show that one can always construct an infinite heap structure $H' = (N', \Theta')$ satisfying Ψ. Fig. 4 gives an example of how this is done. Basically, adding to the heap structure H an infinite number of nodes that point to themselves (and not changing the existing nodes) creates an infinite heap structure H' satisfying Ψ.

The heap structure H' is formally defined as follows. First, we fix an infinite set of nodes N_{Inf} disjoint from N. Then, we define $N' = N \cup N_{Inf}$, and interpretation Θ' as follows:
Interpretation function Θ' interprets each symbol $\sigma \in \mathit{PointerVariables}$ so that

$$\Theta'(\sigma) = \Theta(\sigma)$$

Every pointer function symbol $f \in \mathit{PointerFields}$ is interpreted so that

$$f^{\Theta'}(\tau) = \begin{cases} f^{\Theta}(\tau) & \text{if } \tau \in N \\ \tau & \text{otherwise} \end{cases}$$

Since H is a heap structure satisfying Ψ, the formula Ψ cannot syntactically include any of the nodes in N_{Inf}. Furthermore, for each type of atom, the additional nodes in N_{Inf} cannot change the truth values of those atoms in Ψ, since the new nodes are disconnected from the existing structure, which is unchanged. Therefore, H' also satisfies Ψ, and its domain is infinite. □

The Nelson-Oppen combination schema can be summarized as follows (for a more accurate survey the reader is referred to [32]). The input quantifier-free formula ϕ on $T_1 \cup T_2$ is initially *purified* into an equisatisfiable formula $\phi_1 \wedge \phi_2$ such that ϕ_i belongs to T_i, for $i = 1,2$. This can be easily achieved with the introduction of a set of fresh variables. The procedure is then based on an exhaustive communication between `Dec(`T_1`)` and `Dec(`T_2`)` by means of *interface* equalities, i.e. equalities between variables in $\mathit{vars}(\phi_1) \cap \mathit{vars}(\phi_2)$. Roughly speaking, the exchanging of interface equalities is sufficient for `Dec(`T_1`)` and `Dec(`T_2`)` to achieve an *agreement* on a common model, if such a model exists. This communication has to be implemented around `Dec(`T_1`)` and `Dec(`T_2`)` in order to obtain a correct `Dec(`$T_1 \cup T_2$`)`.

The Nelson-Oppen method is not limited to only two theories. In fact, if T_1 and T_2 are stably infinite, their union $T_1 \cup T_2$ is stably infinite as well. If we are given a decidable stably infinite T_3 over Σ_3 and $(\Sigma_1 \cup \Sigma_2) \cap \Sigma_3 = \emptyset$, than we can apply Nelson-Oppen and obtain a `Dec(`$T_1 \cup T_2 \cup T_3$`)`.

The introduction of a combination framework into an SMT schema can be naively done by considering `Dec(`$T_1 \cup T_2$`)` as a single *theory-solver*, by straightforwardly adapting a DPLL-like `Bool+Dec(`T`)` schema into a `Bool+Dec(`$T_1 \cup T_2$`)` setting.

Delayed Theory Combination (DTC) [10, 11] is an alternative approach specifically studied for SMT solvers, based on the observation that it is possible to lift to the boolean level the communication of interface equalities between the theory-solvers, by exploiting the boolean engine on top of them. The new framework, `Bool+Dec(`T_1`)+Dec(`T_2`)`, can be easily achieved as follows.

Given a purified formula $\phi_1 \wedge \phi_2$, the atom set $E = \{x_1 = x_2 \mid x_1, x_2 \in \mathit{vars}(\phi_1) \cap \mathit{vars}(\phi_2)\}$ is first generated. E is nothing but the set of interface equalities that the two theory-solvers, $\texttt{Dec(}T_1\texttt{)}$ and $\texttt{Dec(}T_2\texttt{)}$, *might* need to exchange at any point in time. Any set of theory-atoms Γ assigned to a truth value by the SAT-solver during the search is divided into $\Gamma_1' = \Gamma_1 \cup \Gamma_E$ and $\Gamma_2' = \Gamma_2 \cup \Gamma_E$, where Γ_i are atoms belonging to T_i, for $i = 1,2$, while Γ_E is a set of atoms in E. The set Γ_i' is fed to the corresponding solver $\texttt{Dec(}T_i\texttt{)}$ to be checked for consistency.

Intuitively, the communication in $\texttt{Dec(}T_1 \cup T_2\texttt{)}$, required for the correctness of the Nelson-Oppen procedure, is now emulated by the introduction of interface equalities that are shared by the two theories. In spite of the (potentially) quadratic number of new atoms generated in E, it is easily possible to control the model enumeration in the SAT-solver, as shown in [13], in order to avoid an enlargement of the search space.

The implementation of a $\texttt{Bool+Dec(}T_1\texttt{)+Dec(}T_2\texttt{)}$ schema presents several advantages with respect to a standard $\texttt{Bool+Dec(}T_1 \cup T_2\texttt{)}$:

- There is no need to build a Nelson-Oppen "box" $\texttt{Dec(}T_1 \cup T_2\texttt{)}$ around $\texttt{Dec(}T_1\texttt{)}$ and $\texttt{Dec(}T_2\texttt{)}$, because the integration is implicitly handled at the boolean level and not at the solver level.
- Mixed-conflict generation is automatic.
- Disjunction in case of non-convex theories is automatically handled at the boolean level, while in Nelson-Oppen it must be handled inside $\texttt{Dec(}T_1 \cup T_2\texttt{)}$. This results in a better efficiency, because of the mechanisms of backjumping and learning implemented in state-of-the-art SAT-solvers.
- The theory-solvers do not need deduction capabilities. In contrast, this is a requirement in Nelson-Oppen. This feature greatly simplified the integration, since our pre-existing decision procedure for the heap logic did not implement deduction.

4.2 Handling Uninterpreted Functions via Ackermann's Expansion

Ackermann's expansion [1] is a technique by means of which it is possible to translate a quantifier-free formula over $T \cup EUF$ into an equisatisfiable formula ϕ' over T only, where EUF is the well-known theory of Uninterpreted Functions with Equality.

Since function symbols are uninterpreted, the only requirement for satisfiability is *functional consistency*, i.e. the implication $(\bigwedge_{i=1}^n t_i = s_i) \rightarrow f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)$ must hold for every function symbol f of arity n, where t_i and s_i are terms.

In Ackermann's expansion, in order to fulfill the above condition, every distinct function application $f(t_1,\ldots,t_n)$ in ϕ is replaced with a fresh variable $v_{f(t_1,\ldots,t_n)}$. For each function symbol f of arity n, the obtained formula is then augmented with a set of axioms of the kind $(\bigwedge_{i=1}^n t_i = s_i) \rightarrow v_{f(t_1,\ldots,t_n)} = v_{f(s_1,\ldots,s_n)}$, for every pair of distinct fresh variables. It is easy to prove that the resulting formula ϕ' no longer contains any UF symbol and it is equisatisfiable to the original ϕ.

The same transformation can be used to remove uninterpreted predicate symbols, using fresh boolean variables and the logical connective $\leftrightarrow$ to equate them in the axiom instantiations.

4.3 Theory Integration

We have integrated the unbounded reachability decision procedure from Sect. 3.1 as a theory-solver $\texttt{Dec(}HMP\texttt{)}$ into MATHSAT, resulting in a framework for the verification of HMPs supporting boolean and integer data fields, but potentially also any other data type already handled by MATHSAT.

The rationale behind our combination is to separate the "heap reachability" part of the formula from the reasoning about "data", in order to achieve a modular SMT($HMP \cup T$) decision procedure, where T is the theory for a generic data type. In particular, in the current implementation, we provide in the input language a binary predicate $\mathsf{data\_bool}(d,h)$, and a binary function $\mathsf{data\_int}(d,h)$ that can be used to select a boolean or an integer stored in $d \in DataField$ of $h \in NodeTerm$. Notice that both constructs are uninterpreted, and they merely represent a modular solution to bridge the data and the heap part.

For boolean data, we can exploit the SAT-solver in MATHSAT to decide subformulae expressed on boolean data, by the Ackermann's expansion of the $\mathsf{data\_bool}(.,.)$ predicate. The interaction between the integer solver $\mathtt{Dec(}LIA\mathtt{)}$ (or in general, the non-boolean) reasoning and $\mathtt{Dec(}HMP\mathtt{)}$ can be dealt with in two different ways, either using a $\mathtt{Bool+Dec(}HMP\mathtt{)+Dec(}LIA\mathtt{)+Dec(}EUF\mathtt{)}$ schema, or a $\mathtt{Bool+Dec(}HMP\mathtt{)}$ $\mathtt{+Dec(}LIA\mathtt{)}$ schema, after the Ackermannization of $\mathsf{data\_int}(.,.)$ symbols.

Update operations on data $\mathsf{update\_dfield}(d,t,v,d')$ may be eagerly replaced with a set of axioms $\{d'(t) \approx v\} \cup \{s \neq t \rightarrow d'(s) \approx d(s) \mid s \in NT\}$, where $\approx$ is the equality $=$ for integer data and $\leftrightarrow$ for boolean data, and NT is the set of *NodeTerms* that appear in the formula. This solution is far from being optimal, but it worked well in practice for our experiments, where only a few updates were required.

$\mathtt{Dec(}HMP\mathtt{)}$, as any other theory-solver, also benefits of the EUF-layer of MATHSAT. Our experiments show that in many cases this layer is sufficient to determine the unsatisfiability of a query.

Example 1. We are given the following quantifier-free unsatisfiable SMT($HMP \cup LIA$) formula ϕ:

$$(\mathsf{data\_int}(d,h_1) + \mathsf{data\_int}(d,h_2) = 1) \wedge (h_1 = h_2)$$

Using Delayed Theory Combination: We first purify ϕ into ϕ' with the introduction of two new fresh variables v_1 and v_2, obtaining ϕ':

$$(v_1 = \mathsf{data\_int}(d,h_1)) \wedge (v_2 = \mathsf{data\_int}(d,h_2)) \wedge (v_1 + v_2 = 1) \wedge (h_1 = h_2).$$

The interface equality $v_1 = v_2$ is also generated. The atoms are assigned to the theories as follows:

$$\begin{array}{ll} HMP & \{h_1 = h_2\} \\ LIA & \{v_1 + v_2 = 1, v_1 = v_2\} \\ EUF & \{v_1 = \mathsf{data\_int}(d,h_1), v_2 = \mathsf{data\_int}(d,h_2), h_1 = h_2, v_1 = v_2\}. \end{array}$$

The SAT-solver assigns every atom in ϕ' to true. The contradiction is derived because $\mathtt{Dec(}LIA\mathtt{)}$ immediately implies $v_1 \neq v_2$, which falsifies the functional consistency in $\mathtt{Dec(}EUF\mathtt{)}$.

Using Ackermann's Expansion: The original formula is expanded into ϕ':

$$(h_1 = h_2) \wedge (v_1 + v_2 = 1) \wedge (h_1 = h_2 \rightarrow v_1 = v_2).$$

Again, $\mathtt{Dec(}LIA\mathtt{)}$ implies $v_1 \neq v_2$ that contradicts $h_1 = h_2 \wedge (h_1 = h_2 \rightarrow v_1 = v_2)$.

program	property	preds	DP calls	old time (s)	new time (s)
List-Reverse	NL	8	184	0.2	0.2
List-Add	NL∧AC∧IN	8	66	0.1	0.1
ND-Insert	NL∧AC∧IN	13	259	0.5	0.6
ND-Remove	NL∧AC∧RE	12	386	0.9	1.2
Zip [23]	NL∧AC	22	9153	17.3	27.3
Sorted-Zip	NL∧AC∧SO∧IN	22	14251	22.8	46.2
Sorted-Insert [27]	NL∧AC∧SO∧IN	20	5990	13.8	25.3
Bubble-Sort [3]	NL∧AC	18	3444	11.1	16.5
Bubble-Sort [3]	NL∧AC∧SO	24	31446	114.9	209.0
Remove-Elements	NL∧CY∧RE	17	3124	8.8	14.9
Remove-Segment [31]	CY	15	944	2.2	10.0
Search-And-Set	NL∧CY∧DT	16	4892	5.3	10.8
Set-Union [36]	NL∧CY∧DT∧IN	21	374	1.4	2.2
Create-Insert	NL∧AC∧IN	24	3020	14.8	15.6
Create-Insert-Data	NL∧AC∧IN	27	8710	39.7	47.3
Create-Free	NL∧AC∧IN∧RE	31	52079	457.4	489.2
Init-List	NL∧AC∧DT	9	81	0.1	0.1
Init-List-Var	NL∧AC∧DT	11	244	0.2	0.4
Init-Cyclic	NL∧CY∧DT	11	200	0.2	0.4
Sorted-Insert-DNodes	NL∧AC∧SO∧IN	25	7918	77.9	108.1
Remove-Doubly	NL∧DL∧RE	34	3238	24.3	33.0
Remove-Cyclic-Doubly [27]	NL∧CD∧RE	27	1695	15.6	15.7
Linux-List-Add	NL∧CD∧IN	25	1240	6.4	8.9
Linux-List-Add-Tail	NL∧CD∧IN	27	1638	7.3	10.0
Linux-List-Del	NL∧CD∧RE	29	2057	24.7	25.2

Table 1. Performance Comparison Against Previous Work [39]. The column "property" specifies the verified property; "preds" is the number of predicates required for verification; "DP calls" is the number of decision procedure queries; "old time" is the total execution time from [39]; "new time" is the total execution time using MathSAT. Our technical report [38] provides pseudocode and lists the required predicates for these examples. Some of the examples have been taken from related work, while the last three are from Linux kernel list container.

5 Experimental Results

We ran MathSAT extended with the unbounded reachability theory on a number of HMP verification queries. The queries are from a simple predicate abstraction [19]-based model checker that we are using to verify HMPs. This tool is a straightforward implementation of the software model checking algorithm with predicate abstraction [4], and is described in previous work [9, 39]. The experiments were executed on a 2.6 GHz Pentium 4 machine.

The first question is how much overhead the greater complexity of an integrated SMT solver imposes. Table 1 gives a performance comparison with the previous results from [39], using the standalone decision procedure for the unbounded reachability logic. The examples have either no data fields or only boolean data fields, so the previous work could handle them. The safety properties we checked (when applicable) of the HMPs are:

- *no leaks* (NL) – all nodes reachable from the head of the list at the beginning of the program are also reachable at the end of the program.
- *insertion* (IN) – a distinguished node that is to be inserted into a list is actually reachable from the head of the list, i.e. the insertion “worked”.
- *acyclic* (AC) – the final list is acyclic, i.e. nil is reachable from the head of the list.
- *cyclic* (CY) – list is a cyclic singly-linked list, i.e. the head of the list is reachable from its successor.
- *doubly-linked* (DL) – the final list is a doubly-linked list.
- *cyclic doubly-linked* (CD) – the final list is a cyclic doubly-linked list.
- *sorted* (SO) – list is a sorted linked list, i.e. each node’s data field is less than or equal to its successor’s.
- *data* (DT) – data fields of selected (possibly all) nodes in a list are set to a value.
- *remove elements* (RE) – for examples that remove node(s), this states that the node(s) was (were) actually removed.

The comparison shows that the integration isn’t a serious overhead. Although MATHSAT, with the integrated unbounded reachability theory, is a more heavyweight tool than the pure unbounded reachability decision procedure we were using previously, the performance penalty is reasonable.

The next question is whether the integration allows effectively verifying example HMPs that could not be handled previously, such as the example in Fig. 1 from Sect. 2.

Without the integration into an SMT solver, we handled integer data fields by bit-blasting them into a fixed number of boolean data fields that represented integers of a certain bit width. We used 1-bit integers in most examples (except for SEARCH-AND-SET where we used 2-bit integers) because the number of states (and therefore the number of decision procedure queries) grows exponentially with integer bit width. Furthermore, for HMP examples that use addition and multiplication, we would also have had to implement n-bit integer addition and multiplication, which would add even more complexity to the verification problem. We didn’t even attempt to verify such examples in our previous work.

With the integration into MATHSAT, a rich set of other theories is available to the verifier. Table 2 shows performance using MATHSAT on the HMP examples that contain (unbounded) integer data fields. In the verification of these examples, we are using a combination of multiple theories, including unbounded reachability, uninterpreted functions, and linear arithmetic. Some examples are the same as before, but with integers expanded from 1 or 2 bits to true integers. There is some slow-down for verification with unbounded integers, but the runtimes are quite comparable to the corresponding versions in Table 1. Several additional examples use arithmetic operators on the unbounded integers and have no analogue in Table 1. Overall, we see that we can efficiently verify many examples using the combined theories.

6 Conclusions and Future Work

The paper describes integration of the unbounded reachability theory described in our previous work into MATHSAT, a general purpose SMT solver. Integrating the theory into MATHSAT — easily accomplished through its theory combination framework —

program	property	preds	DP calls	time (s)
Sorted-Zip	NL∧AC∧SO∧IN	22	5758	53.9
Sorted-Insert	NL∧AC∧SO∧IN	20	2972	40.4
Bubble-Sort	NL∧AC	17	2348	16.9
Bubble-Sort	NL∧AC∧SO	23	17427	371.3
Remove-Elements	NL∧CY∧RE	17	3124	16.4
Remove-Segment	CY	15	944	10.3
Search-And-Set	NL∧CY∧DT	16	5120	13.7
Set-Union	NL∧CY∧DT∧IN	22	766	5.8
Create-Insert-Data	NL∧AC∧IN	27	8710	53.6
Init-List	NL∧AC∧DT	9	81	0.1
Init-List-Var	NL∧AC∧DT	11	244	0.4
Init-Cyclic	NL∧CY∧DT	11	200	0.4
Sorted-Insert-DNodes	NL∧AC∧SO∧IN	25	3636	175.7
Lazy-Simple [7]*	AC∧DT	21	9290	33.4
Lazy-Simple-Backw [7]*	AC∧DT	15	1127	2.2
Init-Increment*	AC∧DT	11	354	1.6
Init-Add*	AC∧DT	11	354	1.8
Init-Add-Flag*	AC∧DT	12	499	1.4
Init-Mult*	AC∧DT	11	354	1.8

Table 2. Performance on Examples with Integer Data Fields. These examples could not be verified without the SMT integration. Some examples are the same as in Table 1, except with integer data fields; other examples, marked with *, are completely new. Pseudocode and the required predicates for these examples can be downloaded from `http://www.cs.ubc.ca/~zrakamar/software/hmp-examples.tar.gz`.

provides access to the rich set of theories it supports. Using a combination of different theories of the extended MathSAT, we verified HMP examples we couldn't handle before. Comparing running times to our previous work shows that the much greater expressiveness comes with only a minor performance penalty. We believe this integration of an HMP-verification logic into a general SMT solver will be broadly applicable to many software verification tools, allowing them to be easily extended to handle both heap-related and other software verification properties.

The primary direction for future work is to improve our predicate abstraction framework to make better use of the capabilities of the combined SMT prover. Our simple predicate abstraction engine eagerly enumerates a huge number of small queries to the SMT solver and is therefore not benefiting from the solver's powerful search algorithm. Using techniques similar to the *AllSAT* approach to predicate abstraction [26] should substantially improve performance.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In *Conf. on Computer Aided Verification (CAV)*, pages 371–383, 2007.

3. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2005.
4. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, 2004.
6. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming (ESOP)*, 1999.
7. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Conf. on Computer Aided Verification (CAV)*, pages 532–546, 2006.
8. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Conf. on Computer Aided Verification (CAV)*, pages 504–518, 2007.
9. J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 207–221, 2006.
10. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, S. Ranise, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Conf. on Computer Aided Verification (CAV)*, pages 335 – 349, 2005.
11. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, S. Ranise, and R. Sebastiani. Efficient theory combination via boolean search. *Information and Computation*, 204:1493 – 1525, 2006.
12. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 system. In *Intl. Conf. on Automated Deduction (CADE)*, pages 315–321, 2005.
13. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: A comparative analysis. In *Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 527–541, 2006.
14. N. Charlton and M. Huth. Hector: Software model checking with cooperating analysis plugins. In *Conf. on Computer Aided Verification (CAV)*, 2007.
15. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.
16. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
17. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA.
18. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.
19. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conf. on Computer Aided Verification (CAV)*, 1997.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symp. on Principles of Programming Languages (POPL)*, pages 58–70, 2002.
21. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *Workshop on Computer Science Logic (CSL)*, pages 160–174, 2004.

22. F. Ivančić, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *Intl. Conf. on Computer Design (ICCD)*, pages 297–308, 2005.
23. J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 226–236, 1997.
24. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Conf. on Implementation and Application of Automata (CIAA)*, 2000.
25. S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 618–631, 2007.
26. S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Conf. on Computer Aided Verification (CAV)*, pages 413–426, 2006.
27. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symp. on Principles of Programming Languages (POPL)*, pages 115–126, 2006.
28. S. K. Lahiri and S. Qadeer. A decision procedure for well-founded reachability, 2007. Microsoft Research Tech Report MSR-TR-2007-43.
29. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conf. on Automated Deduction (CADE)*, 2005.
30. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301, 2000.
31. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 181–198, 2005.
32. Z. Manna and C. G. Zarba. Combining decision procedures. In B. K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2002.
33. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Conf. on Computer Aided Verification (CAV)*, pages 476–490, 2005.
34. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 221–231, 2001.
35. G. Nelson. *Techniques for program verification*. PhD thesis, Stanford University, 1979.
36. G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Principles of Programming Languages (POPL)*, pages 38–47, 1983.
37. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
38. Z. Rakamarić, J. Bingham, and A. Hu. A better logic and decision procedure for predicate abstraction of heap-manipulating programs, 2006. UBC Dept. Comp. Sci. Tech Report TR-2006-02, http://www.cs.ubc.ca/cgi-bin/tr/2006/TR-2006-02.
39. Z. Rakamarić, J. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 106–121, 2007.
40. S. Ranise and C. G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM)*, 2006.
41. G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Foundations of Software Science and Computation Structures (FOSSACS)*, 2006.