Smtlink 2.0

Yan Peng Mark R. Greenstreet
University of British Columbia*
Vancouver, Canada
`yanpeng,mrg@cs.ubc.ca`

`Smtlink` is an extension of ACL2 with Satisfiability Modulo Theories (SMT) solvers. We presented an earlier version at ACL2'2015. `Smtlink` 2.0 makes major improvements over the initial version with respect to soundness, extensibility, ease-of-use, and the range of types and associated theory-solvers supported. Most theorems that one would want to prove using an SMT solver must first be translated to use only the primitive operations supported by the SMT solver – this translation includes function expansion and type inference. `Smtlink` 2.0 performs this translation using a sequence of steps performed by verified clause processors and computed hints. These steps are ensured to be sound. The final transliteration from ACL2 to Z3's Python interface requires a trusted clause processor. This is a great improvement in soundness and extensibility over the original `Smtlink` which was implemented as a single, monolithic, trusted clause processor. `Smtlink` 2.0 provides support for FTY `defprod`, `deflist`, `defalist`, and `defoption` types by using Z3's arrays and user-defined data types. We have identified common usage patterns and simplified the configuration and hint information needed to use `Smtlink`.

1 Introduction

Interactive theorem proving and SMT solving are complementary verification techniques. SMT solvers can automatically discharge proof goals with thousands to hundreds of thousands of variables when the goals are within the theories of the solver. These theories include linear and non-linear arithmetic, uninterpreted functions, arrays, and bit-vector theories. On the other hand, verification of realistic hardware and software systems often involves models with a rich variety of data structures. Their proofs involve induction, encapsulation (or other forms of higher-order reasoning), and careful design of rules to avoid pushing the solver over an exponential cliff of search complexity. Ideally, we would like to use the capabilities of SMT solvers to avoid proving large numbers of simple but tedious lemmas and combine those results in an interactive theorem prover to enable the reasoning of properties in large, realistic hardware and software designs.

`Smtlink` is a book developed for integrating SMT solvers into ACL2. In the previous work [22], we implemented an interface using a large trusted clause-processor, and only supported a limited subset of SMT theories. In this work, we introduce an architecture based on a collection of verified clause processors that transform an ACL2 goal into a form amenable for discharging with an SMT solver. The final step transliterates the ACL2 goal into the syntax of the SMT solver, invokes the solver, and discharges the goal or reports a counter-example based on the results from the solver. This final step is, performed by a trusted-clause processor because we are trusting the SMT solver. Because most of the translation is performed by verified clause processors, the soundness for those steps is guaranteed. Furthermore, the task of the final, trusted clause processor is simple to describe, and that description suggests the form for the corresponding soundness argument. The modularized architecture is readily extensible which makes introducing new clause transformation steps straightforward.

*This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

Our initial motivation for linking SMT solvers with ACL2 was for proving linear and non-linear arithmetic for Analog and Mixed-Signal (AMS) designs [23]. This original version supported the SMT solver's theories of boolean satisfiability and integer and rational arithmetic, but provided no support for other types. For example, to verify theorems involving lists, one would need to expand functions involving lists to a user-specified depth, treat remaining calls as uninterpreted functions which must return a boolean, integer, rational, or real. The lack of support for a rich set of types restricted the applicability of the original `Smtlink` to low-level lemmas that are primarily based on arithmetic. The new `Smtlink` supports symbols, lists, and algebraic datatypes, with a convenient interface to FTY types.

Other changes include better support for function expansion and uninterpreted functions. The new `smtlink-hint` interface is simpler. `Smtlink` generates auxiliary subgoals for ACL2 in "obvious" cases, such as when the user adds a hypothesis for `Smtlink` that is not stated in the original goal. Hints for these subgoals share the same structure as the ACL2 hints and are attached by the `Smtlink` hint syntax to the relevant subgoal – no subgoal specifiers are required! When the SMT solver refutes a goal, the putative counterexample is returned to ACL2 for user examination. Currently, `Smtlink` supports the Z3 SMT solver [20] using Z3's Python API. `Smtlink` is compatible with both Python 2 and Python 3, and can be used in ACL2(r) with `realp`.

Documentation is online under the topic `:doc smtlink`. It describes how to install Z3, configure `Smtlink`, certify `Smtlink` and test the installation. It also describes the new interface of the `Smtlink` hint, contains several tutorial examples, and some developer documentation. Examples shown in this paper assumes proper installation and setups are done as is described in `:doc smtlink`. It is our belief that given a more compelling argument of soundness for the architecture, a richer set of supported types or theories, and a more user-friendly user interface, `Smtlink` can support broader and larger proofs.

In Section 2 we present the architecture of `Smtlink` based on verified clause processors, hint wrappers, and computed hints. Section 3 describes the SMT theories supported by `Smtlink` and sketches the soundness argument. Section 4 gives a simple example where we use lists, alists, symbols, and booleans to model the behavior of a ring oscillator circuit. Section 5 provides a summary of related work, and we summarize the paper along with describing ongoing and planned extensions to `Smtlink` in Section 6.

2 Smtlink 2.0 Architecture

This section describes the architecture of `Smtlink` 2.0. We first introduce a running example to illustrate each of the steps taken by the clause processor. Section 2.2 describes the top-level architecture, followed by a brief description of each of the clause processors used in `Smtlink`.

2.1 An Nonlinear Inequality Example

To illustrate the architecture of `Smtlink`, we use a running example throughout this section. Consider proving the following theorem in ACL2:

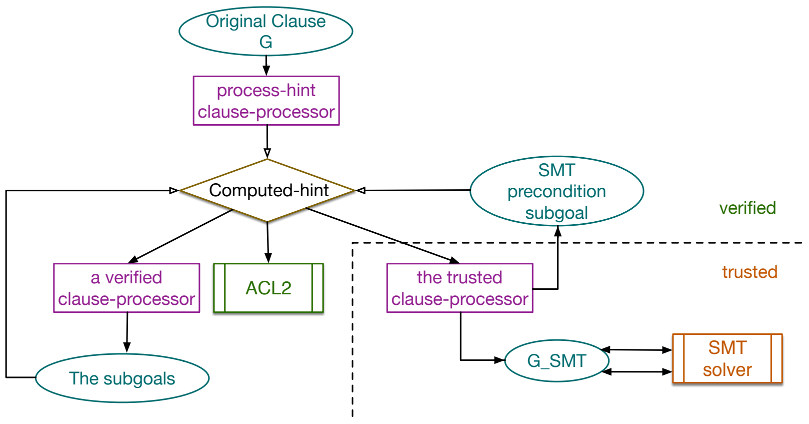
Theorem 2.1 *$\forall x \in R$ and $\forall y \in R$, if $\frac{9x^2}{8} + y^2 \le 1$ and $x^2 - y^2 \le 1$, then $y < 3(x - \frac{17}{8})^2 - 3$.*

Program 2.1 shows the corresponding theorem definition in ACL2. To use `Smtlink` to prove a theorem, a hint `:smtlink [smt-hint]` can be provided using ACL2's standard `:hints` interface. `SMT::smt-hint` is the hint provided to `Smtlink`. `SMT::smt-hint` follows a structure that is described in `:doc smt-hint`. In this example, `:smtlink nil` suggests using `Smtlink` without any additional hints provided to `Smtlink`. The initial goal (underlining represented as clauses) is:

Program 2.1 A nonlinear inequality problem

```
(defun x^2-y^2 (x y) (- (* x x) (* y y)))

(defthm poly-ineq-example
  (implies (and (real/rationalp x) (real/rationalp y)
                (<= (+ (* (/ 9 8) x x) (* y y)) 1)
                (<=  (x^2-y^2 x y) 1))
           (< y (- (* 3 (- x (/ 17 8)) (- x (/ 17 8))) 3)))
  :hints(("Goal"
          :smtlink nil)))
```



smt-architecture table

step tag	next clause-processor
process-hint	add-hypo-cp
add-hypo	expand-cp
expand	type-extract-cp
type-extract	uninterpreted-fn-cp
uninterpreted	smt-trusted-cp
uninterpreted-custom	smt-trusted-cp-custom

Figure 1: Smtlink Architecture

```
(IMPLIES (AND (RATIONALP X) (RATIONALP Y)
              (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (X^2-Y^2 X Y) 1))
         (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                         (+ X (- (* 17 (/ 8))))))))
```

2.2 The Architecture

Let G denote the goal to be proven. As shown in Figure 1, the `Smtlink` hint invokes the verified clause processor called `SMT::process-hint`. The arguments to this clause-processor are the clause, G, and a list of hints provided by the user. The `SMT::process-hint` performs syntactic checks on the user's hints and translates them into an internal representation used by the subsequent clause processors. The user can specify default hints to be used with all invocations of `Smtlink`. These are merged with any hints that are specific for this theorem to produce `combined-hint`. The `SMT::process-hint` adds a `SMT::hint-please` wrapper to the clause and returns a term of the form

```
`(((SMT::hint-please (:clause-processor (SMT::add-hypo-cp clause ,combined-hint))) ,@G))
```

`Smtlink` 2.0 uses the hint wrapper approach from `books/hints/hint-wrapper.lisp`. In particular, we define a "hint-wrapper" called `SMT::hint-please` in package `smtlink` that always returns `nil`.

Clauses in ACL2 are represented as lists of disjuncts, and our hint-wrapper can be added as a disjunct to any clause without changing the validity of the clause. `SMT::hint-please` takes one input argument – the list of hints.

The computed-hint called `SMT::SMT-computed-hint` searches for a disjunct of the form `(SMT::hint-please ...)` in each goal generated by ACL2. When it finds such an instance, the `SMT::SMT-computed-hint` will return a `computed-hint-replacement` of the form:

```
‘(:computed-hint-replacement
  ((SMT::SMT-computed-hint clause))
  (:clause-processor (SMT::some-verified-cp clause ,combined-hint)))
```

This applies the next verified clause-processor called `SMT::some-verified-cp` to the current subgoal and installs the computed-hint `SMT::SMT-computed-hint` on subsequent subgoals again. The `SMT::some-verified-cp` clause-processor is one step in a sequence of verified clause processors. `Smtlink` uses a configuration table called `*smt-architecture*` to specify the sequence of clause processors, as shown in Figure 1. Each clause processor consults this table to determine its successor. By updating this table, `Smtlink` is easily reconfigured.

As described above, the initial clause processor for `Smtlink`, `SMT::process-hint`, adds a `SMT::hint-please` disjunct to the clause to indicate that the clause processor, `SMT::add-hypo-cp` should be the next step in translating the clause to a form amenable for a SMT solver.

2.2.1 add-hypo-cp

A key to using an SMT solver effectively is to tell it what it needs to know but to avoid overwhelming it with facts that push it over an exponential complexity cliff. The user can guide this process by adding hypotheses for the SMT solver – typically these are facts from the ACL2 logical world that don't need to be stated as hypotheses of the theorem. The clause-processor `SMT::add-hypo-cp` is a verified clause processor that adds user-provided hypotheses to the goal. Let **G** denote the original goal and $\mathbf{H_1}$, ... $\mathbf{H_n}$ denote the new hypotheses. Each added hypothesis is returned by the clause processors as a subgoal to be discharged by ACL2. The user can attach hints to these hypotheses – for example, showing that the hypothesis is a particular instantiation of a previously proven theorem. The soundness of `SMT::add-hypo-cp` is established by the theorem:

$$\frac{H_1 \vee G \quad \ldots \quad H_n \vee G \quad \bigwedge_{i=1}^{N} H_i \Rightarrow G}{G} \tag{1}$$

The term $\bigwedge_{i=1}^{N} H_i \Rightarrow G$ is the main clause that gets passed onto the next clause-processor. In the following, let G_{hyp} denote $\bigwedge_{i=1}^{N} H_i \Rightarrow G$.

As for the example in Program 2.1, the user didn't provide any additional guidance. We see that the clauses generated are almost unchanged. The only place that changed, is that `Smtlink` has installed the next clause-processor to be `SMT::expand-cp`.

```
(IMPLIES (NOT (SMT::HINT-PLEASE '(:CLAUSE-PROCESSOR (SMT::EXPAND-CP CLAUSE ...))))
 (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
               (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
               (<= (X^2-Y^2 X Y) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                     (+ X (- (* 17 (/ 8)))))))))
```

2.2.2 expand-cp

The clause-processor `SMT::expand-cp` expands function definitions to produce a goal where all operations have SMT equivalents. Function definitions are accessed using `meta-extract-formula`. By default, all non-recursive functions including disabled functions are expanded. Recursive functions are expanded once and then treated as uninterpreted functions. We attempt one level of expansion for recursive functions given the reasoning that if the theorem can indeed be proved only knowing the return type of the function, it should still be provable when expanded once. If proving the theorem requires expansion of more than one level, or if all occurrences of the function should uninterpreted, then we rely on the user to specify this in a hint to `Smtlink`. Let $\mathbf{G}_{\text{hyp}}$ be the clause given to `SMT::expand-cp` and $\mathbf{G}_{\text{expand}}$ be the expanded version. The soundness of `SMT::expand-cp` is established by the theorem:

$$\frac{G_{\text{expand}} \Rightarrow G_{\text{hyp}} \quad G_{\text{expand}}}{G_{\text{hyp}}} \tag{2}$$

Presently, the clause $G_{\text{expand}} \Rightarrow G_{\text{hyp}}$ is returned to ACL2 for proof, and G_{expand} is the main clause that gets passed onto the next clause-processor.

For the running example, function expansion produces the two subgoals. The main clause that get passed onto the next clause-processor is:

```
(IMPLIES (NOT (SMT::HINT-PLEASE '(:CLAUSE-PROCESSOR (SMT::TYPE-EXTRACT-CP CLAUSE ...))))
 (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
               (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
               (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                        (+ X (- (* 17 (/ 8)))))))))
```

It tells `Smtlink` the next clause-processor is `SMT::type-extract-cp` which does type declaration extraction. The other subgoal is:

```
(IMPLIES
 (AND (NOT (SMT::HINT-PLEASE '(:IN-THEORY (ENABLE X^2-Y^2))))
      (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
                    (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                    (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
               (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                             (+ X (- (* 17 (/ 8)))))))))
 (OR ... ;; some term essentially equal to nil
     (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
                   (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                   (<= (X^2-Y^2 X Y) 1))
           (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                        (+ X (- (* 17 (/ 8))))))))))
```

This second subgoal essentially proves that the expanded clause implies the original clause. Notice how `SMT::hint-please` is used again for passing other sorts of hints (that are not clause-processor hints) to ACL2 for help with the proof.

2.2.3 type-extract-cp

The logic of ACL2 is untyped; whereas SMT solvers such as Z3 use a many-sorted logic. To bridge this gap, the clause-processor `SMT::type-extract-cp` is a verified clause processor for extracting type information for free variables from the hypotheses of a clause. `SMT::type-extract-cp` traverses the clause and identifies terms that syntactically are hypotheses of the form `(type-p var)` where `type-p` is a known type recognizer, and `var` is a symbol. Let G_{expand} denote the clause given to `SMT::type-extract-cp`; $\mathbf{T_1}$, ... $\mathbf{T_m}$ denote the extracted type hypotheses; $\mathbf{G}_{\text{expand}\backslash\text{type}}$ denote G_{expand} with the type-hypotheses removed; and

$$G_{\text{type}} = (\texttt{SMT::type-hyp (list } T_1 ... T_m\texttt{):type}) \Rightarrow G_{\text{expand}\backslash\text{type}} \tag{3}$$

where `SMT::type-hyp` logically computes the conjunction of the elements in the list (`list` T_1 ... T_m) – using `SMT::type-hyp` makes these hypotheses easily identified by subsequent clause processors. The soundness of `SMT::type-extract-cp` is established by the theorem:

$$\frac{G_{\text{type}} \Rightarrow G_{\text{expand}} \quad G_{\text{type}}}{G_{\text{expand}}} \tag{4}$$

$G_{\text{type}} \Rightarrow G_{\text{expand}}$ is the auxiliary clause returned back into ACL2 for proof. G_{type} is the main clause that gets passed onto the next clause-processor.

For the running example, the main clause that gets passed onto the next clause-processor is:

```
(IMPLIES (AND (NOT (SMT::HINT-PLEASE
                    '(:CLAUSE-PROCESSOR (SMT::UNINTERPRETED-FN-CP CLAUSE ...))))
              (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE))
 (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
          (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8)))))
                         (+ X (- (* 17 (/ 8))))))))
```

The other auxiliary clause is:

```
(IMPLIES (AND (NOT (SMT::HINT-PLEASE
                    '(:IN-THEORY (ENABLE SMT::HINT-PLEASE SMT::TYPE-HYP)
                      :EXPAND ((:FREE (SMT::X) (HIDE SMT::X))))))
              (IMPLIES (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE)
                (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
                         (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8)))))
                                        (+ X (- (* 17 (/ 8))))))))))
 (IMPLIES (AND (RATIONALP X) (RATIONALP Y)
              (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
              (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
              (< Y (+ -3  (* 3 (+ X (- (* 17 (/ 8)))))
                             (+ X (- (* 17 (/ 8))))))))))
```

This clause is returned for proof by ACL2. The proof is straightforward – essentially ACL2 simply needs to confirm that the terms we identified as type-hypotheses really are hypotheses of the goal.

2.2.4 uninterpreted-fn-cp

Useful facts about recursive functions can be proven using the SMT solver's support for uninterpreted functions. As with variables, the return-types for these functions must be specified. For soundness, `Smtlink` must show that the ACL2 function satisfies the user given type constraints. This is done by `SMT::uninterpreted-fn-cp`. Let $\mathbf{G}_{\text{type}}$ denote the clause given to `SMT::uninterpreted-fn-cp`, and let $\mathbf{R_1}$, ... $\mathbf{R_p}$ denote the assertions about the types for each call to an uninterpreted function. Let $\mathbf{Q_1}$ be the list of clauses

$$\begin{aligned} Q_1 \quad = \quad & (\texttt{SMT::type-hyp}\ (\texttt{list}\ R_1)\ \texttt{:return}) \vee G_{\text{type}}\ ,\ \dots\ , \\ & (\texttt{SMT::type-hyp}\ (\texttt{list}\ R_p)\ \texttt{:return}) \vee G_{\text{type}} \end{aligned} \tag{5}$$

Finally, let $\mathbf{G_{tcp}}$ be the clause

$$G_{tcp} = \bigwedge_{i=1}^{p} (\texttt{SMT::type-hyp}\ (\texttt{list}\ R_i)\ \texttt{:return}) \Rightarrow G_{\text{type}} \tag{6}$$

The soundness of `expand-cp` is established by the theorem:

$$\frac{Q_1 \quad G_{tcp}}{G_{\text{type}}} \tag{7}$$

The list of clauses Q_1 is returned to ACL2 for proof, and G_{tcp} is tagged with a hint to be checked by the trusted clause processor.

Our running example does not make use of uninterpreted functions in the SMT solver; so, the clause G_{tcp} is the same as G_{type} except for the detail that the `SMT::hint-please` term now specifies that the next clause-processor to use is the final trusted clause-processor `SMT::smt-trusted-cp`:

```
(IMPLIES (NOT (SMT::HINT-PLEASE '(:CLAUSE-PROCESSOR (SMT::SMT-TRUSTED-CP CLAUSE ... STATE))))
 (OR (NOT (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE))
     (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                   (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
              (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                              (+ X (- (* 17 (/ 8))))))))))
```

2.2.5 smtlink-trusted-cp

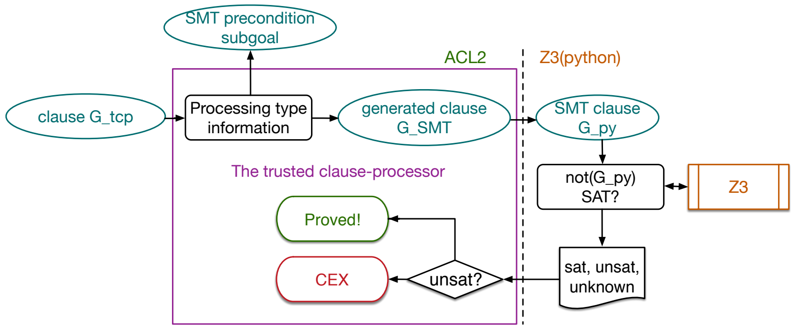


Figure 2: The trusted clause processor

Figure 2 shows the internal architecture of the trusted clause-processor, for which its correctness hasn't been proven. `Smtlink` returns counter-examples generated by the SMT solver back into ACL2. The Python interface to Z3 for `Smtlink` includes code to translate a counterexample from Z3 into list-syntax for ACL2. The form is not necessarily in ACL2 syntax. We wrote a printing function that takes the Z3 counter-examples and prints it out in an ACL2-readable form. These are all done through the trusted clause processor. Currently, the forms returned into ACL2 are not evaluable. For example, counter-examples for real numbers can take the form of an algebraic number, e.g.

```
((Y (CEX-ROOT-OBJ Y STATE (+ (^ X 2) (- 2)) 1)) (X -2))
```

We plan to generate evaluable forms in the future. Learn more about counter-example generation, see `:doc tutorial`.

The trusted clause-processor returns a subgoal called "SMT precondition" back into ACL2. By ensuring that certain preconditions are satisfied, we are able to bridge the logical meaning for tricky cases, and therefore ensure soundness. We will see explanations on this issue in Section 3. For this running example, there are no preconditions to be satisfied and the following clause trivially holds:

```
(IMPLIES (AND (NOT (SMT::HINT-PLEASE
                    '(:IN-THEORY (ENABLE SMT::MAGIC-FIX SMT::HINT-PLEASE SMT::TYPE-HYP)
                      :EXPAND ((:FREE (SMT::X) (HIDE SMT::X))))))
              (NOT (AND (OR (NOT (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1))
                            (LET NIL T))
                        T)))
 (OR (NOT (SMT::TYPE-HYP (HIDE (LIST (RATIONALP X) (RATIONALP Y))) :TYPE))
     (IMPLIES (AND (<= (+ (* (* 9 (/ 8)) X X) (* Y Y)) 1)
                   (<= (LET NIL (+ (* X X) (- (* Y Y)))) 1))
              (< Y (+ -3 (* 3 (+ X (- (* 17 (/ 8))))
                              (+ X (- (* 17 (/ 8))))))))))
```

2.3 A Few Notes about the Architecture

The new architecture clearly separates what's verified from what's trusted. As is shown in Figure 1, given an original goal, the `Smtlink` workflow goes through a series of verified clause-processor, which won't change the logical meaning of the original goal. A computed-hint is installed to provide hints for the next step, but won't change the logical meaning of the goals either. The final trusted clause-processor takes a goal that logically implies the original goal and derives information needed for the translation through three sources of information it uses. First, information encoded in the clause using `SMT::type-hyp`. This is sound because the definition of `SMT::type-hyp` is a conjunction of the input boolean list. Second, information about FTY types from `fty::flextypes-table`. Presently, we trust this table to be correct and intact. We plan to use rules about FTY types to derive this information in the future. Third, we use information stored in `SMT::smt-hint` about configurations, including what is the Python command, whether to treat integers as reals, where is the `ACL2_to_Z3` class file and so on. We believe this information can not accidentally introduce soundness issues from the user. For experimenters and developers, we allow this low-level interface to be overridden by the user using `:smtlink-custom` hint. Using such a "customized" `Smtlink` requires a different trust-tag than the standard version, thus providing a firewall between experiments and production versions. See `:doc tutorial` for how to use customizable `Smtlink`.

3 Types, Theories, and Soundness

`Smtlink` translates the original goal, G to an expanded goal, G_{tcp} for the trusted clause processor through a series of verified clause processors. Thus, we regard the translation to G_{tcp} as sound. The trusted clause processor translates G_{tcp} into a form that can be checked by the SMT solver; we refer to this translated form as G_{smt}. Let $x_1, x_2, \ldots, x_n$ denote the free variables in G_{tcp}, let G_{smt} denote the translated goal, and

$\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n$ denote the free variables of G_{smt}. For soundness, we want

$$\text{SMT} \vdash G_{\text{smt}} \quad \Rightarrow \quad \text{ACL2} \vdash G_{\text{tcp}} \tag{8}$$

In the remainder, we assume

- ACL2 and the SMT solver are both sound for their respective theories.
- The SMT solver is a decision procedure for a decidable fragment of first-order logic. In particular, this holds for Z3, the only SMT solver that is currently supported by `Smtlink`. In addition, we are working with a quantifier-free fragment of Z3's logic.
- There is a one-to-one correspondence between the free variables of G_{tcp} and the free variables of G_{smt}. This is the case with the current implementation of `Smtlink`.

Now, suppose that G_{tcp} is not a theorem in ACL2. Then, by Gödel's Completeness Theorem, there exists a model of the ACL2 axioms that satisfies $\neg G_{\text{tcp}}$. We need to show that in this case there exists a model of the SMT solver's axioms that satisfies $\neg G_{\text{smt}}$. There are two issues that we must address. First, we need to provide, for the interpretation of any function symbol $\texttt{f}_{\text{acl2}}$ in G_{tcp}, an interpretation for the corresponding function symbol $\texttt{f}_{\text{smt}}$ in G_{smt}. This brings us to the second issue: the logic of ACL2 is untyped, but the logic of SMT solvers including Z3 is many-sorted. Thus, there are models of the ACL2 axioms that have no correspondence with the models of the SMT solver. We restrict our attention to goals, G_{tcp} where the type of each subterm in the formula can be deduced. We refer to such terms as translatable. If G_{tcp} is not translatable, then `Smtlink` will fail to prove it.

For the remainder, we restrict our attention to translatable goals. Because G_{tcp} is translatable, there is a set R of unary recognizer functions (primitives such as rationalp that return a boolean) and also a set S of other functions, such that every function symbol in G_{tcp} is a member of R or of S, and every function in S is "well-typed" with respect to R in some sense that we can define roughly as follows. We associate each function symbol $\texttt{f}_{\text{acl2}}$ in S with a function symbol $\texttt{f}_{\text{smt}}$ of Z3, and each predicate r in R with a type in Z3. The trusted clause processor checks that there is a "type-hypothesis" associated with every free variable of G_{tcp} and a fixing function for every type-ambiguous constant (e.g. `nil`) – G_{tcp} holds trivially if any of these type-hypotheses are violated. For every function $\texttt{f}_{\text{acl2}}$ in S, we associate a member of R to each of its arguments (i.e. a "type") and also to the result. For user-defined functions (i.e. uninterpreted function for the SMT solver), `Smtlink` generates a subgoal for each call to $\texttt{f}_{\text{smt}}$: if the arguments satisfy their declared types (i.e., predicates from R), then the result must satisfy its declared type as well. For built-in ACL2 functions (e.g. `binary-plus`) we assume the "obvious" theorems are present in the ACL2 logical world. Now suppose we have a model, M_1, of $\neg G_{\text{tcp}}$, and consider the submodel, M_2, containing just those objects m such that m satisfies at least one predicate in R that occurs in G_{tcp}. Note that M_2 is closed under (the interpretation of) every operation in S, because $\neg G_{\text{tcp}}$ implies that all of the "type-hypotheses" of G_{tcp} are true in M_1. This essentially excludes "bad atoms", as defined by the function `acl2::bad-atom`. Then because G_{tcp} is quantifier-free, M_2 also satisfies $\neg G_{\text{tcp}}$. We can turn M_2 into a model M_2' for the language of Z3, by assigning the appropriate type to every object. (As noted in Section 3.1, M_2' satisfies the theory of Z3 if M_2 is a model of ACL2(r); but for ACL2 that is not the case, so in future work, we expect to construct an extension of M_2' that satisfies all of the axioms for real closed fields.) Then we have the claim: for every assignment s from the free variables of G_{tcp} to M_2 with corresponding typed assignment s' from the free variables of G_{smt} to M_2' , if $\neg G_{\text{tcp}}$ is true in M_2 under s, then $\neg G_{\text{smt}}$ is true in M_2' under s'. Thus, if G_{tcp} is translatable, and $\neg G_{\text{smt}}$ is unsatisfiable, we conclude that G_{tcp} is a theorem in ACL2.

In the rest of this section, we discuss for each of the recognizer functions and each of the basic functions in ACL2, how we associate them with the corresponding Z3 functions.

3.1 Booleans, Integers, Rationals, and Reals

If a term is a boolean constant, then the translation to the SMT solver is direct. Likewise, if x_i is free in G_{tcp} and `(booleanp` x_i`)` is one of the hypotheses of G_{tcp}, then G_{tcp} holds trivially in the case that $x_i \notin \{\texttt{t}, \texttt{nil}\}$. Thus, in G_{smt} `Smtlink` can represent the hypothesis `(booleanp` x_i`)` with the declaration

```
x_i = Bool('x_i')
```

without excluding any satisfying assignments. We assume that the boolean operations of the SMT solver (e.g. `And`, `Or`, `Not`) correspond exactly to their ACL2 equivalents when their arguments are boolean. If a boolean operator is applied to a non-boolean value, then Z3 throws an exception, and we regard G as non-translatable.

Similar arguments apply in the case that x_i is an integer, rational, or real number. We represent ACL2 rational numbers as Z3 real numbers. Because every rational number is a real number, any satisfying assignment of rational numbers to rational variables in $\neg G_{\text{tcp}}$ has a corresponding assignment for $\neg G_{\text{smt}}$. Thus, G_{smt} is a generalization of G_{tcp}. We note that for ACL2, formally proving the soundness of this generalization requires extending our previously discussed M_2' model into a model that satisfies the theory of real closed fields [1], because we are translating rationals in ACL2 to reals in Z3. We haven't wrapped our heads around how to do that extension in a many-sorted setting, therefore we designate this to be future work. As with booleans, we assume that arithmetic and comparison operators have equivalent semantics in ACL2 and the SMT solver. In fact, we use the Python interface code to enforce this assumption. As an example, ACL2 allows the boolean values `t` and `nil` to be used in arithmetic expressions – both are treated as 0. Z3 also allows `True` and `False` to be used in integer arithmetic, with `True` treated as 1 and `False` treated as 0. To ensure that $G_{\text{smt}} \Rightarrow G_{\text{tcp}}$, our Python code checks the sorts of the arguments to arithmetic operators to ensure that they are integers or reals, where the interpretations are the same for both ACL2 and Z3.

When `Smtlink` is used with ACL2(r), non-classical functions are non-translatable. We believe that if $\neg G_{\text{tcp}}$ is classical and satisfiable, then there exists a satisfying assignment to $\neg G_{\text{tcp}}$ where all real-valued variables are bound to standard values. We believe the sketched proof in the beginning of Section 3 works well for ACL2(r). If we are wrong, we hope that one of the experts in non-standard analysis at the workshop will correct us.

3.2 Symbols

A very important basic type in ACL2 is `symbolp`. We represent symbols using an algebraic datatype in Z3. In the z3 interface class, we define a `Datatype` called `z3sym`, with a single field of type `IntSort`. Symbol variables are defined using the datatype `z3sym`. We then define a class called `Symbol`. This class provides a variable `count` and a variable `dict`. It also provides a function called `intern` for generating a symbol constant. This class keeps a dictionary mapping from symbol names to the generated `z3sym` symbol constants. This creates an injective mapping from symbols to natural numbers. All symbol constants that appeared in the term are mapped onto the first several, distinct, naturals.

If a satisfying assignment to $\neg G_{\text{tcp}}$ binds a symbol-valued variable to a symbol-constant that doesn't appear in G_{tcp}, then in our soundness argument, we construct a new symbol value for $\neg G_{\text{smt}}$ using an integer value distinct from the ones used so far – we won't run out. Thus, all symbol values in a satisfying assignment to $\neg G_{\text{tcp}}$ can be translated to corresponding values for $\neg G_{\text{smt}}$. The only operations that we support for symbols are comparisons for equality or not-equals. We assume that these operations have corresponding semantics in ACL2 and the SMT solver.

[1] http://smtlib.cs.uiowa.edu/theories-Reals.shtml

3.3 FTY types

We have added support for common `fty` types that enable `Smtlink` to automatically construct bridges from the untyped logic of ACL2 to the typed logic of Z3. Currently, `Smtlink` infers constructor/destructor relations and other properties of these types from the `fty::flextypes-table`. Thus, the use of `fty` types extends the trusted code to include the correctness of these tables. This trust is mitigated by two considerations. First, `Smtlink` only uses `fty` types that have been specified by the user in a hint to the `Smtlink` clause processor. If the user provides no such hints, no `fty` types are used by `Smtlink`, and no soundness concerns arise.

Second, we expect that the information that `Smtlink` obtains from these tables could be obtained instead from the ACL2 logical world using `meta-extract` in `Smtlink`'s verified clause-processor chain. We see the current implementation as a useful prototype to explore how to seamlessly infer type information from code written according to a well-defined type discipline.

3.3.1 fty::defprod

The algebraic datatypes of Z3 correspond directly to `fty::defprod`. `Smtlink` simply declares a Z3 datatype with a single constructor whose destructor operators are the field accessors of the product type. `Smtlink` requires that the arguments to the `fty` constructors satisfy the constructors' guards – otherwise G_{tcp} is non-translatable. The only operations on product types are field accessors, i.e. destructors. For translatable terms, the SMT type has the same construct/destructor theorems as the FTY type. Thus, the `Smtlink` translation maintains equivalence constructors and field accessors of product types.

3.3.2 fty::deflist

Lists are essentially a special case of a product type. For example,

Listing 1: ACL2 deflist

```
(deflist integer-list
  :elt-type integerp
  :true-listp t)
```

Listing 2: Z3 Datatype

```
integer_list= z3.Datatype('integer_list')
integer_list.declare('cons',
                     ('car', _SMT_.IntSort()),
                     ('cdr', integer_list))
integer_list.declare('nil')
integer_list = integer_list.create()
def integer_list_consp(l):
  return Not(l == integer\_list.nil)
```

ACL2 overloads `cons`, `car`, and `cdr` to apply to any list. In contrast, Z3's typed logic has a separate `cons`, `car`, `cdr` functions for each list type. This is why our examples from Section 4 require fixing functions to convey the type information to the trusted-clause processor. We believe that most of the users' burden of typed lists will be removed in a future release by adding type-inference to `Smtlink`. There are soundness issues that must be addressed. In ACL2, `(equal (car nil) nil)`. In Z3,

    integer_list.car(integer_list.nil)

is an arbitrary integer. To ensure soundness, the trusted-clause processor produces the proof obligation `(consp x)` for every occurrence of `(car x)` that it encounters. Under this precondition, the Z3 translation preserves the constructor/destructor relationship for lists. Likewise

    integer_list.cdr(integer_list.nil)

is an arbitrary `integer_list`. Thus, `Smtlink` enforces the `:true-listp t` declaration for list types.

Because “arbitrary” includes `integer_list.nil` in addition to all other `integer_lists`, these construction ensures that the SMT solver can choose the value for `integer_list.cdr(x̃)` for G_{smt} that corresponds to the value of `(cdr x)` for any assignments for G_{tcp}. The G_{smt} is a generalization of G_{tcp}.

3.3.3 fty::defalist

`Smtlink` represents alists with SMT arrays. We only support the operations `acons` and `assoc-equal` for alists. Then we have:

```
(defthm alist-axioms
  (implies (not (equal key1 key2))
           (and (equal (assoc key1 (acons key1 value alist)) value))
                (equal (assoc key1 (acons key2 value alist)) (assoc key1 alist)))
                (equal (assoc key1 nil) nil))
```

The corresponding theorem in the theory of arrays is

```
(defthm array-axioms
  (implies (not (equal addr1 addr2))
           (and (equal (load addr1 (store addr1 value array)) value))
                (equal (load addr1 (store addr2 value array)) (load addr1 array))))
```

Note that `Smtlink` does not support operations such as `cdr`, `nth`, `member`, or `delete-assoc` that would “remove” elements from an alist. Also, Z3 arrays are typed.

The key issue in the translation is how to handle the case when a key is not found in the alist (ACL2) or array (SMT). Our solution is to make the element type of the SMT array be an option type called `keyType_elementType`. This type can either be a `(key, value)` tuple or `keyType_-elementType.nil`. Thus, any value returned by `assoc-equal` with proper alist and key types has a corresponding `keyType_elementType` value. Thus, any value for an `assoc-equal` terms in G_{tcp} can be represented in G_{smt}.

When applying `cdr` to a `keyType_elementType`, we must ensure that the `keyType_elementType` value is not nil. This is analogous to the issue with lists: in ACL2, `(equal (cdr nil) nil)` but in Z3,

`keyType_elementType.cdr(keyType_elementType.nil)`

is an arbitrary value of `elementType`. Thus, the trusted-clause processor produces the proof obligation for ACL2 `(not (null x))` for every occurrence of `(cdr x)` when `x` is the return value from `assoc-equal`. Under this precondition, `cdr` is only applied to non-nil values from `assoc-equal` and we maintain correspondence of values for terms in G_{tcp} and G_{smt}.

By only providing `acons` and `assoc-equal`, the `Smtlink` support for alists is rather limited. Nevertheless, we have found it to be very useful when reasoning about problems where alists are used as simple dictionaries.

3.3.4 fty::defoption

As is shown in Program 3.3.4, the translation of `defoption` is straightforward.

Listing 3: ACL2 deflist

```
(defoption maybe-integer
           integerp)
```

Listing 4: Z3 Datatype

```
maybe_integer= z3.Datatype('maybe_integer')
maybe_integer.declare('some', ('val', IntSort()))
maybe_integer.declare('nil')
maybe_integer = maybe_integer.create()
```

In this example, the `maybe-integer-p` recognizer maps to `maybe_integer` type. The constructor `maybe-integer-some` maps to `maybe_integer.some`. The destructor `maybe-integer-some->val` maps to `maybe_integer.val`. The none type `nil` maps to `maybe_integer.nil`. Typical users of FTY types won't write `maybe-integer-some` constructor and `maybe-integer-some->val` destructors. They will first check if a term is nil, and then assume the term is an `integerp`. When a program returns an `integerp`, ACL2 knows it is also a `maybe-integerp`. Due to lack of type inference capabilities, `Smtlink` currently requires the user to use those constructors, therefore maintaining the option type through function calls where a `maybe-integerp` is needed and use the destructors where an `integerp` is needed. The constructor function satisfies the same theorems in ACL2 and Z3. Therefore, it's sound. For the field-accessor, when trying to access field of a `nil`, ACL2 returns the fixed default value, while Z3 will return arbitrary value of that `some` type. The Z3 values include the ACL2 value. `nil` is trivially the same. Thus, the `Smtlink` translation maintains equivalence of values of terms for constructors and field accessors of option types.

3.4 Uninterpreted Functions

The user can direct `Smtlink` to represent some functions as uninterpreted functions in the SMT theories. Let `(f arg1 arg2 ...argk)` be a function instance in G_{tcp}. `Smtlink` translates this to

```
    f_smt(arg_smt1, arg_smt2, ..., arg_smtk)
```

The constraints for `f_smt` are the types of the argument, the type of the result, and any user-specified constraints. If the function instance in G_{tcp} violates the argument type constraints, then the term is untranslatable – currently, `Smtlink` produces an SMT term that provokes a `z3types.Z3exception`. For each function instance in G_{tcp} that is translated to an uninterpreted function, `Smtlink` produces a proof obligation for ACL2 that the function instances in G_{tcp} satisfy the given type-recognizers. Likewise, if the user specified any other constraints for this function, they are returned as further ACL2 proof obligations. Under these preconditions, any value that can be produced by `f` satisfies the constraints for `f_smt`. Thus, we maintain correspondence of values for terms in G_{tcp} and G_{smt}.

4 A Ring Oscillator Example

In order to show the power of `Smtlink`, especially what benefit FTY types bring us, in this section, we take a look at how `Smtlink` can be used in proving an invariant of a small circuit. This example uses extensively the FTY types, including product types, list types, alist types, and option types. The simple circuit we want to model is a 3-stage ring oscillator. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring as is shown in Figure 3. A 3-stage ring oscillator consists of three inverters. It oscillates to provide a signal of a certain frequency.

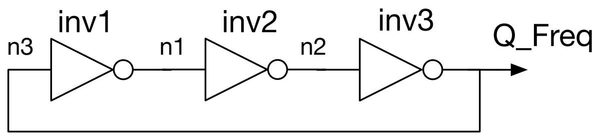


Figure 3: 3-Stage Ring Oscillator

For each inverter in this circuit, we say it is stable if its input is not equal to its output, otherwise, we say the inverter is ready-to-fire. One interesting invariant of this circuit is defined in Theorem 4.1:

Theorem 4.1 *Starting from a state where there is one and only one inverter ready-to-fire, for all future states, the ring oscillator will stay in a state where there is only one inverter ready-to-fire.*

In order to prove this property of this ring oscillator, we will discuss how we used FTY types extensively for circuit modeling and how `Smtlink` helped greatly at proving the theorem. To check the details of the proof, see projects/smtlink/examples/ringosc.lisp

4.1 Circuit and Trace Modeling using FTY Types

We model an **inverter** gate by defining a product type with two fields – `input` and `output`. We then model the 3-stage **ring oscillator** using a product type with six fields – three internal nodes `n1` through `n3` and three submodule inverters `inv1` through `inv3`. We call it `ringosc3-p`. We then define a **connection function** specifying the connections between the upper-level ring oscillator nodes and the lower-level inverter nodes. This describes the shape of the circuit.

As for modeling the behavior of the circuit, we use traces [7]. We define a circuit **state** as an alist mapping from signal names to its values. A **trace** is a list of circuits states, called `any-trace-p`. We define a **step recognizer** for an inverter. This recognizer function takes two consecutive steps from a trace as inputs and serves as a constraint function of what are the allowed behaviors in a step for an inverter. A valid trace for an inverter is defined recursively using the step function. A valid trace for the 3-stage ring oscillator is then defined requiring the trace to be valid for all three inverters. We call the recognizer function for a valid trace of a ring oscillator, `ringosc3-valid`.

We define a **counting** function for an inverter. The counting function returns 1 when an inverter is ready-to-fire and 0 when it's stable. Then we define a counting function for the ring oscillator based on the counting function for an inverter. We say a state of the ring oscillator is *one-safe* if only one inverter is ready-to-fire. We call the function `ringosc3-one-safe-state`. Using this function, we can define `ringosc3-one-safe-trace`, which means all states in a trace are *one-safe*. Theorem 4.1 is defined as Program 4.1.

Program 4.1 The ringosc3-one-safe theorem

```
(defthm ringosc3-one-safe
  (implies (and (ringosc3-p r)
                (any-trace-p tr)
                (consp tr)
                (ringosc3-valid r tr)
                (ringosc3-one-safe-state r (car tr)))
           (ringosc3-one-safe-trace r tr))
  :hints (("Goal"
           :induct (ringosc3-one-safe-trace r tr)
           :in-theory (e/d (ringosc3-one-safe-trace
                            ringosc3-valid
                            inverter-valid)
                           (ringosc3-one-safe-lemma)))
          ("Subgoal *1/1.1"
           :use ((:instance ringosc3-one-safe-lemma
                            (r r)
                            (tr tr)))
           )))
```

In this theorem, `Smtlink` helped to prove the inductive step. Due to space constraints, the details of `ringosc3-one-safe-lemma` are elided in this paper.. We note that proving this theorem using just ACL2 requires proving detailed lemmas about possible transitions of the ring oscillator. More specifically, our trace model requires asking if a signal exists in the state table, which causes a huge amount of case splits. However, this has not been a problem for `Smtlink` and the large amount of cases is handled efficiently. Using `Smtlink`, we are able to expand the functions out to proper steps and then prove the theorem without much human intervention. This demonstrates the potential of applying `Smtlink` to systems and proofs about systems. All this is made convenient because of the new theories supported and the useful user-interface.

5 Related Work

Integrating external procedures like SAT and SMT solvers into ACL2 has been done in several works in the past. Srinivasan [25] integrated the Yices [8] SMT solver into ACL2 for verifying bit-level pipelined machines. They use a trusted clause processor with a translation process. They appear to have mostly used the bit-vector arithmetic and SAT solving capabilities of Yices. Prior to that, in [17], they integrated a decision procedure called UCLID [16] into ACL2 to solve a similar problem. These are works that require fully trusting the integration.

A typical way of ensuring soundness and avoiding trusting external procedures is to followed Harrison and Théry's "skeptical" approach [12] and reconstruct proofs in the theorem prover. Recent work by Luís [13] allows refutation proofs of SAT problems to be reconstructed and replayed inside of ACL2. Their work focused on generating efficient refutation proofs that can be checked by a theorem prover in a short amount of time. Integrating SMT solvers into theorem provers has been a consistently developing area in the past decade [18, 11, 4, 19, 6, 5, 1, 10]. Erkök [10] integrated the SMT solver Yices into Isabelle/HOL. Similar to `Smtlink`, they not only have basic theories but also support algebraic datatypes. They trust Yices as an oracle. Works like [5, 9] do proof reconstruction. Sledgehammer [5, 14] is a proof assistant that integrates a bunch of SMT solvers into the theorem prover Isabelle/HOL. Proof reconstruction task is hard, and as pointed out in [14] the reconstruction can fail, and sometimes take a tremendous amount of time. Armand [1, 9] developed a framework in the theorem prover Coq for integrating external SMT solver results. They developed a set of "small checkers" that are able to take a certificate and call corresponding small checker for proof checks and used Coq tactics for automation. They report having achieved better performance than [5]. Also working on bridging the gap between interactive theorem proving and automated theorem proving (aka solvers), Moura [21] chooses a different path to build a theorem prover called Lean which also uses Z3, the SMT solver, as a back-end, but also provides benefits of an interactive theorem prover.

Several papers showed how their methods could be used for the verification of concurrent algorithms such as clock synchronization [11], and the Bakery and Memoir algorithms [19]. Erkök [10] uses the integration to prove memory safety of small C programs. [11] used the CVC-Lite [2] SMT solver to verify properties of simple quadratic inequalities. SMT solvers have drawn interests in the programming language research where [26] integrates SMT solvers into Haskell for proving refinement types.

Our work is based on our previous work [22]. Previously we showed how the integration of SMT solvers with theorem provers can help to prove properties of Analog/Mixed-Signal Designs. We used a single trusted clause-processor like is done in [25]. This paper describes our recent work of re-architecting `Smtlink` to verify the majority of it using verified clause-processors, therefore greatly improved soundness. We now depend on a very minimal core in a trusted clause-processor. In addition to that, our added support for algebraic datatypes largely broadened the horizon of problems `Smtlink` is able to handle.

6 Conclusion and Future Work

In this paper, we discussed an updated `Smtlink`. Comparing to the previous version, the current `Smtlink` has a more compelling argument of soundness, is extensible, and supports more theories. The architecture of `Smtlink` now clearly separates into verified and trusted parts. We make the trusted core as small as possible. We outlined an approach for verifying soundness, explaining how the gap is bridged between logic of ACL2 and the logic of an SMT solver. The new architecture through a sequence of verified clause-processors makes it extensible and easy to maintain. There are still many aspects we want to keep working on.

First, we want to use the meta-extract [15] capability introduced in last year's workshop. We believe if we can use the meta-extract to fully verify several of our clause-processors, for example, the function expansion clause processor, then the clause-processor won't need to return the clause back to ACL2 for proof. We observe that when projects get larger, the auxiliary clauses can become hard to prove, and making the clause-processor fully verified will reduce time spent on proving the auxiliary clauses to 0.

Second, given that we have the meta-extract capability, we are wondering if we can make a verified type inference engine. Currently, `Smtlink` knows nothing about types of terms and replies on the user for type instrumentation. Fixing functions have to be added to places where such a type information is required. For example, when dealing with a `nil`, which type of `nil` is it? We would love to deduce the types of terms and relieve the burden on the users for specifying types.

Third, we sketched a soundness proof in this paper, but this proof is not complete. Specifically we need to extend the model M_2' as described in Section 3 to a model that satisfies the real-closure axioms in a many-sorted setting. The current soundness proof sketch works well with ACL2(r) but not ACL2. We'd like to complete this proof.

Fourth, we want to make all counter-examples evaluable. This will require knowing FTY types and how to translate their constructors back. We also have to figure out how to translate algebraic numbers.

Fifth, we note `Satlink` has implemented a proof reconstruction interface that allows proofs to be returned from an SAT solver and replayed in ACL2. This can make the single trusted clause-processor goes away and remove all trusts we give to external SMT solvers. Proof reconstruction is an interesting direction that we might want to research more.

For applications, we believe the current `Smtlink` have enough capability that it can be applied to a lot of problems. We have been working on using it to verify properties of an asynchronous FIFO. The results are promising. In the future, we want to use it to prove timing properties making use of its arithmetic reasoning ability.

Acknowledgments

We would like to thank the ACL2 community for all the help in answering our questions while using ACL2 and developing `Smtlink`. We especially want to thank Matt Kaufmann for teaching us model theory and the many discussions to form a sketch of the soundness proof for `Smtlink`. We are also thankful to the anonymous reviewers for the insightful and constructive feedback.

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry & B. Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq Through Proof Witnesses*. In: *1st Int'l. Conf. Certified Programs and Proofs*, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[2] C. Barrett & S. Berezin (2004): *CVC Lite: A New Implementation of the Cooperating Validity Checker*. In: *Computer Aided Verification*, *LNCS* 3114, Springer, pp. 515–518, doi:10.1007/978-3-540-27813-9_49.

[3] C. Barrett, A. Stump & C. Tinelli (2010): *The SMT-LIB Standard: Version 2.0*. `http://www.cs.nyu.edu/~barrett/pubs/BST10.pdf`. [Online; accessed 17-August-2015].

[4] F. Besson (2007): *Fast Reflexive Arithmetic Tactics the Linear Case and Beyond*. In: *2006 Int'l. Conf. Types for Proofs and Programs*, Springer, pp. 48–62, doi:10.1007/978-3-540-74464-1_4.

[5] J.C. Blanchette, S. Böhme & L.C. Paulson (2013): *Extending Sledgehammer with SMT Solvers*. *J. Automated Reasoning* 51(1), pp. 109–128, doi:10.1007/s10817-013-9278-5.

[6] D. Déharbe, P. Fontaine, Y. Guyot & L. Voisin (2014): *Integrating SMT Solvers in Rodin*. *Sci. Comput. Program.* 94(P2), pp. 130–143, doi:10.1016/j.scico.2014.04.012.

[7] David L. Dill (1987): *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA. Available at `http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-88-119.pdf`. AAI8814716.

[8] B. Dutertre (2014): *Yices2.2*. In: *Computer Aided Verification*, *LNCS* 8559, Springer, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In Rupak Majumdar & Viktor Kunčak, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.

[10] Levent Erkök & John Matthews (2008): *Using Yices as an Automated Solver in Isabelle/HOL*. In: *In Automated Formal Methods08*, ACM Press, pp. 3–13. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.8123`.

[11] P. Fontaine, J.-Y. Marion, S. Merz, L.P. Nieto & A. Tiu (2006): *Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants*. In: *12th Int'l. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 167–181, doi:10.1007/11691372_11.

[12] J. Harrison & L. Théry (1998): *A Skeptic's Approach to Combining HOL and Maple*. *J. Automated Reasoning* 21(3), pp. 279–294, doi:10.1023/A:1006023127567.

[13] Marijn Heule, Warren Hunt, Matt Kaufmann & Nathan Wetzler (2017): *Efficient, Verified Checking of Propositional Proofs*. In Mauricio Ayala-Rincón & César A. Muñoz, editors: *Interactive Theorem Proving*, Springer International Publishing, Cham, pp. 269–284, doi:10.1007/978-3-319-66107-0_18.

[14] L. Paulson J. Blanchette (2017): *Sledgehammer*. `https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`. [Online; accessed 14-July-2018].

[15] Matt Kaufmann & Sol Swords (2017): *Meta-extract: Using Existing Facts in Meta-reasoning*. In Slobodová & Jr. [24], pp. 47–60, doi:10.4204/EPTCS.249.4. Available at `http://arxiv.org/abs/1705.00766`.

[16] S.K. Lahiri & S.A. Seshia (2004): *The UCLID Decision Procedure*. In: *Computer Aided Verification*, *LNCS* 3114, Springer, pp. 475–478, doi:10.1007/978-3-540-27813-9_40.

[17] P. Manolios & S.K. Srinivasan (2006): *A Framework for Verifying Bit-Level Pipelined Machines Based on Automated Deduction and Decision Procedures*. *J. of Automated Reasoning* 37(1-2), pp. 93–116, doi:10.1007/s10817-006-9035-0.

[18] S. Mclaughlin, Cl. Barrett & Y. Ge (2006): *Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite*. In: *In Proc. 3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, *ENTCS* 144(2), Elsevier, pp. 43–51, doi:10.1016/j.entcs.2005.12.005.

[19] S. Merz & H. Vanzetto (2012): *Automatic Verification of TLA$^+$; Proof Obligations with SMT Solvers*. In: *18th Int'l. Conf. Logic for Programming, Artificial Intelligence, and Reasoning*, Springer, pp. 289–303, doi:10.1007/978-3-642-28717-6_23.

[20] Leonardo Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer Berlin Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[21] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25*, Springer International Publishing, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[22] Yan Peng & Mark Greenstreet (2015): *Extending ACL2 with SMT Solvers*. In Matt Kaufmann & David L. Rager, editors: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications*, *Electronic Proceedings in Theoretical Computer Science* 192, Open Publishing Association, Austin, Texas, USA, 1–2 October 2015, pp. 61–77, doi:10.4204/EPTCS.192.6.

[23] Yan Peng & Mark Greenstreet (2015): *Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification*. In Klaus Havelund, Gerard Holzmann & Rajeev Joshi, editors: *NASA Formal Methods*, Springer International Publishing, pp. 310–326, doi:10.1007/978-3-319-17524-9_22.

[24] Anna Slobodová & Warren A. Hunt Jr., editors (2017): *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017*. *EPTCS* 249. Available at `http://arxiv.org/abs/1705.00766`.

[25] S.K. Srinivasan (2007): *Efficient Verification of Bit-level Pipelined Machines Using Refinement*. Ph.D. thesis, Georgia Institute of Technology. Available at `http://hdl.handle.net/1853/19815`.

[26] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler & Ranjit Jhala (2017): *Refinement Reflection: Complete Verification with SMT*. *Proc. ACM Program. Lang.* 2(POPL), pp. 53:1–53:31, doi:10.1145/3158141.