EPiC Series in Computing

Volume 99, 2024, Pages 7–14

Proceedings of the 7th and 8th Vampire Workshop

Integrating Answer Literals with AVATAR for Program Synthesis

Petra Hozzová

TU Wien, Vienna, Austria

Abstract

We recently introduced a framework for program synthesis based on functional specifications using a saturation-based theorem prover. To make our synthesis technique efficient, we need to incorporate it into the prover's architecture. In this paper, we describe the considerations of integrating our synthesis technique with the AVATAR splitting framework used in first-order theorem proving. We present an example that illustrates the issues accompanying the integration and describe our solution: constraining the splitting and adding an additional inference rule replacing certain clauses with AVATAR assertions by assertion-free clauses. Our experimental results indicate that the integration significantly improves the synthesis performance on some benchmarks.

1 Introduction

In [2], we present a new framework for program synthesis based on a functional specification using a saturation-based theorem prover. The main idea is to piggyback on theorem proving: search for a proof of the specification, and in the process construct a program satisfying the specification. To this end we modify the saturation algorithm such that it allows to track subsitutions into certain variables, record synthesized program fragmets, and after a proof is found, construct a final program from the recorded fragments. However, to make the most out of the saturation-based theorem prover, we need to integrate our synthesis framework with the prover's efficiency-critical features. Such integration allows us to prove more complex specifications and hence also synthesize more complex programs. In this submission, we explore integration of program synthesis with the AVATAR architecture for clause splitting [6] in the Vampire theorem prover [3].

2 Preliminaries

We briefly introduce saturation-based theorem proving and superposition. For details, we refer to [3]. We assume familiarity with standard multi-sorted first-order logic with equality. We denote variables by x, y, skolem constants by σ, clauses by C, formulas by F, all possibly with indices. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We include a conditional term constructor `if − then − else` in the language, as follows: given a

formula F and terms s, t of the same sort, we write $\texttt{if}\ F\ \texttt{then}\ s\ \texttt{else}\ t$ to denote the term s if F is true and t otherwise.

Saturation-based proof search implements proving by refutation: to prove that F is valid, it establishes the unsatisfiability of $\neg F$. To prove a formula F, saturation-based provers negate F, skolemize it, and convert to clausal normal form (CNF), denoted $\texttt{cnf}(\neg F)$, which forms the initial set S of clauses. This set is then *saturated* by computing logical consequences of S with respect to a sound inference system. If in the process $\square$ is derived, the original set S of clauses is unsatisfiable, and hence the formula F is valid. The most common inference system used by saturation-based provers for first-order logic with equality is the *superposition calculus* [4], including the superposition, binary resolution, equality resolution, equality factoring, and factoring rules. In this paper we use the binary resolution rule:

$$\frac{A \vee C \quad \neg A' \vee D}{(C \vee D)\theta} \text{ [Binary Resolution]}$$

where θ is the most general unifier of A and A' – i.e., it is a substitution such that $A\theta = A'\theta$ and from all such substitutions it is the most general one.

2.1 Saturation-Based Synthesis in a Nutshell

We give a simplified overview of our method from [2]. The method synthesizes programs for specifications given as formulas in first-order logic having the form

$$\forall \overline{x}.\exists y.F[\overline{x}, y], \tag{1}$$

where $F[\overline{x}, y]$ is a formula containing the variables $\overline{x}$ and y. The specification expresses that "for all inputs $\overline{x}$ there exists an output y such that the relation $F[\overline{x}, y]$ holds". Further, the specification identifies a set of symbols that are considered *computable*. Intuitively, the goal is to find a term r which is a witness for the variable y in (1), such that r only consists of computable symbols and variables $\overline{x}$.

To find such an r, we utilize *answer literals* [1] (already introduced to Vampire in [5]). Briefly, answer literals are used to track substitutions into the variable y throughout proof search. We add an answer literal $\texttt{ans}(y)$ to the negated and clausified specification, obtaining

$$\forall y.(\texttt{cnf}(\neg F[\overline{\sigma}, y]) \vee \texttt{ans}(y)),$$

where the skolem constants $\overline{\sigma}$ were introduced by skolemizing the variables $\overline{x}$, and then we saturate the resulting set of clauses. If we derive a clause $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$ where $C[\overline{\sigma}]$ is ground and computable and $r[\overline{\sigma}]$ is computable, then $r[\overline{x}]$ is a witness for y in (1) under the condition that $\neg C[\overline{x}]$ holds. We call such an $r[\overline{x}]$ a *program with conditions for* (1), and we denote it $\texttt{if}\ \neg C[\overline{x}]\ \texttt{then}\ r[\overline{x}]$. In particular, if $C[\overline{x}]$ is empty, then $r[\overline{x}]$ is an unconditional witness for y in (1) and we call it a *program for* (1).

Our framework utilizes saturation-based proof search to derive such clauses $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$. Upon derivation of such a clause, we record the corresponding program with condition and replace the clause in the search space by $C[\overline{\sigma}]$. Finally, when a proof of the given specification (1) is found, we construct a program for (1) by composing the programs with conditions recorded during the proof search using a (nested) $\texttt{if}-\texttt{then}-\texttt{else}$ construction. For more details on the synthesis framework, see [2].

2.2 Clause Splitting in a Nutshell

Clause splitting is one of the key features that make saturation-based theorem proving efficient in practice. The main idea of splitting is as follows. Let S be a set of clauses and $C_1 \vee C_2$ a clause such that C_1, C_2 have no common variables. We call such clauses C_1, C_2 the *components* of the clause $C_1 \vee C_2$. Then $S \cup \{C_1 \vee C_2\}$ is unsatisfiable iff both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. Therefore, instead of checking satisfiability of a set of large clauses, we can check the satisfiability of multiple sets of smaller clauses.

AVATAR [6] is the leading approach to clause splitting, used e.g. by the theorem prover Vampire [3]. It implements splitting by an interplay between a saturation-based first-order theorem prover and a SAT/SMT solver. The first-order prover passes clauses that can be split into multiple components to the SAT/SMT solver. The SAT/SMT solver then finds a set of clause components, satisfiability of which implies satisfiability of all split clauses – a *model*. The components in the model, called *assertions*, are then passed to the first-order prover to be used for further derivations in saturation. A clause C derived using assertions $C_1, \ldots, C_n$ is denoted as $C \leftarrow C_1, \ldots, C_n$ and is called a *clause with assertions* $C_1, \ldots, C_n$. When the first-order theorem prover derives the empty clause with assertions $C_1, \ldots, C_n$, it notifies the SAT/SMT solver of this and the SAT/SMT solver finds a new model of the split clauses such that not all $C_1, \ldots, C_n$ are true in the model. If there is no such model, it means that the whole problem is unsatisfiable, and the contradiction is propagated to the first-order theorem prover, which concludes the proof search.

3 Example without AVATAR

We illustrate the potential pitfalls of the use of answer literals in AVATAR by the following example from [5].

Example 1. The specification asks for a workshop, given the information which workshop happens on which day of a conference and an assertion that it is either Sunday or Monday:

$$\begin{aligned} \text{specification: } & \exists x.\mathsf{workshop}(x) \\ \text{axioms: } & \mathsf{sunday} \to \mathsf{workshop}(\mathsf{arcade}) \\ & \mathsf{monday} \to \mathsf{workshop}(\mathsf{vampire}) \\ & \mathsf{sunday} \vee \mathsf{monday} \end{aligned}$$

One possible program for this input would be:

$$\texttt{if}\ \mathsf{workshop}(\mathsf{arcade})\ \texttt{then}\ \mathsf{arcade}\ \texttt{else}\ \mathsf{vampire}$$

However, it is disputable if this program is helpful: if we ourselves could evaluate whether the condition $\mathsf{workshop}(\cdot)$ holds, arguably we would not need to pose the query $\exists x.\mathsf{workshop}(x)$ at all. Therefore, we annotate the symbol workshop as uncomputable, and thereby disallow its use in the target programs. With this annotation, we can straightforwardly synthesize a program for the input without using AVATAR:

(a)	$\neg\mathsf{workshop}(x) \vee \mathtt{ans}(x)$	[input specification with answer literal]
(b)	$\mathsf{sunday} \vee \mathsf{monday}$	[input]
(c)	$\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$	[input]

(d)	¬monday ∨ workshop(vampire)	[input]
(e)	¬sunday ∨ ans(arcade)	[binary resolution (a), (c)]
(f)	¬monday ∨ ans(vampire)	[binary resolution (a), (d)]

At this point our synthesis saturation algorithm records two programs with conditions:

```
if sunday then arcade
if monday then vampire
```

It also replaces (e) by (g), (f) by (h), and continues with saturation:

(g)	¬sunday	[replacement for (e)]
(h)	¬monday	[replacement for (f)]
(i)	monday	[binary resolution (b), (g)]
(j)	□	[binary resolution (h), (g)]

The final program is constructed by composing the programs recorded during the proof search:

```
if sunday then arcade else vampire
```

Note that we do not need to consider the condition `if monday`, because the proof was concluded by deriving □, and hence ¬sunday combined with the input clauses together implies monday. □

While this problem is simple and admits a short proof without splitting, it can be used to illustrate the issues with using AVATAR with answer literals. We however note that the problems that benefit most from the integration of AVATAR and synthesis are more complex, such as the maximum of n variables (for a sufficiently large given constant n):

$$\forall x_1, \ldots, x_n. \exists y. (\bigwedge_{i=1}^{i \leq n} y \geq x_i \wedge (\bigvee_{i=1}^{i \leq n} y = x_i)) \tag{2}$$

4 Path to Integration

In the process of integrating synthesis with AVATAR we had to deal with two main questions.

Q1. What would happen if we split clauses containing answer literals? We note that answer literals appear in all clauses only with positive polarity. Hence, if we split a clause containing answer literals such that an answer literal becomes a component and pass it to AVATAR, nothing prevents AVATAR from finding a model in which all answer literals are true. This model will satisfy all splittable clauses that contain answer literals. Thus, we might only find a proof by refutation if the input axioms without the negated specification were unsatisfiable, since the axioms correspond to the only input clauses without answer literals. This is clearly undesirable. Therefore, to avoid answer literals being true in the AVATAR model, we *disallow splitting of clauses that contain answer literals.*

To illustrate our second question, let us take another look at the problem from Example 1 using the answer for Q1 from the previous paragraph.

Example 2. We search for a proof of the problem from Example 1 using AVATAR but without splitting clauses containing answer literals:

(a')	$\neg\mathsf{workshop}(x) \vee \mathtt{ans}(x)$	[input specification with answer literal]
(b')	$\mathsf{sunday} \vee \mathsf{monday}$	[input]
(c')	$\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$	[input]
(d')	$\neg\mathsf{monday} \vee \mathsf{workshop}(\mathsf{vampire})$	[input]

AVATAR splits clauses (b')-(d'), denoting $\mathsf{sunday}, \mathsf{monday}, \mathsf{workshop}(\mathsf{arcade}), \mathsf{workshop}(\mathsf{vampire})$ by $1, 2, 3, 4$, respectively:

$$\text{(b')}: 1 \vee 2$$
$$\text{(c')}: \neg 1 \vee 3$$
$$\text{(d')}: \neg 2 \vee 4$$

Clauses (b')-(d') now do not participate in the first-order inferences anymore. AVATAR computes a model $\{1, 3, 4\}$ and introduces the component clauses:

(e')	$\mathsf{sunday} \leftarrow 1$	[component 1]
(f')	$\mathsf{workshop}(\mathsf{arcade}) \leftarrow 3$	[component 3]
(g')	$\mathsf{workshop}(\mathsf{vampire}) \leftarrow 4$	[component 4]

First-order reasoning continues:

(h')	$\mathtt{ans}(\mathsf{arcade}) \leftarrow 3$	[binary resolution (a'), (f')]
(i')	$\mathtt{ans}(\mathsf{vampire}) \leftarrow 4$	[binary resolution (a'), (g')]

At this point there are no more inferences that can be applied. Further, since the clauses containing answer literals (i.e., (h') and (i')) also have AVATAR assertions, they are not in the form $C \vee \mathtt{ans}(r)$, where C is ground and computable, and therefore we cannot directly apply the replacement steps we used in the AVATAR-less proof. This means that the proof attempt gets stuck, which leads us to the second question. □

Q2. What can we do with a clause derived using assertions and which also contains an answer literal? The assertions correspond to additional conditions that entail the clause. Hence, a natural way of converting a clause with assertions to a clause without assertions is to add negations of the assertions as literals. I.e., we can convert a clause $C \leftarrow A_1, \ldots, A_n$, where $A_1, \ldots, A_n$ are assertions, to the clause $C \vee \neg A_1 \vee \cdots \vee \neg A_n$.

Example 3. Let us try converting the clauses with assertions to assertion-free clauses to continue with our proof from Example 2:

(j')	$\mathtt{ans}(\mathsf{arcade}) \vee \neg\mathsf{workshop}(\mathsf{arcade})$	[reintroduce assertions of (h')]
(k')	$\mathtt{ans}(\mathsf{vampire}) \vee \neg\mathsf{workshop}(\mathsf{vampire})$	[reintroduce assertions of (i')]

The clauses (j') and (k') are also not in the form $C \vee \mathtt{ans}(r)$ where C is ground and computable, because the symbol $\mathsf{workshop}$ is uncomputable. Therefore, we cannot apply the recording and

replacement step. To make matters even worse, now the first-order reasoning continues by subsuming clauses (j') and (k') by the clause (a'). Then, there are once again no more first-order inferences that could be applied – we reached saturation with respect to the AVATAR model $\{1, 3, 4\}$, corresponding to $\{\mathsf{sunday}, \mathsf{workshop}(\mathsf{arcade}), \mathsf{workshop}(\mathsf{vampire})\}$. □

Our solution to preclude the situation where we add back assertions only to find that the resulting clause cannot be reasoned with further is to *disallow splitting clauses that are not ground and computable*. Then, all assertions are ground and computable, and thus if we derive a clause $C \vee \mathtt{ans}(r) \leftarrow A_1, \ldots, A_n$ where C is ground and computable, we can record a program based on it and remove the answer literal. Formally, we do this by *adding a new inference rule*

$$\frac{C \vee \mathtt{ans}(r) \leftarrow A_1, \ldots, A_n}{C \vee \neg A_1 \vee \cdots \vee \neg A_n \vee \mathtt{ans}(r)} \text{ [Reintroduce Assertions]},$$

which applies only if C is ground and computable. An application of this rule is always followed by recording of the program $\mathtt{if}\ \neg C \wedge A_1 \wedge \cdots \wedge A_n\ \mathtt{then}\ r$ and replacement of the clause $C \vee \neg A_1 \vee \cdots \vee \neg A_n \vee \mathtt{ans}(r)$ by $C \vee \neg A_1 \vee \cdots \vee \neg A_n$.

Example 4. Let us take a look at the proof of the problem from Example 1 using AVATAR with the constraints and the new rule as described above:

(a")	$\neg\mathsf{workshop}(x) \vee \mathtt{ans}(x)$	[input specification with answer literal]
(b")	$\mathsf{sunday} \vee \mathsf{monday}$	[input]
(c")	$\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$	[input]
(d")	$\neg\mathsf{monday} \vee \mathsf{workshop}(\mathsf{vampire})$	[input]

AVATAR splits the clause (b") (not clauses (c") and (d"), since workshop is not computable):

$$\text{(b")}\colon\ 1 \vee 2$$

Clause (b") now does not participate in the first-order inferences anymore. AVATAR computes a model $\{1\}$ and introduces the component clause:

(e")	$\mathsf{sunday} \leftarrow 1$	[component 1]

First-order reasoning continues:

(f")	$\mathsf{workshop}(\mathsf{arcade}) \leftarrow 1$	[binary resolution (e"), (c")]
(g")	$\mathtt{ans}(\mathsf{arcade}) \leftarrow 1$	[binary resolution (f"), (a")]
(h")	$\neg\mathsf{sunday} \vee \mathtt{ans}(\mathsf{arcade})$	[reintroduce assertions (g")]

At this point the algorithm records the program with condition:

```
if sunday then arcade
```

It also replaces (h") by (i") and continues with saturation:

(i")	$\neg\mathsf{sunday}$	[replacement for (h")]

(j")	$\square \leftarrow 1$	[binary resolution (e"), (i")]
(k")	$\neg 1$	[AVATAR contradiction (j")]

AVATAR recomputes a model $\{2\}$ and introduces the component clause:

(l")	$\mathsf{monday} \leftarrow 2$	[component 2]

First-order reasoning continues:

(m")	$\mathsf{workshop}(\mathsf{vampire}) \leftarrow 2$	[binary resolution (l"), (d")]
(n")	$\mathtt{ans}(\mathsf{vampire}) \leftarrow 2$	[binary resolution (m"), (a")]
(o")	$\neg\mathsf{monday} \vee \mathtt{ans}(\mathsf{vampire})$	[reintroduce assertions (n")]

At this point the algorithm records the program with condition:

```
if monday then vampire
```

It also replaces (o") by (p") and continues with saturation:

(p")	$\neg\mathsf{monday}$	[replacement for (o")]
(q")	$\square \leftarrow 2$	[binary resolution (l"), (p")]
(r")	$\neg 2$	[AVATAR contradiction (q")]

AVATAR tries to recompute the model, but detects that its input clauses are unsatisfiable, which concludes the proof:

(s")	$\square$	[AVATAR refutation (b"), (k"), (r")]

Finally, we construct the program from the programs with conditions collected during the proof analogously to how we did for the derivation without AVATAR:

```
if sunday then arcade else vampire
```

□

5 Conclusions

To integrate our synthesis framework with the AVATAR splitting framework, we add a constraint to only split clauses that are ground, computable, and contain no answer literals. Further, we introduce a new inference rule, which replaces certain clauses with assertions by clauses without assertions.

We acknowledge that the constraints could possibly be weakened – we leave this for future work. However, our experimental evaluation [2] witnesses that our approach already works well in practice: with AVATAR integration, we can synthesize the maximum function for specification (2) up to $n = 23$, while without the integration we can only synthesize the maximum up to $n = 9$ (all within a 5 minute time limit).

Acknowledgements. We thank Johannes Schoisswohl for proofreading. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, and the FWF grants LogiCS W1255-N23 and LOCOTES P 35787.

References

[1] Cordell Green. Theorem-Proving by Resolution as a Basis for Question-Answering Systems. *Machine Intelligence*, 4:183–205, 1969.

[2] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In *CADE*, pages 307–324, 2023.

[3] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[4] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasonings*, volume I, pages 371–443. Elsevier and MIT Press, 2001.

[5] Giles Reger. Revisiting question answering in vampire. *EPiC Series in Computing*, 53:64–74, 2018.

[6] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *CAV*, pages 696–710, 2014.