Master Project

Trace Abstraction with Accelerated Interpolants

Proposal

Jonas Werner

March 4, 2020

1 Introduction

Assume we want to verify whether a program is correct or not. A possible method is to construct the program's control-flow graph and using an automata-based instance of the CEGAR scheme, called trace abstraction, on it. Trace abstraction aims at constructing automata [6] from infeasible traces leading to an error location, so called error traces, such that the language of the union of these automata are a superset of the language recognized by the program's control-flow graph.
For each possible error trace check its feasibility, in case that the trace is feasible, the program is incorrect. If the trace is infeasible it is possible to compute a generalization that can exclude more error traces. A common strategy [5] to calculate a generalization is using Craig interpolation [3]. However these interpolants are not guaranteed to be the most general and with that not guaranteed to exclude many traces. This issue is most notably in program loops. Assume that the given program contains a loop with guard $x < 5000$, x being an integer variable. It is possible that trace abstraction cannot generate an interpolant that excludes every loop iteration, leading to the need of proving every of the 5000 possible error traces individually.
A possible solution for this problem would be accelerating the loop, meaning computing its transitive closure, and applying interpolation on that.
This project aims at implementing exactly that. The goal is to combine interpolation and loop acceleration on the basis of the work of Hojjat et al [7] in the software analysis framework Ultimate [1].
The remainder of this proposal is structured as follows, chapter 2 will give an overview over needed background information, like a more detailed look at trace abstraction, loop acceleration, and the combination of interpolation and acceleration. Chapter 3 will detail the approach this project will

take to implement accelerated interpolation in Ultimate, and finally an outline of the project's deliverables and schedule.

2 Background

This project aims at combining loop acceleration and interpolant calculation based on the findings of Hojjat et al [7], however, instead of utilizing a CEGAR-scheme with predicate abstraction, it will be implemented as an automata-based CEGAR-scheme, called trace abstraction.
This section will introduce the basic ideas behind trace abstraction, loop acceleration, and finally accelerated interpolants.

2.1 Interpolating Trace Abstraction

Given a program P and its control-flow graph $A_P = (Loc, \Delta, \ell_0, \ell_E)$, where Loc is a set of control states, Δ is a set of triples (ℓ, stm, ℓ') representing program transitions with $\ell, \ell' \in Loc$ and program statement stm, ℓ_{init} being the initial location, and ℓ_E being an error location. One can interpret the control-flow graph as an automaton with ℓ_E being an accepting state.
To check the reachability of ℓ_E, and with that correctness of P, use trace abstraction with interpolation according to the following paradigm [5]:

1. Get an error trace which starts at ℓ_0 and ends in ℓ_E:

   $$\ell_0 \xrightarrow{stm_1} \ell_1 \xrightarrow{stm_2} \cdots \xrightarrow{stm_{n-1}} \ell_{n-1} \xrightarrow{stm_n} \ell_E$$

   With each $(\ell_i, stm_i, \ell_{i+1}) \in \Delta$.

2. Construct an infeasibility proof.
   In case that the trace is proven feasible, the program is incorrect

3. Use the infeasibility proof to calculate interpolants.
   Construct an automaton, $\mathcal{A}_i$, from the interpolants.

4. If the language $\mathcal{L}(\mathcal{A}_\mathcal{P})$, that is recognized by the program's control-flow graph, is a subset of the union of languages recognized by the constructed automata: $\mathcal{L}(\mathcal{A}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup ... \cup \mathcal{L}(\mathcal{A}_i)$ then the program is correct, else start again at step 1.

The interpolants generated in step 3 serve to generalize the infeasibility proof to exclude other possible error traces. However, the interpolants are not guaranteed to be general enough to exclude infinitely many error traces. Which poses a problem for loops. In the following we introduce a way to exclude an infinite number of traces going through a loop.

2.2 Trace Schemes

To make the exclusion of error traces through a loop possible, we need to preprocess error traces. Assume we are given the following error trace τ:

$$\tau : \ell_0 \xrightarrow{stm_1} \ell_1 \xrightarrow{stm_2} \ell_2 \xrightarrow{stm_3} \ell_3 \xrightarrow{stm_4} \ell_2 \xrightarrow{stm_5} \ell_E$$

We see there is a loop from ℓ_2 to ℓ_3 and back to ℓ_2.

To be able to efficiently exclude the loop, calculate the composition of the loop's statements. This composition is then used to remove the loop from the trace, resulting in a trace scheme:

$$\tau' : \ell_0 \xrightarrow{stm_1} \ell_1 \xrightarrow{stm_2} \overset{stm_3 \circ stm_4}{\overset{\frown}{\ell_2}} \xrightarrow{stm_5} \ell_E$$

Assume the trace scheme is infeasible, it is possible to construct an interpolant sequence, that is, however, not guaranteed to be useful. This is where the loop acceleration is needed.

2.3 Loop Acceleration

The goal of acceleration is to compute a loop's transitive closure. In general, this is not possible as the transitive closure consists of infinitely many disjunctions. There are loops that can be expressed as so called octagon relations [8].
These relations have a certain constraint of the following form:

$$\pm x_i \pm x_j \leq c_{i,j}$$

With integer variables $x_i, x, c_{i,j}$
It has been shown, that the transitive closure of an octagonal relation can be efficiently computed [2].

2.4 Meta Trace Schemes

The transitive closure of a loop contains every trace going through it. To make use of a loop acceleration, we need to pull apart the looping location ℓ_2 by introducing so called meta-transitions of the form:

$$\overset{stm_3 \circ stm_4}{\overset{\frown}{\ell_2}} \quad \Rightarrow \quad \ell_2' \xrightarrow{(stm_3 \circ stm_4)^*} \ell_2''$$

Where $(stm_3 \circ stm_4)^*$ symbolizes the calculated transitive closure of the loop.
Using meta-transitions, we can transform our trace scheme into a meta-trace:

$$\bar{\tau} : \ell_0 \xrightarrow{stm_1} \ell_1 \xrightarrow{stm_2} \ell_2' \xrightarrow{(stm_3 \circ stm_4)^*} \ell_2'' \xrightarrow{stm_5} \ell_E$$

The feasibility of the meta-trace is the same as the trace scheme before. When we assume that it is infeasible, we can generate an interpolant sequence:

$$I_{\bar{\tau}} : \langle \top, I_1, I_2', I_2'', \bot \rangle$$

To guarantee inductiveness to the loop, we have to compute the strongest post of each interpolant, that coincide with a location in the loop, and its transitive closure:

$$I_{\bar{\tau}}^{post} : \langle \top, I_1, post(I_2', stm_3 \circ stm_4), I_2'', \bot \rangle$$

This sequence is now general enough to exclude infinitely many traces.

3 Approach

We want to implement accelerated interpolation into the software verification framework Ultimate [1]. Ultimate consists of multiple different libraries that can be executed in serial to form so called toolchains. There are already toolchains implementing trace abstraction as described above. This project extends these toolchains with accelerated interpolation by creating a new interpolating trace checking library.
This library will be used for step 3 of the trace abstraction paradigm. For that it has to fulfill the following specifications:

- Computing loop accelerations of octagonal relations using ultimately periodic relations as shown by Bozga et al [2].
- Construct trace schemes from given error traces and extend them using loop acceleration to meta schemes as presented by Hojjat et al [7].
- Check meta schemes' feasibility
- Construct interpolants for infeasible meta schemes to be used in trace abstraction.

This project is finished when this library has been implemented, tested, and evaluated by using it on several verification tasks.

Master projects award 12 ECTS points which translates to 12 weeks of work. Resulting in the following approach:

1. *Understanding the Matter:*
   Grasp how the combination of loop acceleration and interpolation works.
   Moreover, there have been previous projects that implemented the basis [9] and the ultimately periodic acceleration scheme itself [4] in Ultimate. It is important to understand those implementations before trying to adapt them for the usage of accelerated interpolants.

   Duration in weeks: 1

   Result: Being able to use the previous work and having an understanding on how to implement an accelerated interpolation library.

2. *Implement Trace Scheme Transformation:*
   The accelerated interpolation library should be able to transform a given error trace into a trace scheme.

   Duration in weeks: 1

   Result: Given an error trace, accelerated interpolation is able to transform it into a trace scheme.

3. *Implement Own Version of Loop Acceleration:*
   Implement a way of using Fast Acceleration of Ultimately Periodic Relations in the accelerated interpolation library.

Duration in weeks: 3

Result: Accelerated interpolation is able to construct the transitive closure of loops to use for interpolant generation

4. *Combining Trace Schemes and Loop Acceleration:*
   Accelerated interpolation can construct meta-schemes from trace schemes by inserting the transitive closure of the loop, prove its feasibility, and furthermore, is able to generate interpolants from the meta-scheme.

   Duration in weeks: 3

   Result: Accelerated interpolation returns a sequence of interpolants for use in the trace abstraction paradigm, or returns that the program is incorrect, depending on the feasibility of an error trace.

5. *Evaluating the Library:*
   Search for possible bugs and compare it to other trace checkers, such as PDR.

   Duration in weeks: 2

   Result: A bugfree accelerated interpolation library and data comparing it to other trace checkers.

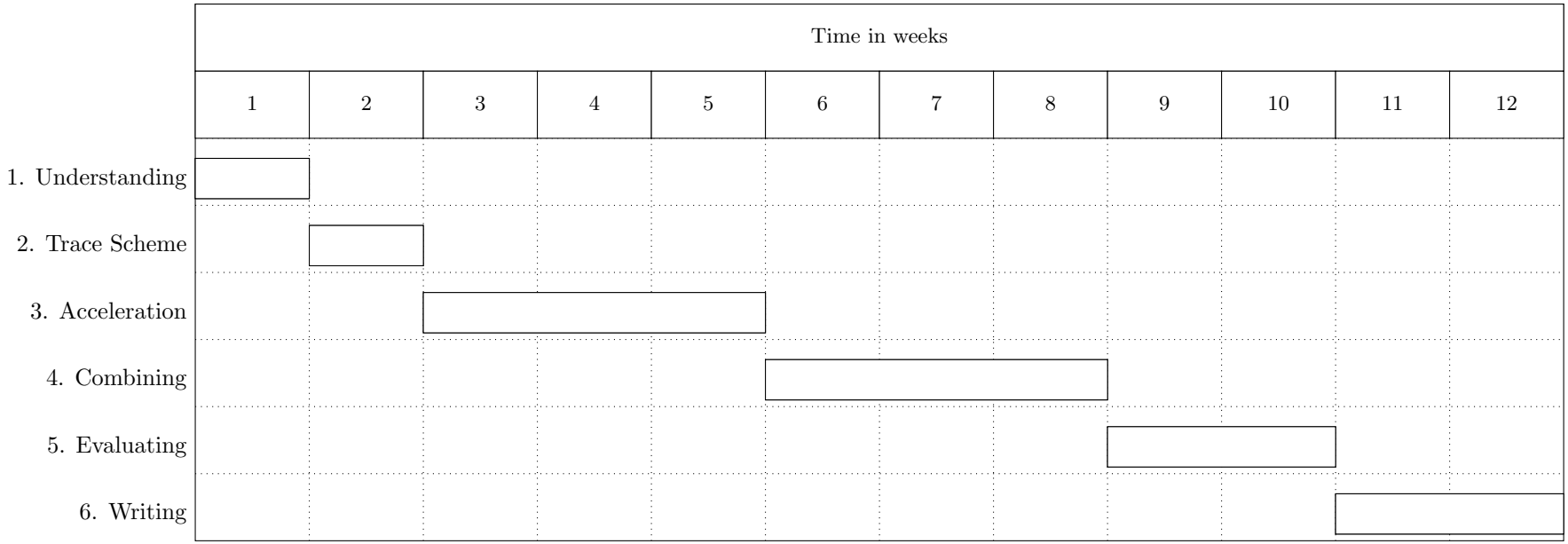
6. *Writing a Documentation about the Project:*
   Write in more detail than this proposal how accelerated interpolation works, and what new knowledge we discovered during this project

   Duration in weeks: 2

   Result: A written documentation about this project.

4 Schedule

This schedule illustrates the approach and shows the sequence of the individual tasks.



References

[1] Ultimate. https://ultimate.informatik.uni-freiburg.de. Accessed: 2020-02-14.

[2] Marius Bozga, Radu Iosif, and Filip Konečný. Fast acceleration of ultimately periodic relations. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 227–242, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[4] Jill Enke. Precise loop acceleration of ultimately periodic relations in ultimate. Bachelor's Thesis, 2017.

[5] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 69–85, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 36–52, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Hossein Hojjat, Radu Iosif, Filip Konečný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, pages 187–202, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] Antoine Miné. The octagon abstract domain. *CoRR*, abs/cs/0703084, 2007.

[9] Claus Schätzle. An octagon abstract domain for ultimate. Bachelor's Thesis, 2016.