Quanitified Underapproximation via Labeled Bunches (Artifact)

ANONYMOUS AUTHORS

The artifact is a Docker image contains the source OCaml code of a proof search program demonstrating the capability of LabelBI, which is the logical system developed in the main paper. This article discusses its scope, contents and methods of use.

ACM Reference Format:
Anonymous Authors. 2018. Quanitified Underapproximation via Labeled Bunches (Artifact). *J. ACM* 37, 4, Article 111 (August 2018), **??** pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The artifact is a Docker image with the program source code located at `/implementation`. The program is a proof search program that takes the system structure, a set of analyses on different subcomponents, and the goal representing the overall guarantee of the system as input, and then automatically applies LabelBI rules to build a proof tree showing the goal can or cannot be achieved. We list the directories in the project with a brief introduction.

- `ast`: The abstract syntax defined for LabelBI judgement, and the parser.
- `core`: The proof search program.
- `input`: The data structures used to represent the system structure and the analysis.
- `test`: Test files illustrating the capability of LabelBI.

We have three claims for the artifact functionality in Section 7.1 of the main paper, i.e.,

- **User-specified DAG and generation of the default behavior**. The feature is supported by the data structure `component` defined in `input/dag.ml`, and the function `compute_comp_formula` defined in the same file.
- **Bunched judgements for analyses**. The feature is supported by the abstract syntax definition in the `ast` directory, and the `build_judgement` function defined in `input/input.ml` which transform each analysis into a judgement.
- **Interactive proof search**. The feature is supported by the function `prove` defined in `core/analyzer.ml`.

In addition to that, We claimed that we have mechanized building the proof derivation for the two case studies, corresponding to the Fig. 10 in the main paper and Fig. 17 in the supplementary material, using the proof search program.

- **Fig. 10. Derivation generated for Ariane-5 case study**. The derivation can be reproduced using code at `test/ariane/analyzer_ariane.ml`.
- **Fig. 17. Derivation generated for iVotronic case study**. The derivation can be reproduced using code at `test/voting/analyzer_voting.ml`.

2 Hardware Dependencies

The artifact is developed and tested on an Apple M1 (aarch64) platform. It is also tested on a Windows X86-64 machine. Any reasonably modern hardware capable of running the specified

Author's Contact Information: Anonymous Authors.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Docker image below should be sufficient to run and continue developing this artifact. Regarding storage, approximately 2GB of free disk space is recommended.

3 Getting Started Guide

Installed Docker environment is the only prerequisite to evaluate the artifact. The proof assistant program depends on OCaml 4.14, menhir and dune, which have been properly setup in the docker image. Run the following command to get access to the source code directory.

```
# Pull image
docker pull langinteger/labelbi_prover:1.0.0
# Run the image as a new container with internal port 22 mapped to host machine port
    49123. An interactive terminal session will be launched
docker run -it -p 49123:22 --name labelbi_prover_container langinteger/labelbi_prover
    :1.0.0
# Change to the source code directory
cd ~/implementation
```

The container is also running a ssh server. You can use Visual Studio Code to connect to the container as a remote machine using SSH connection command `ssh opam@localhost -p 49123` and password `ocaml`. This step makes check and edit source code file easier but is optional.

4 Step by Step Instructions

4.1 Ariane-5 Case Study

With the directory changed to `~/implementation` in the interactive terminal launched above, the proof derivation for the Ariane-5 case study will be created by running the following command:

```
dune exec ./test/ariane/analyzer_ariane.exe
```

The derivation will be saved to `~/implementation/derivation.txt`, which contains the same content as in Fig. 10 of the main paper, except for minor formatting differences. A more readable and user-friendly proof tree, corresponding to Fig. 16 of the supplementary material is also saved to `~/implementation/result.html`.

4.2 iVotronics Case Study

Similarly, the command to reproduce the iVotronics case study is as follows:

```
dune exec ./test/voting/analyzer_voting.exe
```

The derivation and proof tree will be saved separately to the same location as Ariane-5 Case study, with each corresponding to Fig. 17 and Fig. 18, respectively.

4.3 Interactive Proof Search

As mentioned in section F.2 of the supplementary material, the termination of root-first proof search for LabelBI is not guaranteed. Thus, some rules, e.g., contraction rule, and downgrade rule are not considered in the automatic proof search. When the program is running in interactive mode, the user have the chance to input judgement to apply the rule manually. The two case studies above complete successfully without user intervention not because no such rules are involved, but because the predefined user hints are provided to the program as text files. Specifically, the files `~/implementation/test/ariane/proof_hint.txt` and `~/implementation/test/voting/proof_hint.txt` correspond to the Ariane-5 and iVotronics case studies, respectively.

The predefined user hint file consists of multiple groups of three lines. The first line specifies a judgement, the second indicates the rule to attempt, and the third provides the premise to use. The program will try the rule with the premise when the judgement specified in the first line cannot be proved after trying all the rules automatically. To complete the two case studies manually by interacting with the program, one can disable the predefined user hint feature, enable the interactive mode, and restart with the command `dune exec the-case-study-path`. We recorded two videos to demonstrate this process for Ariane-5 and iVotronics separately.

5 Reusability guide

6 Features

6.1 Plain text judgement parsing

We define the judgement to be of the form $\Sigma \mid \Gamma \Vdash A : B$, where Γ is a bunch, A is an analysis name[1], B is the guarantee of the analysis, and Σ is the set of test variables. For example, following judgement

$$\overbrace{\cdot}^{\Sigma} \mid \overbrace{\text{[output=A]ch1@MAX\_foo}}^{\Gamma} \text{|-} \overbrace{\text{goal}}^{A} : \overbrace{\text{[output=A]ch1@MAX\_foo}}^{B}$$

is used in the test case `./test/basics/reflexivity.ml`. We read it as: given the assumption, i.e., Γ, that the output channel `ch1` produces a value `A` for the given test level `MAX_foo`, B is guaranteed that the output channel `ch1` will produce a value `A` for the given test level `MAX_foo`. For the full abstract syntax of LabelBI judgement, please check the file `./ast/lexer.mll` and `./ast/parser.mly`.

By calling the API `parse_and_analyze` defined in the module `Test_main_entry.Test_main`, the passed-in plain text judgements, e.g., the above one, representing goal or premise will be parsed at first and then used for proof checker as discussed in next subsection.

6.2 Proof checker

The proof checker lies at the heart of the program. It tries to build the derivation tree for the goal based on a set of premises. As other sequent calculi without admissibility of contraction and cut, the termination of root-first proof search for LabelBI is not guaranteed. To the best of our knowledge, no one has yet proven the provability of bunched implication, the logic that our sequent calculus is based.

```
let prove (axioms:judgement list) (axioms_cut:judgement list) (goal:judgement) (
    level: int) (proof_node: int) : proof =
  let strategies = [
    ("by_sub", build_goal_by_sub);
    ("by_axiom", build_goal_by_ax);
    ("by_id", build_goal_by_id);
    ("by_cut", build_goal_by_cut);
    ("by_logic_left", build_goal_by_logic_left_rule);
    ("by_logic_right", build_goal_by_logic_right_rule);
    ("by_user", build_goal_by_user);
  ] in
  let rec try_strategies = function
    | [] -> Node (goal, NotExist, None, proof_node)
```

[1]The analysis name is solely used to refer to an individual analysis and does not have any computational meaning.

```
  | (name, strategy) :: rest ->
      let proof = build_goal name strategy axioms axioms_cut goal level
          proof_node in
      if is_proof_complete proof then
        proof
      else
        try_strategies rest
  in
  try_strategies strategies
```

Currently, we apply a brute-force back trace method to build the derivation tree. As shown in the code ??, the `strategies` defines the set of possible rules that the current goal built upon. The function `try_strategies` tries these rules with DFS algorithm. Following is the application of some rules explained in detail.

(1) by_logic $\wedge$R. If the judgement representing the current goal, again in the form of $\Sigma \mid \Gamma \Vdash A : B$, has a guarantee, i.e., the B, in the form of $B_1 \wedge B_2$, then the goal can be achieved by proving the two premises $\Sigma \mid \Gamma \Vdash A : B_1$ and $\Sigma \mid \Gamma \Vdash A : B_2$ recursively.

(2) by_logic $\otimes$R. If the judgement representing the current goal, again in the form of $\Sigma \mid \Gamma \Vdash A : B$, has a bunch, i.e., the Γ, in the form of Γ_1, Γ_2, and a guarantee, i.e., the B, in the form of $B_1 \otimes B_2$, then the goal can be achieved by proving either of the following two premise groups:
   (a) $\Sigma \mid \Gamma_1 \Vdash A : B_1$ and $\Sigma \mid \Gamma_1 \Vdash A : B_2$
   (b) $\Sigma \mid \Gamma_2 \Vdash A : B_1$ and $\Sigma \mid \Gamma_1 \Vdash A : B_2$

   This serves as an initial application of the Ex rule. For more intricate cases, the user may need to apply the Ex rule manually, as we will discuss next.

Other rules, e.g., the weakening rule, are not considered in the automatic proof search. Instead, if the goal cannot be achieved and the program is working in default interactive mode, the user can specify a manual strategy and provide plain text judgement to aid the proof. Possible manual strategies are:

(1) weakening: applying weakening rule
(2) contraction: applying contraction rule
(3) exchange: applying exchange rule
(4) upgrade: applying upgrade rule
(5) downgrade: applying downgrade rule
(6) continue_uninteractive: continue with uninteractive mode
(7) continue_interactive: continue with interactive mode

Finally, the proof checker will print the proof tree in Hilbert-style if the proof search is successful, otherwise it reports that the goal cannot be achieved.

For example, the test case `./test/basics/reflexivity.ml` is simple enough that the proof checker can automatically build the proof tree by applying the Id rule. Other cases, e.g., the `./test/basics/exchange_advanced.ml` needs user intervention to complete the proof. After initiating the test case by `dune exec ./test/basics/exchange_advanced.exe`, the program will stop with output:

```
Recall that the axioms are as follows:
  .|([output=1]ch1@MAX_{a},([output=2]ch2@MAX_{b},[output=3]ch3@MAX_{c}))|-goal:[
       output=3]ch3@MAX_{foo}
current goal fails after try all the possible rules: .|(([output=1]ch1@MAX\_{a},[
     output=2]ch2@MAX\_{b}),[output=3]ch3@MAX\_{c})|-goal:[output=3]ch3@MAX\_{foo}
```

```
Choose a command:
    - weakening: applying weakening rule
    - contraction: applying contraction rule
    - exchange: applying exchange rule
    - upgrade: applying upgrade rule
    - downgrade: applying downgrade rule
    - continue_uninteractive: continue with uninteractive mode
    - continue_interactive: continue with interactive mode
```

To continue, the user should send the exchange command and then provide the following plain text judgement:

·|([output=1]ch1@MAX_a,([output=2]ch2@MAX_b,[output=3]ch3@MAX_c))|-goal:[output=3]ch3@MAX_foo

6.3 Better user input

Instead of providing goal and premises by plain text judgement, the user can take advantage of the predefined data structures to define input, which will ease the workload expressing goal and premises in system analysis scenarios.
Such an input file consists of a `system` defined as a DAG, a set of `analyses` describing how each component has been evaluated, and a `goal` specifying the final behavior that we want to establish about the system.

System as a DAG. Following code defines the type for a system, where `inputs` and `outputs` represent the input and output channels for a component, respectively. The integrity of the system as a DAG will be ensured by the program, e.g., the absence of cycles.

```
type component = {
  name : component_name;
  resource: resource_name list;
  inputs : channel list;
  outputs : channel list;
}

type system = component list
```

Analysis. The definition of the analysis is as follows. An `analysis` consists of a group of assumptions and guarantees. Each `assumption` describes the expected behavior of an input channel while each `guarantee` specifies the behavior of an output channel. At present, an analysis can only contain one single component, the support for multiple components will be introduced in future iterations.

```
type analysis = {
  component: D.component;
  name: A.analysis_name;
  assumptions: predicate list;
  resource_tested: A.test_level;
  guarantees: predicate list;
}
```

Goal. We ask the user to specify the final goal with the same structure as an analysis.

6.4 Default behavior based on better user input

With the better user input mentioned in last subsection, we need to build a judgement for each analysis and goal. To achieve this, given the DAG, we associate a formula $\mathsf{db}(v)$ to each component v to describe its **d**efault **b**ehavior. The following algorithm constructs $\mathsf{db}(v)$ by induction on a topological ordering $\leq_T$ induced on the DAG:

Base case. For a component v with output channels $O_v = \{o_v^1, \ldots, o_v^n\}$ that does not have any input channels, we associate the formula $\mathsf{db}(v) := o_v^1 \wedge_{\mathsf{cm}} \ldots \wedge_{\mathsf{cm}} o_v^n$.

Inductive case. Consider a component v with output channels $O_v = \{o_v^1, \ldots, o_v^n\}$ and input channels $I_v = \{i_v^1, \ldots, i_v^m\}$. We associate the formula $\mathsf{db}(v) := \mathsf{idb}(I_v) \multimap_{\mathsf{cm}} (o_v^1 \wedge_{\mathsf{cm}} \ldots \wedge_{\mathsf{cm}} o_v^n)$, where $\mathsf{idb}(I_v)$ is a formula describing the behavior of the processes providing the input along channels I_v to v. We define $\mathsf{idb}(I)$ by induction on the size (m) of I as:

Base case ($m = 1$). We put $\mathsf{idb}(\{i\}) := i$,

Inductive case ($m > 1$) Consider the set $I' \subseteq I$ such that the DAGs providing channels $i \in I'$ are not mutually disjoint from each other but they all are disjoint from the rest of DAGs providing the channels in $I - I'$. For each $i \in I'$, we build the set S^i consisting of all the atomic components that are in the DAG providing another channel $i' \in I'$, for $i' \neq i$, but are not a part of the DAG provided by i. Put $S^i = \{s_1^i \cdots s_{k_i}^i\}$. We define the formula $\mathsf{idb}(I)$ as $\mathsf{idb}(I) := \bigwedge_{i \in I'} (i \bigotimes_{s^i \in S^i} T^{R_{s^i}}) \otimes f_{I-I'}$.

We generate the default behavior for each component in the system, by implementing the algorithm above as:

```
let rec compute_o (channels : channel list) : A.component =
  match channels with
  | [] -> raise EdgeCaseException
  | [s] -> OutputChannel s
  | h::t -> And (OutputChannel h, compute_o t)
and compute_i (channels : channel list) (sys : system) : A.component =
  match channels with
  | [] -> EmptyComponent
  | [s] -> OutputChannel s
  | h::d -> begin
    let grouped = group_by_ancestor channels sys in
    let parts = List.map (fun chs ->
      compute_i_group_formula chs sys
    ) grouped in
    combine_with_tensor parts
  end
let compute_f (sys : system) (comp : component) : A.component =
  let o_formula = compute_o comp.outputs in
  let i_formula = compute_i comp.inputs sys in
  match (comp.inputs, comp.outputs) with
  | ([], []) -> raise EdgeCaseException
  | ([], _) -> o_formula
  | (_, []) -> raise EdgeCaseException
  | _ -> Depend (i_formula, o_formula)
```

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.