Supporting Materials for Exploiting Symmetry in GR(1) Synthesis

ANONYMOUS AUTHOR(S)

This document includes supporting materials for a submission titled "Exploiting Symmetry in GR(1) Synthesis".

ACM Reference Format:
Anonymous Author(s). 2024. Supporting Materials for Exploiting Symmetry in GR(1) Synthesis. 1, 1 (September 2024), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 CORRECTNESS OF SYMMETRY-AWARE GR(1) ALGORITHM

We prove that in Alg. 2, which is the realizability checking algorithm with syntactic substitution from Section 5 in the paper ("Exploiting GR(1) Symmetry"), the syntactically substituted BDDs are equal to the corresponding BDDs had they been computed semantically. We first handle the blue code section.

1.1 Z Fixed-Point Syntactic Substitution Correctness

Given specification S, let σ be a GR(1) symmetry in S. Let j and k be two indexes of justice guarantees in the same permutation cycle of σ^s. Let J^s_j and J^s_k be the corresponding justice guarantees BDDs, such that $\sigma(J^s_j) = J^s_k$. Denote Z as the BDD in the current iteration of the outermost Z fixed-point computation. We first prove that the return value of $\textsc{Compute}(j, Z)$ (`mZ[j]` intermediate result) after applying σ is the same as $\textsc{Compute}(k, Z)$ (`mZ[k]` intermediate result).

Theorem 1. $\sigma(\textit{Compute}(j, Z)) = \textit{Compute}(k, Z)$

Observe that a permutation that is defined over a variable set is invariant under Boolean operations where both operands have support that is a subset of this set. This is the case in the GR(1) algorithm where all BDDs have support in $\mathcal{X} \cup \mathcal{Y}$, and σ is defined over this set. For every permutation σ, BDD f and BDD g where σ is a symmetry, we can write $f \wedge g = \sigma(f) \wedge \sigma(g) = \sigma(f \wedge g)$. Similarly with $\vee$.

We prove the following lemma concerning Z BDD value computed during the fixed-point iterations.

Lemma 1. *σ is a symmetry in Z BDD.*

Proof. We prove in induction on the number of fixed-point iterations. In the first iteration, Z_0 is initialized to true, hence symmetric. Otherwise, assume iteration k. Z_k is computed from ρ^s, ρ^e, the whole J^e set, the whole J^s set, and Z_{k-1} from a previous iteration. The elements that are parts of the specification S are symmetric by definition. Z_{k-1} is symmetric from the induction hypothesis. Z_k is thus computed from symmetric BDDs and therefore symmetric itself. □

We now prove the main theorem.

XXXX-XXXX/2024/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 1 GR(1) Realizability Checking Algorithm [1]

1: $Z \leftarrow$ **true**
2: **while** not reached fixed-point of Z **do**
3:     **for** $j = 0$ **to** n **do**
4:         $Y \leftarrow$ **false**; $r \leftarrow 0$
5:         **while** not reached fixed-point of Y **do**
6:             $start \leftarrow J_j^s \wedge \Diamond Z \vee \Diamond Y$
7:             $Y \leftarrow$ **false**
8:             **for** $i = 0$ **to** m **do**
9:                 $X \leftarrow Z$
10:                 **while** not reached fixed-point of X **do**
11:                     $X \leftarrow start \vee (\neg J_i^e \wedge \Diamond X)$
12:                 **end while**
13:                 $Y \leftarrow Y \vee X$
14:                 `X[`j`][`i`][`r`]` $\leftarrow X$
15:             **end for**
16:             `Y[`j`][`r`++]` $\leftarrow Y$
17:         **end while**
18:         $Z \leftarrow Y$
19:         `Z[`j`]` $\leftarrow Y$
20:     **end for**
21: **end while**
22: Return Z

Proof. Compute in the order agnostic GR(1) realizability checking variant is a procedure that can be described as a series of Boolean operations (logical and - $\wedge$, logical or - $\vee$, and logical negation - $\neg$) on BDDs (its code comprising of lines 3-19 in Alg. 1, also presented in the preliminaries section in the main paper). This computation depends on a BDD set that is comprised of parts of the specification S (ρ^s, ρ^e, and J^e set), the current justice guarantee J_j^s and Z. By definition, σ is a symmetry in ρ^s, ρ^e, the J^e set. From the lemma, it is a symmetry in Z. We can denote Compute in more detail as follows:

$$\textsc{Compute}(j, Z) = \textsc{Compute}(J_j^s, Z, \rho^s, \rho^e, J^e)$$

Since Compute does not depend on the justice guarantee order all the BDDs the procedure depends on are identical for all justice guarantee indexes, except J_j^s BDD. Now follows,

$$\begin{aligned}
\sigma(\textsc{Compute}(j, Z)) &= \sigma(\textsc{Compute}(J_j^s, Z, \rho^s, \rho^e, J^e)) \\
&= \textsc{Compute}(\sigma(J_j^s), \sigma(Z), \sigma(\rho^s), \sigma(\rho^e), \sigma(J^e)) \\
&= \textsc{Compute}(J_k^s, Z, \rho^s, \rho^e, J^e) \\
&= \textsc{Compute}(k, Z)
\end{aligned}$$

□

Observe that the computation of `mY[j][r]` intermediate results for all j and r in the same Z fixed-point iteration is done similarly to `mZ[j]`, so without loss of generality we can apply to it the same reasoning. The only difference in the proof is that `mY` array is not the return value of Compute, but rather computed as a side effect during the procedure execution. The general

Algorithm 2 GR(1) Realizability Checking Algorithm with Syntactic Substitution

procedure GR1Realizability
  $Z \leftarrow true$
  while not reached fixed-point of Z **do**
    $Z_{tmp} \leftarrow true$
    $computed \leftarrow$ initialize n-length array with false
    for $j = 0$ to n **do**
      if $computed[j]$ **then** continue
      $Z_{tmp} \leftarrow Z_{tmp} \wedge \text{Compute}(j, Z)$
      $computed[j] \leftarrow true$
      $j' \leftarrow j$
      while $\sigma^s(j') \mathrel{!}= j$ **do**
        $Z_{tmp} \leftarrow Z_{tmp} \wedge \text{Substitute}(j')$
        $computed[j'] \leftarrow true$
        $j' \leftarrow \sigma^s(j')$
      end while
    end for
    $Z \leftarrow Z_{tmp}$
  end while
  return Z
end procedure

procedure Substitute(j)
  $\mathtt{mZ}[\sigma^s(j)] \leftarrow substitute(\mathtt{mZ}[j], \sigma)$
  for $r = 0$ to $length(\mathtt{mY}[j])$ **do**
    $\mathtt{mY}[\sigma^s(j)][r] \leftarrow substitute(\mathtt{mY}[j][r], \sigma)$
    for $i = 0$ to m **do**
      $\mathtt{mX}[\sigma^s(j)][r][\sigma^e(i)] \leftarrow substitute(\mathtt{mX}[j][r][i], \sigma)$
    end for
  end for
  return $\mathtt{mZ}[\sigma^s(j)]$
end procedure

takeaway from the theorem is that we can compute the same BDD value by performing a fast syntactic substitution instead of a costly series of semantic operations.

1.2 X Fixed-Point Syntactic Substitution Correctness

X BDDs substitution is more tricky than Y and Z BDDs substitution because of the need to handle both i and j indexes of the justice assumption and the justice guarantee respectively.

We first illustrate the challenge with an example. Consider a simplified arbiter specification in Listing 1. It is easy to see that all the N request-grant pairs in the specification are pairwise-symmetric, and $\sigma^s = \sigma^e = (0 \ldots N-1)$. However, during the computation of X fixed-points, not all the BDDs are isomorphic to each other. Assume that we compute X while $j = 0$ during the first Y iteration, and consider the resulting fixed-points for $i = 0$ and for $i \neq 0$. By definition, every X is a safety fixed-point such that from every state in X, the system can move closer toward satisfying justice guarantee j or it can force the environment to violate justice assumption i. When $i = 0$,

```
env boolean[N] request;
sys boolean[N] grant;

// Eventually no request
asm eventualRelease{Int(0..(N-1)) i} :
    alwaysEventually !request[i];

// Only one grant at a time
gar mutualExclusion{Int(0..(N-1)) i, Int(0..(N-1)) j} :
    always (i != j) -> !(grant[i] & grant[j]);

// Eventually no request or grant
gar eventualGrant{Int(0..(N-1)) i} :
    alwaysEventually !request[i] | grant[i];
```

Listing 1. Simplified arbiter version

either $request[0] = true$ and J_0^e is violated, or $request[0] = false$ and J_0^s is satisfied. Hence X BDD for $i = j = 0$ is $true$. However, when $i \neq 0$, the state assignment such that $request[0] = true$, $grant[0] = false$, and $request[i] = false$ is not contained in X BDD because neither J_i^e is violated, nor J_0^s is satisfied.

Now consider computing X with $j = 1$ during the first Y iteration. It is X BDD for $i = j = 1$ that is $true$ now, while the rest of the X array with $i \neq 1$ has other values, symmetric to the computed X array for $j = 0$ and $i \neq 0$. We want to use the already computed memory for the first justice guarantee index $j = 0$ to perform syntactic substitution and get the memory for $j = \sigma^s(0) = 1$. We cannot just substitute X BDDs for $j = 0$ to get the corresponding BDDs for $j = 1$ with the same i index, but we need to take into account the permutation of the justice assumptions. In this particular case, for all Y fixed-point iterations r, we syntactically substitute `mX[0][r][0]` BDD to get the `mX[1][r][1]` BDD where $i = \sigma^e(0) = 1$. We similarly substitute `mX[0][r][1]` to get `X[1][r][2]` and so on, up to `mX[0][r][N-1]` to get the `mX[1][r][0]` BDD where $i = \sigma^e(N-1) = 0$. Note that the whole X array for $j = 0$ is already computed.

In general, to substitute X BDDs correctly in Substitute (lines 25-27 in Alg. 2), we take the already computed X BDD for justice guarantee j and justice assumption i, and substitute it using σ to get the X BDD for justice guarantee $\sigma^s(j)$ and justice assumption $\sigma^e(i)$.

We proceed with the correctness proof. Similarly to Compute we denote ComputeX that comprises of lines 9-14 in Alg. 1 as follows:

$$\textsc{ComputeX}(j, i, Z, Y) = \textsc{ComputeX}(J_j^s, J_i^e, Z, Y, \rho^s, \rho^e)$$

This procedure depends on Z, on Y that is computed every Y fixed-point iteration (lines 6-16), as well as on ρ^s and ρ^e from the specification S. σ is a symmetry in Y following similar reasoning from lemma 1. ComputeX does not depend on the whole justice assumptions set, but rather on a single justice each time. The return value of ComputeX is the intermediate result `mX[j][r][i]` for given j and i, and r that counts the Y iterations. Let i and l be two indexes of justice assumptions in the same permutation cycle of σ^e. Let J_i^e and J_l^e be the corresponding justice guarantees BDDs, such that $\sigma(J_i^e) = J_l^e$.

Very similarly to the previous proof, we show that:

Theorem 2. $\sigma(\textit{ComputeX}(j, i, Z, Y)) = \textit{ComputeX}(k, l, Z, Y)$

PROOF.

$$\begin{aligned}\sigma(\textsc{ComputeX}(j,i,Z,Y)) &= \sigma(\textsc{ComputeX}(J^s_j, J^e_i, Z, Y, \rho^s, \rho^e)) \\ &= \textsc{Compute}(\sigma(J^s_j), \sigma(J^e_i), \sigma(Z), \sigma(Y), \\ &\quad \sigma(\rho^s), \sigma(\rho^e)) \\ &= \textsc{Compute}(J^s_k, J^e_l, Z, Y, \rho^s, \rho^e) \\ &= \textsc{Compute}(k, l, Z, Y)\end{aligned}$$

□

This proof illustrates why we need to take into account both index permutations σ^e and σ^s when syntactically substituting the X BDDs in the purple code section.

2 CORES EVALUATION RESULTS

Specification	`Orig` >60 sec.	`Sym` / `Orig` Regr. Base <1	`SymD` / `Orig` Regr. Base <1	`Sym` / `Orig` Max Ratio <1	`SymD` / `Orig` Max Ratio <1	`Sym` / `Orig` Best 95% Avg. Ratio	`SymD` / `Orig` Best 95% Avg. Ratio
AMBA	66	**56**	28	**63**	43	**0.58**	0.70
Unreal. AMBA (extra justice g.)	37	35	**36**	13	**24**	**0.63**	0.66
Unreal. AMBA (extra safety g.)	5	**5**	4	0	**1**	0.80	**0.64**
Unreal. AMBA (missing justice a.)	66	**51**	49	**49**	39	**0.73**	**0.73**
GenBuf	49	**20**	17	**26**	20	**0.74**	0.76
Unreal. GenBuf (extra justice g.)	44	33	**38**	30	**38**	0.76	**0.69**
Unreal. GenBuf (extra safety g.)	45	**16**	**16**	1	**7**	>1	>1
Unreal. GenBuf (missing justice a.)	3	**2**	**2**	0	0	**0.91**	0.94
Abcg Arbiter	10	1	**8**	0	0	>1	>1
Full Arbiter	14	**13**	0	**14**	0	**0.48**	>1
Unreal. Full Arbiter	3	**3**	0	**3**	0	**0.46**	>1
Load Balancer	8	**8**	**8**	0	0	>1	>1
Unreal. Load Balancer	8	**7**	**7**	0	0	>1	>1
Prioritized Arbiter	15	3	**6**	5	**7**	>1	**0.57**
Unreal. Prioritized Arbiter	2	0	0	**1**	0	**0.63**	0.94
Round-Robin Arbiter	0	-	-	-	-	-	-
Unreal. Round-Robin Arbiter	3	**3**	2	**3**	2	**0.59**	0.63
Example Arbiter	5	0	**2**	**1**	0	0.94	**0.53**
Generalized Arbiter	5	**2**	0	**3**	2	**0.83**	0.91
Dining Philosophers	6	**6**	4	**5**	3	**0.39**	0.59

Table 1. Realizability checking results for the specifications in the corpus. >60 sec. column shows the number of mutants where at least one `Orig` measurement is greater than 60 seconds; Regr. Base<1 and Max Ratio<1 columns represent the number of mutants, out of the >60 ones, where the log-linear regression base was smaller than 1, and where the maximum ratio was smaller than 1, respectively; Avg. Ratio columns represent the average ratio across all measures of the >60 mutants. The better result is in **bold**.

	`Orig` Time-outs	`Sym` Time-outs	`SymD` Time-outs	`Sym` / `Orig` Regr. Base <1	`SymD` / `Orig` Regr. Base <1	`Sym` / `Orig` Max Ratio <1	`SymD` / `Orig` Max Ratio <1	`Sym` / `Orig` Best 95% Avg. Ratio	`SymD` / `Orig` Best 95% Avg. Ratio
Realizable	27.93%	**18.78%**	28.83%	**61.61%**	41.71%	**61.61%**	41.23%	**0.82**	0.88
Unrealizable	15.01%	**10.10%**	15.50%	**73.44%**	61.72%	**38.80%**	33.07%	0.86	**0.83**

Table 2. Realizability checking times for the specifications in the corpus divided by realizable and unrealizable instances. The data is presented in percentages and not in absolute numbers. The total number of realizable mutants and variants is 216 and 1478 respectively, and the total number of unrealizable mutants and variants is 385 and 2691 respectively. The timeout percentage is out of all variants. Other percentages are out of all mutants. The better result is in **bold**.

REFERENCES

[1] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.