A Sound and Complete Algorithm to Identify Independent Variables in a Reactive System Specification

Josu Oca
University of the Basque Country
San Sebastián, Spain
josu.oca@ehu.eus

Montserrat Hermo
University of the Basque Country
San Sebastián, Spain
montserrat.hermo@ehu.eus

Alexander Bolotov
University of Westminster
London, UK
A.Bolotov@westminster.ac.uk

***Abstract*—We present a sound and complete algorithm for the detection of independent variables in linear temporal logic formulae. These formulae are often used to specify reactive systems. The algorithm is based on the use of model checkers.**

***Index Terms*—linear temporal logic, reactive systems, independent variables, specification decomposition, A/G contracts.**

I. Introduction

Specifications of complex reactive systems (that interact with their environment [5], [6]) are often written in Linear Temporal Logic (LTL). It is important to have methods decomposing complex specifications into independent and simpler sub-specifications [1]–[3]. One strategy here is to identify a set of variables independent of each other [1], [3]. [1] proposed an algorithm for solving this problem based on the use of a model checker. This algorithm, we call it $\mathcal{DC}$, was developed in the context of Assume/Guarantee Contracts, which explicitly handles pairs of specifications, (S_1, S_2), where S_1 and S_2 represent, respectively, assumptions about the environment and the promises of the system under those assumptions [9].

Providing an example showing that $\mathcal{DC}$ (and its previous version presented in [4]) is incorrect and explaining why, we rectify this in a new algorithm, `Partition`: it adapts $\mathcal{DC}$ but also introduces a relevant change that allows us to prove its soundness and completeness. Finally, we remark that although `Partition` deals with LTL formulae, it can also be used with Assume/Guarantee Contracts.

II. Corrected partitioning algorithm

We assume the reader is familiar with the syntax and semantics of LTL [7]. For the purpose of specification, the set of variables in LTL formulae is divided into two disjoint subsets: I, the set of input variables controlled by the environment, and O, the set of output variables controlled by the system. Infinite traces over variables $I \cup O$ are denoted by $\sigma = \sigma_0, \sigma_1, \cdots, \sigma_j, \cdots$. For any $j \geq 0$, we have $\sigma_j = I_j \cup O_j$, where $I_j \subseteq I$ and $O_j \subseteq O$. When σ is a model of an LTL formula φ, we write $\sigma \models \varphi$. For example, consider $\varphi = \Box(p \rightarrow \bigcirc a)$, where $I = \{p\}$ and $O = \{a\}$. Let $\sigma = \{a\}, \{\}, \{p, a\}^\omega$ such that at state 0 the value of variable p (resp. a) is False (resp. True); at state 1, both p and a are False; and from state 2 onwards both p and a are True. Then $\sigma \models \varphi$. Given an LTL formula, we are interested in those sets of variables that operate independently of each other. This notion is captured by the following definitions (from Definitions 3 and 10 in [1], and Definition 4 and Corollary 1 in [8]).

Definition 1 (Projection formula):

Let $\varphi(I, O)$ be an LTL formula, such that $V \subseteq O = \{v_1, \ldots, v_{n_1}\}$ while $\overline{V} = (O \setminus V) = \{w_1, \ldots, w_{n_2}\}$. The projection formula of φ over V, denoted as φ_V, is the formula $\varphi(I, V, w'_1 \cdots w'_{n_2})$, where w'_j $(1 \leq j \leq n_2)$ is a fresh variable. Similarly, the projection formula of φ over $\overline{V}$, denoted as $\varphi_{\overline{V}}$, is the formula $\varphi(I, v'_1, \cdots v'_{n_1}, \overline{V})$, where v'_j $(1 \leq j \leq n_1)$ is a fresh variable.

Definition 2 (Independent Variables):

Let $\varphi(I, v_1, \cdots, v_{n_1}, w_1, \cdots, w_{n_2})$ be an LTL formula and let $V = \{v_1, \ldots, v_{n_1}\} \subseteq O$. The set V is independent in φ if and only if $((\varphi_V \wedge \varphi_{\overline{V}}) \rightarrow \varphi)$ is a valid formula, i.e, for any trace σ, if $\sigma \models (\varphi_V \wedge \varphi_{\overline{V}})$ then $\sigma \models \varphi$.

Algorithm 1: `Partition`

Input : I, O, φ
Output: Partition of O into sets of independent variables

clusters, R $\leftarrow \{\}, O$;
while $R \neq \emptyset$ **do**
  choose $p \in R$;
  V, passed $\leftarrow \{p\}$, False;
  repeat
    $\overline{V} \leftarrow O \setminus V$;
    passed, $\sigma \leftarrow$ `CheckValidity`$((\varphi_V \wedge \varphi_{\overline{V}}) \rightarrow \varphi)$;
    if *not passed* **then**
      $D \leftarrow$ `ParseTrace`(O, φ, V, σ);
      $V = V \cup D$;
  until *passed*;
  clusters $\leftarrow$ clusters $\cup V$;
  $R \leftarrow O \setminus (\bigcup_{S \in \text{clusters}} S)$
return clusters;

We present Algorithm 1 `Partition`, inspired by [1]. `Partition` divides O into sets that are independent in φ. We

This work is funded by the European Union (ERDF funds) under grant PID2020-112581GB-C22, European COST Action CA20111 EuroProofNet, and by the University of the Basque Country under project LoRea GIU21/044.

proved the following properties of any set V of the partition returned by the algorithm: 1.) V is independent in φ 2.) V is minimal, i.e. no proper subset of V, other than the empty set, is independent in φ. The former constitutes the soundness of the algorithm `Partition` and the latter shows its completeness. The proof of correctness, the logical meaning of both projection formulae and independent variables, and the whole formalisation can be found in [8].

`Partition` queries a model checker (line 7) about the validity of a formula. If the formula is not valid, the answer is a counterexample σ and then passed = False. Otherwise, σ = None and passed = True.

The running time of `Partition` is polynomial in the number of variables in O multiplied by the cost of each query made to the model checker.

Algorithm 2: `ParseTrace`

Input : O, φ, V, and the trace σ provided by `Partition`
Output: a variable that, for sure, depends on V

1 passed $\leftarrow False$;
2 constraints $\leftarrow False$;
3 **while** *not passed* **do**
4 $\quad Z = \{z \in \overline{V} : z \in \sigma_i \leftrightarrow z' \notin \sigma_i$ at some state $i \geq 0\}$;
5 $\quad z \leftarrow$ any variable in Z;
6 $\quad$ constraints $\leftarrow$ constraints $\vee \Diamond(z \leftrightarrow \neg z')$;
7 $\quad$ passed, $\sigma \leftarrow$ `CheckValidity`(constraints $\vee ((\varphi_V \wedge \varphi_{\overline{V}}) \rightarrow \varphi)$);
8 **return** $\{z\}$;

We explain Algorithm 1, `Partition`, with an example.

Example 1: Let $\varphi = \square((p \rightarrow \bigcirc(v \wedge \neg t)) \wedge (\neg p \rightarrow \bigcirc(\neg v \wedge t)) \wedge (v \rightarrow \bigcirc(\neg w \wedge u)) \wedge (\neg v \rightarrow \bigcirc(w \wedge \neg u)))$ where $I = \{p\}$ and $O = \{t, u, v, w\}$. `Partition` could start by selecting w in O (line 3) and asking a model checker (line 7) to check the validity of $\Phi = ((\varphi_{\{w\}} \wedge \varphi_{\{t,u,v\}}) \rightarrow \varphi)$. Assume the model checker returns $\sigma_1 = \{t, v'\}, \{t, t', w', u'\}, \{t, t', w, w'\}^\omega$, which is a model of $\neg\Phi$. At this point, we know that $\{w\}$ is dependent in φ and `Partition` calls to `ParseTrace` (line 9).

Now, the iteration of `ParseTrace` starts in line 4, calculating $Z = \{t, u, v\}$ and choosing a variable in Z (line 5). Assume `ParseTrace` takes t, constructs $\Diamond(\neg t \leftrightarrow t')$ (line 6), and checks the validity of the formula $\Diamond(\neg t \leftrightarrow t') \vee \Phi$ (line 7). The model checker could respond with $\sigma_2 = \{v'\}, \{t, t', w', u'\}, \{t, t', w, w'\}^\omega$ and a new iteration is made with $Z = \{u, v\}$. If `ParseTrace` chooses $v \in Z$, the model checker ensures that $\Diamond(\neg t \leftrightarrow t') \vee \Diamond(\neg v \leftrightarrow v') \vee \Phi$ is valid. Then, the iteration stops and `ParseTrace` returns $\{v\}$ to `Partition`, which adds v to V (line 10).

Now, $V = \{v, w\}$ and the iteration of `Partition` goes on checking the validity of $\Phi = ((\varphi_{\{v,w\}} \wedge \varphi_{\{t,u\}}) \rightarrow \varphi)$ (line 7). Suppose the model checker responds again with the counterexample σ_2. `ParseTrace` is called and $Z = \{u\}$. In this case, $\Diamond(\neg u \leftrightarrow u') \vee \Phi$ is valid and `ParseTrace` returns $\{u\}$. `Partition` adds u to V and now, $((\varphi_{\{u,v,w\}} \wedge \varphi_{\{t\}}) \rightarrow \varphi)$ is valid. Thus, the set $V = \{u, w, z\}$ is included in clusters (line 12). At the end, `Partition` finishes dividing the variables of O into two clusters: $\{u, v, w\}$ and $\{t\}$.

III. Application of the algorithm in LTL Assume/Guarantee Contracts

Formally, a contract is a tuple $C = (I, O, A, G)$, where both A and G are formulae over $I \cup O$. C is said to be an LTL A/G contract when A and G are LTL formulae. The algorithm $\mathcal{DC}$ proposed in [1] (Algorithm 12) decomposes an LTL A/G contract received as input, into several subcontracts, treated independently to simplify the synthesis problem.

`Partition` is like $\mathcal{DC}$ except for two points: `Check Validity` and `ParseTrace`. In fact, the difference in `CheckValidity` is in the formula with which the model checker is consulted, and is not significant. $\mathcal{DC}$ uses projection formulae and Definition 2 to construct two queries: one for the assume A and the other for the guarantee G. The difference with respect to the second point is much more important. `ParseTrace` function in $\mathcal{DC}$ examines a counterexample provided by the model checker and simply returns the set of all variables that behave differently. Formally, $\texttt{ParseTrace}_{\mathcal{DC}}(\sigma) = \{z \in O : z \in \sigma_i \leftrightarrow z' \notin \sigma_i$ at some state $i \geq 0\}$. Our `ParseTrace` function is more complicated and returns a single variable only when we are sure that there is a dependency between the variable and the set V.

The execution of $\mathcal{DC}$ on contract $(I, O, true, \varphi)$, where I, O, and φ correspond to Example 1, could return (depending on the counterexamples provided by the model checker) a single cluster formed by $\{t, u, v, w\}$. However, $\{t\}$ and $\{u, v, w\}$ are independent in φ. While $\mathcal{DC}$ is not complete, replacing the $\texttt{ParseTrace}_{\mathcal{DC}}$ function in $\mathcal{DC}$ with $\texttt{ParseTrace}_{\texttt{Partition}}$ makes $\mathcal{DC}$ sound and complete. Moreover, this change does not over-compromise the efficiency of $\mathcal{DC}$.

$\mathcal{DC}$ has been tested with good performance for small synthesis tasks, allowing designers to manage complexity and synthesise designs that are not feasible with other techniques [1]. We expect our method to benefit the LTL-based Assume/Guarantee Contracts community, where $\mathcal{DC}$ is used.

References

[1] A. Iannopollo, "A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications," UC Berkeley, 2018.
[2] S. Bansal, G. De Giacomo, A. Di Stasio, Y. Li, M.Y. Vardi and S. Zhu, "Compositional Safety LTL Synthesis," Verified Software. Theories, Tools and Experiments, Springer-Verlag, pp. 1–19, 2023.
[3] B. Finkbeiner, G. Geier and N. Passing, "Specification Decomposition for Reactive Synthesis," Innovations in Systems and Software Engineering 19, pp. 339–357, 2023.
[4] A. Iannopollo, S. Tripakis, and A. Vincentelli, "Specification decomposition for synthesis from libraries of LTL Assume/Guarantee Contracts," Design, Automation & Test in Europe Conference & Exhibition DATE, pp. 1574-1579, 2018
[5] A. Pnueli and R. Rosner, "On the synthesis of an asynchronous reactive module," Proceedings of ICALP, LNCS Springer, pp. 652–671, 1989.
[6] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," Proceedings of POPL, ACM, pp.179–190, 1989.
[7] M. Ben-Ari, "Mathematical Logic for Computer Science (Third edition)," Springer, 2012.
[8] J. Oca, M. Hermo, and A. Bolotov, "Revisiting the specification decomposition for synthesis based on LTL solvers," arXiv, 2023.
[9] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," Proceedings of FMCO. LNCS Springer, vol 5382, pp. 200–225, 2008.