Control-flow Guided Property Directed Reachability for Imperative Synchronous Programs

Xian Li and Klaus Schneider
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—Property directed reachability (PDR) has been introduced as a very efficient verification method for synchronous hardware circuits that is based on induction rather than fixpoint iteration. However, hardware circuits are usually synthesized from more abstract high-level languages like synchronous languages (or synchronous subsets of hardware description languages). In this paper, we show that it is possible to derive from such high-level languages additional control-flow information that can be added to the transition relation to make PDR even more efficient. As will be shown, PDR can benefit from this additional information since many safety properties become inductive only with respect to the enhanced transition relations. The added control-flow information is not needed for the synthesis and is therefore not explicitly encoded in the generated systems, but it can be easily derived from the original programs and used for verification. We present two methods to compute additional control-flow information that differ in how precisely they approximate the reachable control-flow states and also in the runtime required for their computation.

I. INTRODUCTION

Property directed reachability (PDR) [9]–[11], [18], [19], [21], [22], [37] is currently considered to be the most efficient verification method for safety properties. The core algorithm has been introduced in [9] for hardware model checking problems and has been implemented in a tool called IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness). In essence, given a symbolic representation of a state transition system $\mathcal{K}$ and a state property $\Phi$, the algorithm tries to prove that $\Phi$ holds on all reachable states of $\mathcal{K}$ by means of induction. To this end, it first checks whether $\Phi$ holds on all initial states (induction base), and then checks whether $\Phi$ holds on all successor states of those states that satisfy $\Phi$ (induction step). However, the latter may fail even though $\Phi$ holds on all reachable states since there may exist unreachable states satisfying $\Phi$ that have successor states that do not satisfy $\Phi$. Such states are called counterexamples to induction (CTI) and have to be incrementally learned and excluded from consideration by the PDR method.

PDR is very efficient, since in the best case, it may just use a SAT or SMT solver to prove the induction base and induction step. To this end, it has no need to compute fixpoints as done by symbolic model checking [2], [3], neither to unroll the transition relation as required in bounded model checking [1], [12] nor has it a need to construct Craig interpolants as required by interpolation-based model checking [26]. Instead, it maintains a sequence of predicates $\Psi_0, \ldots, \Psi_k$ that include the state sets $X_0, \ldots, X_k$ that are reachable in no more than $0, \ldots, k$ steps, respectively. If the induction proof fails, PDR either increments this sequence with a new predicate $\Psi_{k+1}$ or improves the approximations of the predicates $\Psi_i$ by removing CTIs. The latter is done by first checking the unreachability of the CTI and then adding a most general inductive clause to the predicates to remove the CTI together with as many further CTIs as possible within one step. These steps are repeatedly applied until either finally a reachable counterexample is found or one of the predicates $\Psi_i$ becomes inductive.1

In the worst case, PDR may have to compute the state sets that are also computed by a forward fixpoint computation of the reachable states. In the best case, it may just check one induction base and step. Usually, it has to construct some predicates $\Psi_0, \ldots, \Psi_k$ until the induction proof works, but that work is typically less compared with the fixpoint iteration – both the number of predicates $k$ can be less than the number of iteration steps in the fixpoint computation and also the work done within one iteration step can be less.

PDR has meanwhile been integrated in many model checkers like nuXmv 2, ABC [7], IIMV 3, PuTRAV 4, and Kind2 5, and many detailed optimizations have been added: In particular, [18] suggested a couple of modifications to the original PDR method to improve its performance and to simplify its implementation. An algorithm using SAT solvers instead of ternary simulation was proposed in [14] to determine most general clauses from the CTIs. A lazy abstraction-refinement technique has been combined with the PDR method for large industrial hardware designs in [38]. An improved clause generalization procedure to explore states farther than the counterexamples has been considered in [22]. Finally, [19] implemented different variants of PDR for hardware model checking in nuXmv, and conducted a systematic evaluation using the benchmarks of the latest hardware model checking competition.

The original PDR method was introduced for hardware model checking, and therefore operates directly on proposi-

1 This must finally happen since those predicates $\Psi_i$ that cover the reachable states $S_{\text{each}}$ will converge to $S_{\text{each}}$ due to improving the approximations, and $S_{\text{each}}$ is an inductive set.
2 http://nuxmv.fbk.eu
3 http://ccee.colorado.edu/~bradleya
4 http://fmgroup.polito.it
5 http://kind2-mc.github.io/kind2
tional formulas $Ψ_I$ and $Ψ_R$ that encode the initial states and the transition relation of a considered hardware circuit with finitely many states. It is however clear that PDR can handle also infinite state systems \cite{17, 23} by replacing SAT solvers with SMT solvers, so that PDR can be also applied to software model checking.

Besides the use of non-boolean data types, another distinction between hardware and software model checking is the consideration of the syntax of the programs, i.e., the control-flow of the programs. One of the first attempts in that direction was presented with the TREE-IC3 method in \cite{17} which unrolls the control-flow graph (CFG) to an abstract reachability tree. An adaptation of the TREE-IC3 algorithm is used in \cite{36} to synthesize controllers for discrete-event systems modeled by extended finite state machines (EFSMs). Also, \cite{40} presented a software verification algorithm based on the refinement of loop invariants using a generalization of PDR to the theory of quantifier-free formulas over bitvectors \cite{39}. In \cite{24}, the relative inductive reasoning is performed over regions that are defined by symbolic representations with respect to certain locations of a control-flow automaton.

Hence, PDR has already been optimized in many ways. In particular, it has been integrated with SMT solvers to deal with higher data types and it has been enriched with the use of loop invariants and control-flow graphs for software model checking. However, these refinements are not just useful for software model checking since hardware circuits are usually synthesized from more abstract high-level languages like synchronous programs or synchronous subsets of hardware description languages. In these programs, the synchronous model of computation reflects the executions of hardware circuits: At each tick of the clock, new input values are read, internal states are updated, and outputs are immediately computed. Similar to traditional imperative programming languages, imperative synchronous programming languages like Esterel \cite{4}, \cite{6} and Quartz \cite{35} also have statements like conditionals, sequences, various kinds of loops, etc. Thus, one can also derive control-flow information from these kinds of hardware descriptions that is only implicitly available in the generated synchronous circuits. As we will show, this additional control-flow information can significantly help to verify safety properties of synchronous programs by means of PDR.

For formal verification, synchronous programs $S$ are usually translated to equivalent state transition systems that are symbolically represented by means of formulas $Ψ_I$ and $Ψ_R$ that encode the initial states and the transition relation of the underlying state transition system. The same or similar transition systems can also be used for code generation \cite{6}, \cite{29}, \cite{31}, \cite{35}. Ignoring the use of higher data types, at this stage PDR can be applied without any changes. However, as we will show in this paper, it is beneficial to enhance the usual transition relation $Ψ_R$ by additional control-flow information that contains invariants about the reachable (control-flow) states. In particular, it is added that no state is reachable where the control would be active in both substatements of an if-statement $\text{if}(\sigma) S_1 \text{ else } S_2$ or a sequence $S_1; S_2$. By the semantics of the programs, this can also be proven with the original formulas, and in particular, it can also be verified by PDR or any other suitable verification method. However, this requires the computation of the reachable states that is actually what PDR wants to avoid as much as possible.

In this paper, we therefore show that it is straightforward to compute a control-flow invariant $\text{InvarCF}(S)$ for any synchronous program $S$ even in time linear in the size of the program. We then apply PDR to the enhanced transition relation $Ψ_R := Ψ_R \land \text{InvarCF}(S)$ where some transitions of unreachable states are removed. As will be shown, there are safety properties that become inductive only with the enhanced formulas, so that PDR can prove them in one step, while otherwise, PDR would have to apply arbitrarily many incremental steps. Actually, this control-flow invariant $\text{InvarCF}(S)$ was already computed by the compiler of our Averest\footnote{Arbitrary means here that based on parameters, we can increase the number of additionally required steps beyond every bound.} framework.

Since this control-flow invariant just states the disjointness of substatements of if-statements $\text{if}(\sigma) S_1 \text{ else } S_2$ and sequences $S_1; S_2$, it is a coarse over-approximation of the reachable (control-flow) states. As a second method, we therefore only consider the part of the control-flow $Ψ^c_I$ and $Ψ^c_R$ that is contained in $Ψ_I$ and $Ψ_R$ and compute its reachable states with a symbolic state space traversal. This way, we obtain with the reachable states $\text{ReachCF}(S)$ of the control-flow automaton another over-approximation of the reachable control-flow states that is more precise than $\text{InvarCF}(S)$, but still abstracts from the data flow and just considers the control-flow. PDR can greatly benefit from adding control-flow information given by either $\text{InvarCF}(S)$ or $\text{ReachCF}(S)$ since many safety properties become inductive only with respect to the enhanced transition relation.

To the best of our knowledge, there is not much directly related work: One of the first attempts to check synchronous programs by the PDR method is presented in \cite{13}. However, the main topic of \cite{13} was the integration of PDR with predicate abstraction that was demonstrated with synchronous Lustre programs \cite{20}. The latter is however a data flow synchronous language without a control-flow and is also the input language of the Kind2 model checker \cite{15}, \cite{16} that focuses on invariant generation and compositional reasoning. Closer to the work presented in this paper is \cite{24}, where additional control-flow information similar to ours has been added for sequential programs. However, the method proposed in \cite{24} is limited to sequential programs, concurrent programs like synchronous programs were not considered there at all.

For our experiments, we consider our Esterel-like language Quartz \cite{35}, and its translation to guarded actions \cite{8}, \cite{30}, \cite{35} as intermediate representation, and the final construction of symbolic representations from the latter. In a previous paper \cite{25}, we used the control-flow information of imperative synchronous programs to reason about the unreachability of counterexamples and to generalize them to all states with

\footnote{\url{http://www.averest.org}}
the same control-flow states. The transition relations used in this paper will prevent the generation of those CTIs whose unreachable could be checked quickly with the method described in [25], but the generalization of clauses as described there is still useful.

The outline of the paper is as follows: In the next section, we describe the PDR method in a nutshell. Section III introduces the basics of the synchronous language Quartz, its translation to guarded actions, and the construction of symbolic representations of state transition systems. The core of the paper is contained in Section IV which explains the computation of the control-flow information InvarCF(S) and ReachCF(S) for any synchronous Quartz program S. We demonstrate the enhanced method with three typical Quartz programs in Section V that show that the added control-flow information can save arbitrarily many incrementation steps of PDR. The paper will be concluded in Section VI.

II. PDR IN A NUTSHELL

In this section, we give a brief introduction to PDR as far as required for this paper. We thereby restrict our consideration to boolean programs, but note that our results are not restricted this way. In the next Section II-A, we first list some preliminaries about symbolic model checking and focus then on PDR in Section II-B.

A. Symbolic Model Checking

To start with, we assume that a state transition system $K = (V, \Psi_I, \Psi_R)$ is symbolically represented by means of a finite set of boolean variables $V$, and propositional formulas $\Psi_I$ and $\Psi_R$ for its initial states and its transitions, respectively. A state $s \subseteq V$ of $K$ is a subset of $V$ such that those variables hold in the state that belong to $s$ while others are false. As usual, every propositional formula $\phi$ over the variables $V$ is associated with a set of states $[\phi]_K \subseteq 2^V$ of the transition system which are those states that satisfy $\phi$ if the states are viewed as variable assignments. Analogously, every propositional formula over the variables $V$ and a related set $V'$ denotes a set of transitions so that the assignments to variables $V$ and $V'$ correspond with the current and next state, respectively.

To reason about temporal relationships of states, we define the existential/universal predecessor/successor states of a set $Q$ as follows:

- $\text{pre}_\exists(Q_2) := \{ s_1 \in S_1 \mid \exists s_2,(s_1,s_2) \in R \land s_2 \in Q_2 \}$
- $\text{pre}_\forall(Q_2) := \{ s_1 \in S_1 \mid \forall s_2,(s_1,s_2) \in R \to s_2 \in Q_2 \}$
- $\text{succ}_\exists(Q_1) := \{ s_2 \in S_2 \mid \exists s_1,(s_1,s_2) \in R \land s_1 \in Q_1 \}$
- $\text{succ}_\forall(Q_1) := \{ s_2 \in S_2 \mid \forall s_1,(s_1,s_2) \in R \to s_1 \in Q_1 \}$

While the above definitions are given in terms of sets of states, their counterparts in the syntax of formulas are the following modal operators taken from $\mu$-calculus [34]:

- $[\Box \phi]_K := \text{pre}_\forall([\phi]_K)$
- $[\Diamond \phi]_K := \text{pre}_\exists([\phi]_K)$
- $[\Box \phi]_K := \text{succ}_\forall([\phi]_K)$
- $[\Diamond \phi]_K := \text{succ}_\exists([\phi]_K)$

Propositional logic with the above modal operators is however not powerful enough to reason about interesting temporal properties. In the $\mu$-calculus [34], one adds fixpoint formulas $\mu x. \phi$ and $\nu x. \phi$ that denote the least and greatest sets of states $x$ such that $x = \phi$ holds. Fixpoint formulas can be evaluated by means of the Tarski-Knaster iteration [34]. For example, the set of reachable states $S_{\text{reach}}$ can be computed as the set of states satisfying $\Psi_{\text{reach}} := \mu x. (\Psi_I \lor [\Diamond x]_K)$ which leads to the fixpoint iteration $X'_0 := \{ \}$ and $X'_{i+1} := [\Psi_I]_K \cup \text{succ}_\exists(X'_i)$ or the equivalent one starting with $X_0 := [\Psi_I]_K$ and iterating $X_{i+1} := X'_i \cup \text{succ}_\exists(X'_i)$ (as used in the following).

Temporal logics like CTL, LTL, and CTL* can all be expressed in the $\mu$-calculus [34]. For this paper, we are just interested in safety properties, i.e., we want to prove that some property $\Phi$ holds on all reachable states. This can be expressed in the temporal logic CTL as $\text{AG}\Phi$ where $\text{A}$ quantifies over all paths leaving a state and $\text{G}$ quantifies over all points of time on that path. It is known that $\text{AG}\Phi$ is equivalent to the $\mu$-calculus formula $\nu x. \Phi \land \Box x$ holds. A fixpoint-based model checker would therefore compute this set of states according to the Tarski-Knaster iteration starting with $Q_0 := S$ (the set of all states), and iterating $Q_{i+1} := [\Phi]_K \cap \text{pre}_\forall(Q_i)$.

Instead of computing all states that satisfy a safety property $\text{AG}\Phi$ by fixpoint iteration, and checking afterwards whether all initial states are included, we can also prove them by means of one of the following induction rules:

$$
\begin{align*}
\text{if } & \Psi_I \to \Phi, \Psi_{\text{reach}} \land \Phi \to \Box \Phi \to \Psi_{\text{reach}} \to \Phi \\
& \text{then } \Psi_I \to \Phi, \Psi_{\text{reach}} \land \Phi \to \Box \Phi \to \Psi_{\text{reach}} \to \Phi 
\end{align*}
$$

The rule on the right-hand side obviously follows from the one on the left-hand side, since its second subgoal is stronger (validity of $\Phi \to \Box \Phi$ implies validity of $\Psi_{\text{reach}} \land \Phi \to \Box \Phi$). To prove the correctness of the rule on the left-hand side, one can prove (by induction on $i$) that $X_i \subseteq [\Phi]_K$ holds where $X_i$ are the sets of states that can be reached in no more than $i$ steps, i.e., those that occur in the fixpoint iteration $X_0 := \{ \}$ and $X_{i+1} := [\Psi_I]_K \cup \text{succ}_\exists(X_i)$ to compute the reachable states $S_{\text{reach}}$.

In these kinds of proofs, as well as those of the correctness of PDR, the following lemmata are very useful and hold for every set of states $Q$:

- $\text{succ}_\exists(\text{pre}_\forall(Q)) \subseteq Q$ and $Q \subseteq \text{pre}_\forall(\text{succ}_\exists(Q))$
- $Q \subseteq \text{pre}_\forall(Q)$ holds iff $\text{succ}_\exists(Q) \subseteq Q$ (induction steps)
- $\text{succ}_\exists([\text{AG}\Phi]_K) \subseteq [\text{AG}\Phi]_K \subseteq [\Phi]_K \cap \text{pre}_\forall([\Phi]_K)$
- If $\text{AG}\Phi$, i.e., $\nu x. \Phi \land \Box x$, holds on all initial states, then it holds on all reachable states (and is therefore a safety property).

8For example, the correctness of the first induction rule is proved as follows: The induction step is trivial since $X_0 := \{ \}$ holds. For the induction step, we have the induction hypothesis $X_i \subseteq [\Phi]_K$ and we have to prove that $X_{i+1} \subseteq [\Phi]_K$ holds. By the second subgoal, we have (1) $S_{\text{reach}} \cap [\Phi]_K \subseteq \text{pre}_\forall([\Phi]_K)$. By monotonicity, we get (2) $\text{succ}_\exists(S_{\text{reach}} \cap [\Phi]_K) \subseteq \text{pre}_\forall([\Phi]_K)$) from (1). From (2), we now get (3) $\text{succ}_\exists(S_{\text{reach}} \cap [\Phi]_K) \subseteq [\Phi]_K$ and from the IH, we get (4) $S_{\text{reach}} \cap X_i \subseteq S_{\text{reach}} \cap [\Phi]_K$, i.e., (4) $X_i \subseteq S_{\text{reach}} \cap [\Phi]_K$. Further, by monotonicity, we have (5) $\text{succ}_\exists(X_i) \subseteq \text{succ}_\exists(S_{\text{reach}} \cap [\Phi]_K)$ and by transitivity of (3) and (5), we get (6) $\text{succ}_\exists(X_i) \subseteq [\Phi]_K$. By the first subgoal, we have (0) $[\Psi_I]_K \subseteq [\Phi]_K$ and therefore $X_{i+1} := [\Psi_I]_K \cup \text{succ}_\exists(X_i) \subseteq [\Phi]_K$ holds by (0) and (6).
B. Incremental Induction by PDR

PDR tries to prove that a property $\Phi$ holds on all reachable states $S_{\text{reach}}$ by means of induction. The induction rules of the previous section may however not be strong enough, since even though $\Phi$ may hold on all reachable states, the induction step may fail due to counterexamples to induction (CTIs): $\Phi$ may also hold on an unreachable state that has a successor state where $\Phi$ does not hold. For this reason, PDR checks the unreachability of state sets to identify counterexamples as CTIs, and generalizes them to larger state sets.

A property $\varphi$ where $\varphi \rightarrow \Box \varphi$ is valid$^9$ is called inductive, which means that no transition starting in states satisfying $\varphi$ will leave this state set, i.e., $\varphi$ contains all its successor states. In particular, $\Psi_{\text{reach}}$ is inductive, and the goal of PDR is to generate an inductive property $\Psi_i$ that implies a considered property $\Phi$.

PDR is used to check whether a property $\Phi$ holds on at least all reachable states of a transition system $K = (V, \Psi_I, \Psi_R)$. To this end, PDR first makes the following checks

- $\Psi_I \rightarrow \Phi$: If not valid, we have a counterexample of length 0, since the counterexample describes a state that satisfies $\Psi_I$ but not $\Phi$.
- $\Psi_I \rightarrow \Box \Phi$: If not valid, we have a counterexample of length 1, since the counterexample describes a state that satisfies $\Psi_I$ but not $\Box \Phi$, thus there is an initial state with a successor state that violates $\Phi$.

If one of the two is not valid, we have a counterexample for $\Phi$, otherwise, there are no counterexamples of length 0 or 1. PDR continues then by checking the following:

- $\Psi_I \rightarrow \Box \Psi_I$: If valid, we have a proof, since then only the initial states are reachable states, and we already know that they all satisfy $\Phi$.
- $\Phi \rightarrow \Box \Phi$: If valid, we have a proof since then also the induction step for $\Phi$ has been proved this way.

Otherwise, there are reachable states other than the initial states, and there are states that satisfy $\Phi$, but at least one of their successor states violates $\Phi$. Thus, PDR starts then its real work. To this end, it sets up the following initial sequence of predicates$^{10}$:

- $\Psi_0 := \Psi_I$
- $\Psi_1 := \Phi$

This way, we have an initial sequence of predicates $\Psi_0$ and $\Psi_1$. In general, PDR maintains an incrementally increasing sequence $\Psi_0, \ldots, \Psi_k$ with the following properties:

- $\Psi_i = \Psi_0$
- $\Psi_i \rightarrow \Psi_{i+1} \land \Box \Psi_{i+1}$ for $i = 0, \ldots, k - 1$
- $\Psi_k \rightarrow \Phi$, i.e., also $\Psi_i \rightarrow \Phi$ for $i = 0, \ldots, k$
- no $\Psi_i$ is inductive, in particular, we have $\Psi_i \neq \Psi_{i+1}$ for $i = 0, \ldots, k - 1$

One can easily prove that by these invariants, we have the following situation where $X_i$ denotes the states that can be reached from the initial states in no more than $i$ transition steps, and $S_{\text{reach}}$ are the reachable states:

$$
\begin{align*}
\Psi_I & = X_0 \subseteq X_1 \subseteq \ldots \subseteq X_k \subseteq S_{\text{reach}} \\
\Psi_I & = [\Psi_0]_K \subseteq [\Psi_1]_K \subseteq \ldots \subseteq [\Psi_k]_K \subseteq [\Phi]_K
\end{align*}
$$

Having an approximation $\Psi_0, \ldots, \Psi_k$ of $X_0, \ldots, X_k$, the main algorithm of PDR then checks whether $\Psi_k \rightarrow \Box \Phi$ is valid, i.e., whether there are some states in $\Psi_k$ that have successors violating $\Phi$. Depending on this, one of the following is done:

- **Case 1**: $\Psi_k \rightarrow \Box \Phi$ is valid. In this case, we have not yet a counterexample, but also the induction proof was not yet successful. Thus, we have to incrementally extend the sequence of predicates (note that with a sufficiently large $k$, we must finally have $S_{\text{reach}} = X_k \subseteq [\Psi_k]_K \subseteq [\Phi]_K$ so that a proof will be finally found if we can make $\Psi_k$ close enough to $S_{\text{reach}}$, so that it must become inductive). To this end, we could define $\Psi_{k+1} := \Phi$ which would satisfy all PDR invariants, but PDR narrows $\Phi$ first by propagating clauses from $\Psi_k$ to $\Phi$. Hence, it defines $\Psi_{k+1} := \text{narrow}(\Phi, \Psi_k)$, where the latter is defined as follows: narrow$(\Phi, \Psi_k)$ is obtained by checking for all clauses $c_i$ of $\Psi_k := \bigwedge_{i=0}^m c_i$ whether $\Psi_k \rightarrow \Box c_i$ is valid. Assume this is the case for the clauses $c_0, \ldots, c_n$ with $n \leq m$, then we define $\Psi_{k+1} := \text{narrow}(\Phi, \Psi_k) := \Phi \land \bigwedge_{i=0}^n c_i$ to increase the chance of making it inductive. All PDR invariants are maintained this way, and with increasing $k$, we proceed towards a proof or a counterexample.

- **Case 2**: $\Psi_k \rightarrow \Box \Phi$ is not valid. Then, the SAT solver generates a counterexample that satisfies $\Psi_k \land \neg \Box \Phi$, i.e., a (partial) assignment to some of the variables $V$, which is a cube $C_k$, i.e., a conjunction of literals of these variables. PDR now has to check whether one of these states is reachable from the initial states within $k$ steps as explained in Section II-C below. If one of them is reachable within $k$ steps, it is a real counterexample since the property $\Phi$ then does not hold on all reachable states. Otherwise, the states in $C_k$ are all unreachable, thus form a CTI that can now be removed from all predicates $\Psi_i$. To that end, PDR first generalizes the cube $C_k$ to another cube $C'_k$ as explained in Section II-D below so that as many unreachable states as possible are removed by adding the clause $\neg C'_k$ to all $\Psi_0, \ldots, \Psi_k$.

PDR therefore terminates with either a counterexample in case 2, or with a proof as soon as one $\Psi_i$ satisfies $\Psi_i \equiv \Psi_{i+1}$, i.e., became inductive in case 1 or in case 2. As explained above in these two cases, two important procedures are therefore (1) checking the unreachability of a cube $C_k$ within $k$ steps and (2) the generalization of a cube $C_k$ that has been identified as a CTI. These procedures work as explained in more detail in the following two sections.

---

$^9$Note that checking an implication $\psi \rightarrow \Box \varphi$ can be done by a SAT solver by just checking $\psi \land \Psi_k \rightarrow \varphi'$ where $\varphi'$ is obtained from $\varphi$ by replacing every occurrence of a variable $v \in V$ by its corresponding one $x' \in V$.

$^{10}$For the sake of simplicity, we ignore here that PDR actually works on clause sets, which is however important for its implementation.
C. Checking Unreachability of Cubes

To prove the unreachability of (all states of) a cube \( C_k \) within \( k \) steps, we would essentially have to check whether \( \Psi_{k-1} \rightarrow □¬C_k \) is valid. If it is valid, then no state of \( C_k \) can be reached within \( k \) steps. Otherwise, there are states in \( \Psi_{k-1} \) that have successor states in \( C_k \) (but these could be unreachable). Bradley observed that it suffices to check the validity of \( \Psi_{k-1} \land ¬C_k \rightarrow □¬C_k \) and \( \forall \rightarrow ¬C_k \) instead: If both are valid, we conclude that no state in \( C_k \) is reachable within \( k \) steps\(^{11}\), and we can therefore remove all of these states from the predicates \( \Psi_0, \ldots, \Psi_k \). On the other hand, if \( \forall \rightarrow ¬C_k \) should not be valid, we see that some states of \( C_k \) are initial states and are therefore reachable.

Finally, if \( \Psi_{k-1} \land ¬C_k \rightarrow □¬C_k \) should not be valid, then there is another counterexample, thus another cube \( C_{k-1} \), which contains states that satisfy \( \Psi_{k-1} \land ¬C_k \) and that have at least one successor state in \( C_k \). Therefore, we have to recursively check the reachability of \( C_{k-1} \) in \( k - 1 \) steps in the same way. Finally, if we get a counterexample \( C_0 \) from \( \Psi_0 \land ¬C_0 \rightarrow □¬C_0 \), then the sequence \( C_0, \ldots, C_k \) contains a path in the transition system reaching a state in \( C_k \). Otherwise, we conclude that no state in \( C_k \) is reachable in \( k \) steps and can remove it from the approximations \( \Psi_0, \ldots, \Psi_k \) by adding the clause \( ¬C_k \) to these approximations, or better by adding a generalized clause \( ¬C_k' \) as explained in the next section.

D. Generalization of CTIs

Once a cube \( C_k \) has been proved unreachable in \( k \) steps, i.e., finally \( \Psi_{k-1} \land ¬C_k \rightarrow □¬C_k \) is valid, we may add its clause \( ¬C_k \) to all approximations \( \Psi_0, \ldots, \Psi_k \). However, PDR tries to generalize the clause \( ¬C_k \) before adding it by removing single literals from the clause as long as it remains unreachable. This way as many unreachable states as possible are removed. In essence, PDR searches a subclause \( ¬C_k' \) of \( ¬C_k \) such that \( \Psi_{k-1} \land ¬C_k' \rightarrow □¬C_k' \) and \( \forall \rightarrow ¬C_k' \) still remain valid (hence a larger set of unreachable states).

To this end, PDR constructs the subclause lattice \( L_{C_k} := \langle 2^{C_k}, \sqsubseteq \rangle \) whose elements are the subclauses of \( ¬C_k \) that are ordered by the subclause relative relation \( \sqsubseteq \) defined as follows: Two subclauses \( c_1, c_2 \in 2^{C_k} \) satisfy the relation \( c_1 \sqsubseteq c_2 \) iff \( c_1 \subseteq c_2 \) and \( c_1 \land \forall \Psi_k \rightarrow \exists c_2 \) holds. For example, assume that the SAT solver returns \( C_k := p_1 \land ¬p_2 \) as a CTI over \( V := \{p_1, p_2\} \). We have \( ¬C_k = \{¬p_1, p_2\} \), and the subclause lattice is defined by \( L_{C_k} = \langle \{\}, \{¬p_1\}, \{p_2\}, \{¬p_1, p_2\}, \{¬p_1, ¬p_2\} \rangle \). Starting from the top element \( ¬C_k \) of \( 2^{C_k} \), which is inductive relatively to \( \Psi_k \) as proved by the above reachability analysis,

III. SYNCHRONOUS PROGRAMS AND THEIR COMPILATION

A. The Quartz Language

Quartz [35] is an imperative synchronous language that is derived from the Esterel language [5]. The execution paradigm of the Quartz language is based on the synchronous reactive model of computation: The execution of a synchronous reactive system proceeds in discrete reaction steps where in each step, inputs are read from the environment, outputs are immediately generated as a reaction to these inputs, and the internal state of the system is updated for the next reaction step. Reaction steps are declared in Quartz programs by \( \ell : \text{pause} \) statements that define the end of the current and the start of the next macro step. Each \( \ell : \text{pause} \) statement introduces a control-flow label \( \ell \) that is given a name by the compiler or the programmer. All assignments between such \text{pause} statements are executed in virtually zero time.

Due to space limitations, we cannot give a complete overview of the language here, and refer to [35] for more details. Instead, we just list some of the statements used in this paper and refer to their intuitive meaning:

- nothing (empty statement)
- \( x = \tau \) and next(\( x \)) = \( \tau \) (immed./delayed assignments)
- \( \ell : \text{pause} \) (start/end of macro step)
- \{\( \alpha \ x; S \}) (local variable \( x \) of type \( \alpha \))
- \( S_1; S_2 \) (sequences)
- \( S_1 \parallel S_2 \) (synchronous concurrency)
- if(\( \sigma \)) \( S_1 \) else \( S_2 \) (conditional)
- while(\( \sigma \)) \( S \) (while-loop)
- do \( S \) while(\( S \)) (do-while-loop)
- [weak][immediate] abort \( S \) when(\( \sigma \)) (abortion)
- [weak][immediate] suspend \( S \) when(\( \sigma \)) (suspension)

nothing is just an empty statement that is useful for many program transformations. Assignments can be either immediate \( x = \tau \) or delayed next(\( x \)) = \( \tau \) which is similar to hardware circuits where the former describes a wire and the latter a register assignment. In case a variable is not assigned a value by either an immediate or a delayed assignment in a macro step, its value is determined by its storage class: Memorized variables (declared with keyword \text{mem}) store the value they had in the previous macro step while event variables (declared with keyword \text{event}) were reset to a default value. \{\( \alpha \ x; S \}) declares local variable \( x \) to be visible only in \( S \). Sequences, conditionals, and loops should be intuitively clear. Abortion and suspension statements behave as their body statement \( S \) but observe in parallel their preemption condition \( \sigma \). If the latter should be true, an abortion statement terminates,
while a suspension statement freezes the control-flow at the current locations until \( \sigma \) will release it. For abortion and suspension, there are four variants that differ in whether \( \sigma \) is already checked at starting time (\textit{immediate}) and whether the data flow should even take place in the macro steps where a preemption occurs (\textit{weak}).

### B. Compilation of Synchronous Programs

The semantics of Quartz programs has been formally defined by means of structural operational semantics [28] as given in [35] similar to Esterel [6]. For compilation, we have proved the formal correctness [30]–[33] of a translation to so-called \textit{synchronous guarded actions}. That translation runs in quadratic time over a program \( S \) and will generate also at most quadratically many guarded actions in terms of the size of a program \( S \).

Guarded actions are pairs \((\gamma, \alpha)\) such that \( \gamma \) is a boolean program expression, and \( \alpha \) is an atomic action, i.e., an assignment\(^{12}\). For any pause statement \( \ell \) : \texttt{pause} with location variable \( \ell \), there is also a guarded action \((\gamma, \texttt{next}(\ell) = \texttt{true})\) such that \( \gamma \) holds whenever the control-flow moves next to program location \( \ell \). The location variables are thereby considered as \textit{event} variables. In addition, the compiler will generate a further location variable \( \texttt{run} \) that will be initially false, and otherwise true by just adding the guarded action \((\texttt{true}, \texttt{next}(\texttt{run}) = \texttt{true})\) for it.

The meaning of synchronous guarded actions is very simple: In every macro step, we have to evaluate all guards \( \gamma \) and fire all actions \( \alpha \) whose guards are enabled. In more detail, one has to consider the causal dependencies between these actions where sophisticated algorithms based on ternary simulation are used. Using guarded actions, we can also generate equation generation methods. For the following, we focus on the construction of symbolic representations for state transition systems which will be sufficient for this paper.

To that end, assume that our intermediate representation by guarded actions contains the following immediate and delayed actions for some variable \( x \):

\[
(\gamma_1, x = \tau_1), \ldots, (\gamma_p, x = \tau_p), (\delta_1, \texttt{next}(x) = \pi_1), \ldots, (\delta_q, \texttt{next}(x) = \pi_q)
\]

Figure 1 shows then the definition of the initial states predicate \( \text{Init}_x \) and the transition relation \( \text{Trans}_x \) for variable \( x \) which are explained as follows: The initial value of a variable \( x \) can only be determined by its immediate actions. Hence, if one of the guards \( \gamma_i \) of the immediate actions holds, the corresponding immediate assignment defines the value of \( x \). If none of the guards \( \gamma_i \) should hold, i.e., \( \Gamma := \bigvee_{j=1}^p \gamma_j \) is false, the initial value of \( x \) is determined by its default value (which is determined by the semantics, e.g., false for boolean variables and 0 for numeric ones).

The transition relation can be explained similarly: First, also the immediate assignments have to be respected for the current point of time, i.e., whenever a guard \( \gamma_i \) of the immediate actions holds, the corresponding immediate assignment defines the current value of \( x \). If one of the guards \( \delta_i \) of the delayed assignments holds at the current point of time, the next value of \( x \) is determined by the corresponding delayed assignment. Finally, if the next value of \( x \) is not determined by an action, i.e., neither \( \Gamma := \bigvee_{j=1}^p \gamma_j \) holds at the next point of time nor does \( \Delta := \bigvee_{j=1}^q \delta_j \) hold at the current point of time, then the next value of \( x \) is determined by the reaction to absence. Depending on whether it is a memorized or event variable, it may store its previous value or will be reset to a default value.

The definitions given in Figure 1 can be literally used to define input files for symbolic model checkers. For the remainder of the paper, it is moreover important to distinguish between the control-flow and the data flow which are defined as follows where \( \text{Loc}(S) \) denotes all names of pause locations (i.e., control-flow variables) and \( \text{Var}(S) \) denotes all input, output, and local variables of program \( S \):

- \( \Psi_{I}^S := \bigwedge_{\ell \in \text{Loc}(S)} \text{Init}_\ell \) and \( \Psi_{R}^S := \bigwedge_{\ell \in \text{Loc}(S)} \text{Trans}_\ell \)
- \( \Psi_{I}^S := \bigwedge_{x \in \text{Var}(S)} \text{Init}_x \) and \( \Psi_{R}^S := \bigwedge_{x \in \text{Var}(S)} \text{Trans}_x \)

The overall state transition system has then the initial states \( \Psi_{I} := \Psi_{I}^S \wedge \Psi_{R}^S \) and the transition relation \( \Psi_{R} := \Psi_{R}^S \wedge \Psi_{R}^S \).

### IV. Generating Control-flow Invariants for Synchronous Programs

In this section, we present two methods to generate suitable control-flow predicates to enhance the transition relations of synchronous programs for a more efficient formal verification. In particular, the PDR method can benefit a lot from these enhancements as we will show by means of three examples in the next section.

---

\(^{12}\)The Quartz language has further atomic actions like assertions and assumptions that are however not relevant for this paper.
A. Compiler Generated Control-flow Invariants

For the compilation of the guarded actions, the definition of certain control-flow predicates were introduced originally in [32]. One of these predicates is \( \text{insd}(S) \) which is just the disjunction of the program locations contained in \( S \). It holds whenever the control-flow is active in \( S \) and it is just the disjunction of the control-flow location contained in \( S \). Using \( \text{insd}(S) \), we can recursively compute the control-flow predicate \( \text{InvarCF}(S) \) for any Quartz program \( S \) as shown in Figure 2.

Intuitively, \( \text{InvarCF}(S) \) states that the control-flow can never be active at both substatements \( S_1 \) and \( S_2 \) of sequences and conditional statements.

Actually, this invariant has already been generated in [32] (see Lemma 2 in that paper), where it has been required to prove the correctness of the generated set of guarded actions with respect to the structural operational semantics of the language. The validity of this predicate on all reachable states has been formally proved there, and it can also be proved by means of induction on the state transition system that is symbolically encoded by the formulas defined in the previous section.

**Theorem 1 (InvarCF(S)):** For every program \( S \), all reachable states satisfy the predicate \( \text{InvarCF}(S) \) as defined by the algorithm shown in Figure 2.

Even though the computation of \( \text{InvarCF}(S) \) is straightforward (it can be done in linear time in terms of the size of \( S \)), this control-flow predicate contains important information about the unreachability of states that PDR otherwise has to prove first. We can avoid this proof overhead by simply adding \( \text{InvarCF}(S) \) to the transition relation and thus use \( \Psi_\text{R} := \Psi_\text{R} \wedge \text{InvarCF}(S) \) for PDR instead.

We can even add a bit more information that is equally simple to get: In particular, note that all control-flow labels are false at the initial point of time, and this is also encoded in \( \Psi_\omega \). Moreover, due to the single guarded action \((\text{true, next (run)} = \text{true})\) of the active control location, \( \text{run} \) will hold at every point of time other than the initial point of time. Therefore, any other label implies \( \text{run} \).

B. Control-flow Invariants by Fixpoint Computation

Recall that the symbolic representation of Quartz programs given by the predicates \( \Psi_\omega \) and \( \Psi_\text{R} \) in Section III-B describes a possibly infinite state transition system with discrete transitions (due to infinite data types). However, any program has only finitely many control-flow locations, and therefore, projecting its reachable states \( S_\text{reach} \) to the control-flow locations will yield a finite state transition system. Its states are called in the following the reachable control-flow states \( S_\text{reach}^{\text{cf}} \).

The predicate \( \text{InvarCF}(S) \) is an over-approximation of the reachable control-flow states \( S_\text{reach}^{\text{cf}} \) due to two facts: First, it does not take care of the infeasibility of control-flow conditions \( \sigma \) that occur in conditionals, while-loops, do-while-loops, abortion and suspension statements. Instead, \( \text{InvarCF}(S) \) considers both \( \sigma \) and its negation \( \neg \sigma \) to be satisfiable, so that all substatements can be activated. If such a condition should however be unsatisfiable, some of the contained control-flow locations will not be reachable. Note that this can also be due to nested statements like nested conditionals where the combination of their conditions may become unsatisfiable.

Heuristics may be applied to check for infeasible path conditions as it is classically done in WCET analysis, and it can be done today much better by the help of SMT solvers. In practice, however, it is usually never the case that such infeasible paths occur. We therefore do not consider this problem, but focus on another one that is simpler to handle, but more relevant in practice.

Another reason why \( \text{InvarCF}(S) \) may be an over-approximation of the reachable control-flow states \( S_\text{reach}^{\text{cf}} \) is that it does not consider how the control-flow locations contained in the substatements of parallel statements are related to each other. We can significantly improve the approximation obtained by \( \text{InvarCF}(S) \) by computing the reachable states of the state transition system by projecting its initial states and transition relation to the Loc(S) variables only.

**Definition 1 (Abstract Control-flow Transition System \( K^{\text{abs}} \)):** Given a program \( S \) with control-flow variables \( \text{Loc}(S) \) and input, output, and local variables \( \text{Var}(S) = \{ x_1, \ldots, x_n \} \), the abstract control-flow transition system \( K^{\text{abs}} \) is defined over the variables \( \text{Loc}(S) \) with the following initial states and transition relation:

- \( \Psi_\omega := \exists x_1, \ldots, x_n, \Psi_\text{cf}(S) \)
- \( \Psi_\text{R} := \exists x_1, \ldots, x_n, \exists x_1', \ldots, x_n', \Psi_\text{R}(S) \)

The reachable states \( \text{ReachCF}(S) \) of \( K^{\text{abs}} \) are defined by the iteration \( \chi_i^{\text{abs}} := \Psi_\omega^{\text{abs}} \) and \( \chi_{i+1}^{\text{abs}} := \chi_i^{\text{abs}} \cup \text{insd}(\chi_i^{\text{abs}}) \).

Note that \( K^{\text{abs}} \) has only finitely many states, and usually also not that many states. These are the states of the corresponding extended finite state machine (EFSM) that is also often considered for code generation. The above abstraction makes nondeterministic choices on all control-flow expressions \( \sigma \) that occur in \( \Psi_\omega \) and \( \Psi_\text{R} \) so that no variables other than the location variables occur in \( \Psi_\omega \) and \( \Psi_\text{R} \). In practice, we can quickly compute the reachable states \( \text{ReachCF}(S) \) of \( K^{\text{abs}} \) by means of symbolic model checking using BDDs.
The control-flow invariant \( \text{InvarCF} \) is better and is equivalent to \( \text{ReachCF} \). Hence, \( \text{ReachCF} \) and \( \text{InvarCF} \) are equivalent.

**Theorem 2 (ReachCF(S)):** For every program \( S \), all reachable states imply ReachCF(S), so that the projection of the reachable states of the program \( S \) to the location variables \( \text{Loc}(S) \), i.e., \( S^c_{\text{reach}} \) is a subset of ReachCF(S). Moreover, we always have ReachCF(S) \( \rightarrow \) InvarCF(S), so that ReachCF(S) is a better approximation of \( S^c_{\text{reach}} \) than InvarCF(S).

It has to be remarked here that one core idea of PDR is to avoid the computation of the reachable states by means of a fixpoint computation. It may therefore seem to be counterintuitive that we are using a fixpoint computation at this stage. However, since this one is just restricted to the control-flow, typical BDD-based approaches can quickly compute ReachCF(S). The challenges for the reachable state space computation are rather in the data flow, and we therefore employ PDR for this part only.

**V. EXAMPLES**

In this section, we consider three generic modules \( \text{CfSeq} \), \( \text{CfIte} \), and \( \text{CfPar} \) shown in Figures 3, 6, and 7 to illustrate our definitions and their effects on PDR.

**A. Example 1: Module \( \text{CfSeq} \)**

Module \( \text{CfSeq} \) shown in Figure 3 is just a sequence of \( N \) \text{pause} statements. Figure 3 shows the computed state transition system given by \( \Psi_I \) and \( \Psi_R \) that is for this program identical with the control-flow state transition system \( \Phi_I^c \) and \( \Phi_R^c \) and its abstraction \( \Phi_I^{abs} \) and \( \Phi_R^{abs} \). The control-flow invariant InvarCF(\( \text{CfSeq} \)) even encodes the correct reachable states, and thus ReachCF(\( \text{CfSeq} \)) cannot do better and is equivalent to InvarCF(\( \text{CfSeq} \)) in this example. Hence, ReachCF(S) and even InvarCF(S) precisely compute the reachable control-flow states \( S^c_{\text{reach}} \).

Figure 4 shows the state transition system for module \( \text{CfSeq} \) for \( N = 2 \) where the reachable states are given in green color and the initial state is drawn with double lines. For the three state variables \( V = \{ \text{run}, p_1, p_2 \} \), there are eight possible states, but only four of the eight are reachable. Thus, there are many CTIs for possible safety properties. For example, to prove that \( \neg(p_1 \land p_2) \) holds on all reachable states, we will have a CTI at state \( s_4 \) since it satisfies this property, but its successor state \( s_7 \) does not. PDR will therefore not be able to prove this property within one step.

However, if we add the control-flow invariant InvarCF(\( \text{CfSeq} \)) to the transition relation, then only the reachable states will have outgoing transitions. Thus, no CTIs are possible, and PDR will be able to prove every valid safety property by just checking the induction base and induction step. Note that in case of general \( N \), there are \( 2^N \) reachable states, but \( 2^N \) states exist in the state transition system. The additional proof overhead of PDR grows with \( N \).

**B. Example 2: Module \( \text{CfIte} \)**

Module \( \text{CfIte} \) shown in Figure 6 is a conditional statement whose two substatements are both sequences of \( N \) \text{pause} statements. The decision depends on an input variable \( i \). Figure 6 shows the computed state transition system given by \( \Psi_I \) and \( \Psi_R \) that is for this program identical with the control-flow state transition system \( \Phi_I^c \) and \( \Phi_R^c \) since there are again no assignments to local or output variables.

In contrast to module \( \text{CfEq} \), we have however an input variable, so that the abstract control-flow automaton has another symbolic representation that is obtained by existential quantification over the control-flow conditions, which is just \( i \) in this case. The decision whether to branch from the initial state to the location \( p_{i-1} \) or \( q_{i-1} \) depends in \( \Psi_R^c \) on the input variable \( i \), but is made nondeterministically in \( \Phi_R^{abs} \). Obviously, this does not change the reachable control-flow states, so that ReachCF(S) will precisely compute the reachable control-flow states \( S^c_{\text{reach}} \).

The control-flow invariant InvarCF(\( \text{CfIte} \)) just encodes that if one branch of the conditional statement is active, none of the labels of the other branch can be active, and since these are again sequences as module \( \text{CfSeq} \), it furthermore states that only one of the locations can be active. Again, InvarCF(\( \text{CfIte} \)) precisely computes the reachable control-flow states \( S^c_{\text{reach}} \) like ReachCF(S).

Figure 5 shows the state transition system for module \( \text{CfIte} \) for \( N = 2 \) where again the reachable states are given in green color and the initial state is drawn with double lines. For the five state variables \( V = \{ \text{run}, p_1, p_2, q_1, q_2 \} \), there are \( 2^5 = 32 \) possible states, but only six of them are reachable. Again, there are many CTIs for possible safety properties that may trouble PDR.

As in the previous example, the addition of the control-flow invariant InvarCF(\( \text{CfIte} \)) to the transition relation will only allow the reachable states to have outgoing transitions. Thus, no CTIs are possible anymore, and PDR will be able to prove every valid safety property by just checking the induction base and induction step.
C. Example 3: Module CfPar

Module CfPar shown in Figure 7 is a parallel statement whose two substatements are both sequences of $N$ pause statements. When it starts, both pause statements $p_1$ and $q_1$ are entered by the control-flow, then $p_2$ and $q_2$, and so on.

Figure 7 shows the computed state transition system given by $\Psi_T$ and $\Psi_R$ that is for this program identical with the the control-flow state transition system $\Psi_{\text{cf}}^T$ and $\Psi_{\text{cf}}^R$ and also identical to the abstract control-flow state transition system $\Psi_{\text{abs}}^T$ and $\Psi_{\text{abs}}^R$.

The control-flow invariant $\text{InvarCF}(\text{CfPar})$ just encodes that at most one of the control-flow locations $p_1, \ldots, p_N$ and also at most one of the control-flow locations $q_1, \ldots, q_N$ is valid. However, it does not relate the two threads to each other. Hence, this predicate allows all combinations of pairs $p_i, q_j$ to be active at the same time which is however not possible in any reachable state. $\text{InvarCF}(\text{CfPar})$ is therefore a coarse abstraction.

The computation of the reachable states with $\Psi_{\text{abs}}^T$ and $\Psi_{\text{abs}}^R$ yields however the precise set of reachable control-flow states $S_{\text{reach}}$, and can therefore give PDR the exact information from the beginning.

The state transition diagram is shown in Figure 8 for $N = 2$ where again the reachable states are given in green color and the initial state is drawn with double lines. For the five state variables $V = \{\text{run}, p_1, p_2, q_1, q_2\}$, there are $2^5 = 32$ possible states, but only four of them are reachable so that PDR may have to deal with CTIs.

In contrast to the previous example, the addition of the control-flow invariant $\text{InvarCF}(\text{CfPar})$ will not remove all transitions from unreachable states. Of course, $\text{InvarCF}(\text{CfPar})$ holds in the four reachable states, but also in additional six states: $s_{17}, s_{18}, s_{20}, s_{22}, s_{24}, s_{25}$. It can therefore help PDR with some, but not with all safety properties. On the other hand, ReachCF($S$) computes in this example precisely the reachable control-flow states $S_{\text{reach}}^\text{cf}$ so that PDR can prove any safety property directly without even starting the incrementation procedure.

VI. Conclusions

We presented two methods to generate control-flow information from imperative synchronous programs that can be used to refine the transition relation for making PDR more efficient. Both methods approximate the reachable states of the program by the reachable control-flow states only and therefore do not consider the data flow at all. The first method computes the predicate InvarCF in a single pass over the program. In case of sequential programs, it often computes the reachable control-flow states precisely, but in the presence of parallel threads, it cannot relate the locations of different threads to each other. The second method is able to do this by a symbolic computation of the reachable states ReachCF of an abstract control-flow automaton that is obtained by existential quantification over the data flow variables of the formulas for the initial states and the transition relation of the control-flow of a synchronous program.

PDR can benefit from this additional control-flow information. Since the predicate InvarCF is already computed by our compiler, we decided to always add it to the transition relations for formal verification. It will always speed up PDR as well as also the fixpoint computations. For programs where many parallel statements are involved, one has to decide whether it is better to compute ReachCF or to let PDR work on checking the reachability problems. To our experience, the computation of ReachCF was always quickly performed, so that we would recommend this procedure to enhance PDR. PDR can then completely focus on the data flow, where the compilers can give less help.

References


Synchronous Program

```c
module CfIte(bool ?1) {
  if(1) {
    p_1: pause;
    p_2: pause;
    ...
    p_N: pause;
  } else {
    q_1: pause;
    q_2: pause;
    ...
    q_N: pause;
  }
}
```

State Transition System

\[
\Phi := \{i, \text{run}, p_1, \ldots, p_N, q_1, \ldots, q_N\} \\
\Psi_I := \Psi^I_{2} := \neg \text{run} \wedge \bigwedge_{i=1}^{N} \neg p_i \wedge \neg q_i \\
\Psi_R := \Psi^R_{2} := \neg \text{run} \wedge \bigwedge_{i=1}^{N} \neg p_i \wedge \neg q_i \\
\Psi_{abs}^I := \Psi^I_{2} := \neg \text{run} \wedge \bigwedge_{i=1}^{N} \neg p_i \wedge \neg q_i \\
\Psi_{abs}^R := \Psi^R_{2} := \neg \text{run} \wedge \bigwedge_{i=1}^{N} \neg p_i \wedge \neg q_i \\
\text{InvarCF}(\text{CfIte}) \Leftrightarrow \neg \text{run} \rightarrow \bigwedge_{i=1}^{N} \neg p_i \wedge \neg q_i \\
\text{ReachCF}(\text{CfIte}) \Leftrightarrow \text{InvarCF}(\text{CfIte})
```

Fig. 6. Example Quartz program CfIte with state transition systems and control-flow invariants


Fig. 8. State transition system for module CFPaR for \( N = 2 \).