Inferring Environment Assumptions in Model Refinement

Srinivas Nedunuri
Sandia National Laboratories
Livermore, CA, USA

Douglas R. Smith
Kestrel Institute
Palo Alto, CA, USA

Abstract

Model Refinement is a novel approach to reactive program synthesis that iteratively refines an over-approximating model of a desired system behavior by eliminating undesired behaviors. In contrast to many current automata based approaches to reactive synthesis, it does not require a finite state space or user supplied templates. Instead it symbolically computes the required invariant by solving a system of definite constraints.. The original work on model refinement, however, assumed that both the assumptions on the environment (in an Assume-Guarantee setting) and the constraints on the system variables necessary to guarantee the required behavior were fixed and known. Sometimes, though, the designer of a system has some intended behavior and wishes to know what the minimal assumptions are on the environment under which the system can guarantee the required behavior; or to know what the constraints are on the system variables under known environment assumptions. In other words, we wish to solve a *parametric* model refinement problem. Our contribution in this paper is to show how such a problem can be solved when the constraints are assumed to be an interval of the form $m \ldots n$.

ACM Reference Format:
Srinivas Nedunuri and Douglas R. Smith. 2022. Inferring Environment Assumptions in Model Refinement. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Model Refinement [12] is a novel approach to reactive program synthesis that iteratively refines an over-approximating model of a desired system behavior by eliminating undesired behaviors, as defined by some safety requirement. In contrast to many current automata based approaches to reactive synthesis [1, 3, 9], it does not require a finite state space or user supplied templates. Instead it symbolically computes the required invariant by solving a system of definite constraints. The original work on model refinement, however, assumed that both the assumptions on the environment (in an Assume-Guarantee setting [8]) and the constraints on the system variables necessary to guarantee the required behavior were fixed and fully defined. Sometimes, though, the designer of a system has some intended system behavior in mind and wishes to know what the minimal assumptions are on the environment under which the system can guarantee the required behavior; or to know what the necessary constraints on the system variables are under known environment assumptions. In other words, we wish to solve a *parametric* model refinement problem. Our contribution in this paper is to show how such a problem can be solved when the constraints are assumed to be an interval of the form $m \ldots n$.

After covering the preliminaries, Section 3 briefly reviews the theory and background for model refinement. Our work starts in Section 4.

2 Preliminaries

2.1 Required Properties

We focus on safety properties formulated in a simple linear temporal logic of actions, similar to Lamport's TLA [7]. A *state* is a map from variables to (type-consistent) values. *State formulas* are boolean-valued expressions formed over the variables of a state and the constants (including functions) relevant to an application domain. A state formula p denotes a predicate $[\![p]\!]$ over states, so $p(s)$ denotes the truth value $[\![p]\!](s)$ for state s. *Actions* are boolean-valued expressions formed over variables, primed variables, and the constants (including functions) relevant to an application domain. An action a specifies a state transition and it denotes a predicate $[\![a]\!]$ over a pair of states, and $a(s,t)$ denotes the truth value $[\![a]\!](s,t)$ for states s and t. The expression $x = x' + 1 + y$ is a typical action where the unprimed variables refer to the first state and primed variables refer to the second state.

A *basic safety property* (or simply a safety property) has the form φ where φ is a state formula or an action. The truth of a safety formula φ at position n of a trace σ (an infinite sequence of states) is denoted $\sigma, n \models \varphi$ and defined, by induction on the structure of φ, as follows:

- $\sigma, n \models p$, for p a state formula, if p holds at state $\sigma[n]$, i.e. $[\![p]\!](\sigma[n])$;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- $\sigma, n \models a$, for a an action, if a holds over the states $\sigma[n], \sigma[n+1]$, i.e. $[\![a]\!](\sigma[n], \sigma[n+1])$;
- $\sigma, n \models \neg\varphi$ if $\sigma, n \not\models \varphi$;
- $\sigma, n \models \varphi \wedge \psi$ if both $\sigma, n \models \varphi$ and $\sigma, n \models \psi$;
- $\sigma, n \models \varphi \Rightarrow \psi$ if either $\sigma, n \models \neg\varphi$ or $\sigma, n \models \psi$, or both;
- $\sigma, n \models \varphi$ if $\sigma, i \models \varphi$ for all $i \geq n$.

$stateP(\varphi)$ holds if φ is a state formula. $actionP(\varphi)$ holds if φ is an action.

2.2 Behavioral Models

Formally, a model is a *labeled control flow graph*[1] (LCFG) $\mathcal{M} = \langle \mathcal{V}, N, A, \mathcal{L} \rangle$ where

- $\mathcal{V}$: a countable set of variables; implicitly each variable has a type with a finite (typically first-order) specification of the predicates and functions that provides vocabulary for expressions and constrains their meaning via axioms. The aggregation of these variable specifications is called the *application domain theory* (or simply domain theory) of the problem at hand.
- N: is a finite set of nodes. Associated with each node $m \in N$, we have a finite subset of observable/local variables $V(m) \subseteq \mathcal{V}$. N has a distinguished node m_0 that is the initial node. An LCFG is *arc-like* if it also has a designated final node m_f. Multi-node LCFGs are useful for modeling the behavior of multi-player games with differing invariants for the players.
- A: a finite set of directed arcs, $A \subseteq N \times N$. Each node m may have an identity self-transition $id_m = \langle m, m \rangle$, called *stutter* that changes the values of no observable variables.
- $\mathcal{L}$: a set of labels. For each node $m \in N$, we have a label $L_m \in \mathcal{L}$ that is a state formula over $V(m)$ representing a node invariant. When there is only one node, we drop the subscript. For each arc $a = \langle m, n \rangle$, label $L_a \in \mathcal{L}$ is an action over $V(m) \times V(n)$.

In reactive system design, it is commonly the case that the variables at all nodes are the same, so $V(m) = V(n)$ for all nodes $m, n \in N$ and all variables are global. In functional algorithm design it is typical that the variables at each node are disjoint, effectively treating all variables as local to a unique node. Most programming languages support models that have both global and local variables. Here we consider only single node systems, but our work generalizes to the multi-node case [12].

Generally, a *state* st_m is a type-consistent map of values to the observable variables of node $m \in N$. A node m denotes the set of states $[\![m]\!] = \{st \mid [\![L_m]\!](st)\}$. The label L_{m_0} is the *initial condition* of the model and denotes the set of initial states.

Arc label L_a generally specifies a non-deterministic action, whose non-determinism may be reduced under refinement.

[1]An LCFG differs from a Labeled Transition System by supporting multiple nodes and multiple arcs between nodes

In reactive systems, which have a game-like character, some of the non determinism is due to the uncontrollable behavior of the environment or an adversarial agent. For refinement purposes, it is necessary to specify which parts of the non determinism are refinable/controllable and which are unrefinable/uncontrollable. Accordingly, the label L_a of an action has the general form:

$$L_a(st_m, e, u, st_n) \;\equiv\; e \in E_a(st_m) \wedge U_a(st_m, u) \wedge st_n = f_a(st_m, u, e)$$

where

1. input e is treated as an uncontrollable environment or adversary input, and ranges over the unrefinable set $E_a(st_m)$;
2. input u is treated as a controllable input, which satisfies the refineable constraint $U_a(st_m, u)$;
3. function f_a gives the deterministic response of the action.

The variability of the control input specifies the refinable part of $L_a(st_m, e, u, st_n)$. This kind of formulation of actions is common in modeling discrete and continuous control systems [13]. We have

$$[\![a]\!] = \{\langle st_m, st_n \rangle \mid \exists e, u.\ L_a(st_m, e, u, st_n)\}.$$

Note that e and u are independent of each other. Alternative formulations are easily made in which one depends on the other.

Semantics. A *trace* is an infinite sequence of states. An LCFG $\mathcal{M} = \langle \mathcal{V}, N, A, \mathcal{L} \rangle$ generates a trace $tr = st_0, st_1, \ldots$ if

1. Initially, st_0 is a legal state of the initial node m_0, i.e. $st_0 \in [\![m_0]\!]$;
2. Inductively, if $i \geq 0$ and st_i is a legal state of node m, i.e. $st_i \in [\![m]\!]$, then there exists arc $a = \langle m, n \rangle$ where $\langle st_i, st_{i+1} \rangle \in [\![a]\!]$ and where st_{i+1} is a legal state of node n; i.e. $st_{i+1} \in [\![n]\!]$.

$\mathcal{M}$ denotes the set of all traces that can be generated by $\mathcal{M}$, written $[\![\mathcal{M}]\!]$.

A node m and a legal state st_m is *nonblocking* if there is an arc $a = \langle m, n \rangle$ and control choice u such that $U_a(st_m, u)$ and a transitions to a legal state of n regardless of the environment input. In game-theoretic terms, if all reachable nodes and states of the model are nonblocking, then the system has a winning strategy. A key part of model refinement is the elimination of blocking states in the model.

3 Model Refinement

This section briefly reviews the model refinement approach, see [12] for details.

3.1 Specification and Refinement.

A *system specification* $\mathcal{S} = \langle \mathcal{M}, \Phi \rangle$ is comprised of a LCFG $\mathcal{M}$ and a set of required properties Φ. A system specification denotes the set of traces generable by $\mathcal{M}$ that also satisfy all

properties in Φ:

$$[\![\mathcal{S}]\!] = \{tr \mid tr \in [\![\mathcal{M}]\!] \wedge tr \models \Phi\}.$$

We introduce a preorder relation on system specifications called *refinement*: $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$ if $[\![\mathcal{S}_1]\!] \supseteq [\![\mathcal{S}_2]\!]$. We say $\mathcal{S}_1$ refines to $\mathcal{S}_2$ or $\mathcal{S}_2$ is a refinement of $\mathcal{S}_1$. [2]

Given system specification $\mathcal{S} = \langle \mathcal{M}, \Phi \rangle$, the goal of model refinement is to transform $\mathcal{S}$ to a new system specification $\mathcal{S}' = \langle \mathcal{M}', \{\} \rangle$ that refines $\mathcal{S}$. The intent is that $\mathcal{M}'$ is the least refinement of $\mathcal{M}$ that satisfies all properties in Φ. Here we only look at refinements that strengthen the node and variable part of arc labels. Refinements that refine nodes and arcs themselves into sub LCFGs are examined in [12].

3.2 Model Refinement as Constraint Solving

Model refinement transforms a model $\mathcal{M}$ and required properties Φ into a model $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq \mathcal{M}' \wedge \mathcal{M}' \models \Phi$. We define now a constraint system whose solutions correspond to refinements of $\mathcal{M}$ that satisfy Φ. The intent is to find the greatest solution of the constraint system, which corresponds to the minimal refinement of $\mathcal{M}$ that satisfies Φ. In some cases, we may need to settle for a near-greatest solution instead. In formulating model refinement as a constraint satisfaction problem, we treat the node labels L_m and arc labels L_a as variables, whose assigned values are state and action predicates, respectively. We can view the constraint system as taking place in the Boolean lattice of formulas with implication as the partial order (i.e. a Tarski-Lindenbaum algebra). Each constraint provides an upper bound on feasible values of one variable. A feasible solution to the constraint system is an assignment of formulas to each variable that satisfies all the constraints of the system. We discuss below how to assure finite convergence of the constraint solving process as the lattice may be of infinite height. To simplify the presentation, in this paper we consider only a single node (but multiple state) systems and assume LIA as the underlying theory. The work generalizes to the multi-node case.

First define a formula *weakest controllable predecessor* (WCP) denoted wcp is a predicate transformer:

$$wcp(L) \equiv \forall e.\, e \in E(st) \implies L(f_a(st, e, u)) \qquad (1)$$

That is, wcp is the weakest formula over $V(m) \bigcup \{u\}$ characterizing those states from which there is some control value such that for any environment input e the transition a is assured to reach a state st_n satisfying the post-state predicate L. The formula ensures that the system transitions to a legal state regardless of the environment input.We characterize the model refinement transformation by a two-stage constraint system. The first stage enforces initial state constraints and the second stage enforces general behavioral constraints.

Stage 0. (Initialization).

[2] generally, there is a map from $[\![\mathcal{S}_2]\!]$ to $[\![\mathcal{S}_1]\!]$

Let m_0 be a solution to the constraint-satisfaction problem posed by the conjunction Θ of required properties that are state properties (not temporal properties). For each variable $v \in V_{m_0}$, set the initial value of v to $m_0(v)$.

Stage 1. Generate the following constraints for each required temporal property $\Box\varphi$:

1. **Node Localization:** $L \implies \varphi$ if φ is a state predicate expressed over the variables at m;
2. **Arc Localization:** $L_a \implies \varphi$ for the arc $a = \langle m, m \rangle \in A$ if φ is an action expressed over the variables at m;
3. **Control Constraint:** $U \implies wcp(L)$,
4. **Node Invariant:** $L \implies \exists u.\, U$

Given a specification $\mathcal{S} = \langle M, \Phi \rangle$, the model refinement transformation first refines $\mathcal{S}$ by solving the stage 0 constraint problem to initialize state variables, then further refines the specification by solving the stage 1 constraints.

The Initialization constraints (1) and (2) provide upper bounds on the node labels. The Control constraints (3) are the essentially synthetic aspect of model refinement as they serve to eliminate any blocking states at a node with respect to an outgoing arc. The Node Invariant constraints (4) serve to eliminate blocking states at a node with respect to all of its outgoing arcs.

ALGORITHM 1: Model Refinement Algorithm

```
if stateP(φ)
    then: L ← L ∧ φ
    else L_a ← L_a ∧ φ
do
    U ← U ∧ wcp(L)
    L ← L ∧ ∃u. U
until L ⟹ U ∧ L.
```

A straightforward algorithm for solving the constraint system over the labels on a model is presented in Figure 1. The iteration converges to a fixpoint when the labels do not change in an iteration. An efficient control strategy for this algorithm is presented in [10], which maintains a queue of constraints that are possibly violated, resulting in an optimal algorithm for solving definite constraints. Assuming the algorithm in Figure 1 converges to a greatest fixpoint, then (1) the result is a weakest refinement of $\mathcal{S}$ that satisfies the required properties Φ, and (2) the *derived initial condition* is the final refined invariant L_0' (which may be the same as the initial invariant L_0). The derived input condition defines the set of nonblocking initial states from which the system can ensure that all behaviors stay within the specified safety conditions. In a model-checking scenario where the model doesn't check, the derived initial condition may provide a useful characterization of the model's failure, complementing any derived counterexamples.

Several key problems arise in solving the constraint system: (1) eliminating the quantifiers in wcp instances, (2) simplifying intermediate formulas, and (3) ensuring termination. For (1) we make use of Z3's built in QE procedure for LIA. For (2) we have developed a custom simplifier which aggressively carries out context-dependent simplification. For (3), forcing termination in a fixpoint iteration algorithm is addressed by widening mechanisms from abstract interpretation [5].

3.3 Example: Packet Flow Control

In this example, based on [11], a buffer is used to control and smooth the flow of packets. We model this problem as in discrete control theory with a plant (buffer), environment/disturbance input e, and control input u. This plant is modeled by a single linear transition that updates the state of the plant. The goal is to assure that the system keeps no more than 20 packets in the buffer buf and keeps the outflow rate out at no more than 4 packets per time unit. The environment supplies a stream of packets that varies up to 4 packets per time unit. The system can change the outflow rate by ±1 per time unit, that is $-1 \leq u \leq 1$.

We specify this system using the following TLA-like notation, which lists the (global) state variables and their initial invariant, the one transition and its initial action, and the required safety properties. This is a classical discrete control problem with a single global state and single transition.

Module FC
  State
    $buf, out : Integer$
  State Invariant
    $0 \leq buf \wedge 0 \leq out$
  Transitions
    $Update([u],[e]) \triangleq$
      $-1{\leq}u{\leq}1 \wedge 0{\leq}e{\leq}4\wedge$
      $buf' = buf + e - out \wedge out' = out + u$
  Required Properties
    $buf = 0$
    $out = 0$
    $\Box\ 0{\leq}buf \wedge buf{\leq}20 \wedge 0{\leq}out \wedge out + u{\leq}4$
End Module

Alg. 1 applied to this problem converges after 4 iterations (~5s on a 3GHz i7 laptop) with an invariant (i.e. node label, L) of $0 \leq out{\leq}4 \wedge 0{\leq}buf{-}out{\leq}16 \wedge -3{\leq}buf{-}3*out{\leq}11 \wedge -6{\leq}buf{-}4{*}out{\leq}10$. With the invariant in hand, it is straightforward to calculate the guard U using the Node Invariant constraint above:
$-1{\leq}u{\leq}1 \wedge\ \ 0{\leq}out + u{\leq}4 \wedge\ \ -6{\leq}buf - 4*u - 5*out{\leq}6 \wedge\ -1{\leq}buf - 2*u - 3*out{\leq}9$.
A control function (satisfying U) can then be extracted from this space [12].

4 Inferring Environment Assumptions

[12] uses an Assume-Guarantee style of reasoning [8]. Both the assumption E and the system properties required to guarantee the component's behavior have to be provided by the designer. However, often the designer of a system has some intended system behavior in mind and wishes to know what the minimal assumptions are on the environment under which the system can guarantee the required behavior. In the case of the packet buffer example, this might be to know what the largest amount of incoming packets the buffer can handle and still maintain the invariant. Another question that often arises is, under some given environment assumptions, what does the system require in order to provide the required guarantee of behavior? Again in the packet buffer example, this might be needing to know the smallest size of buffer that wold prevent overflow. In this paper, we look at both of these problems. Consider again the WCP formula (eqn 1):

$$wcp(L)\ \equiv\ \forall e.\, e \in E(st) \rightarrow L(f(st,e,u))$$

As we are dealing with linear systems we will consider E of the form $m_e \leqq e \leqq n_e$[3]. Our goal is to infer the values of m_e and n_e which impose the least restrictions on e, namely an interval which is at least as large as any other interval which still satisfies the invariant. To achieve this we retain the variables m_e and n_e from the WCP above in the iterative algorithm 1 while applying Z3's quantifier elimination and our own custom simplification at every step. When Alg. 1 converges with an inductive state invariant, $L(s, m_e, n_e)$, we simply ask the SMT solver to solve for the following constraint:

$$L(s,m_e,n_e) \wedge \forall s', m_e', n_e' \centerdot L(s', m_e', n_e') \rightarrow n_e' - m_e' \leq n_e - m_e \quad (2)$$

That is determine the largest interval $[m_e, n_e]$ for which the invariant L holds. This step is carried out once Alg. 1 terminates with an inductive invariant L.

Example 4.1. If we replace the fixed upper bound of 4 on e in Example 3.3 above with n_e resp. and solve using our approach, Alg. 1 terminates after 9 iterations, and the SMT solver returns 4 as a solution to constraint 2. That is, the largest possible range for e is $0 \leq e \leq 4$. Doubling the buffer size to 40, the solver returns $0 \leq e \leq 6$.

As an example of where we can apply the same idea to discovering system parameters, we may also wish to know the smallest buffer size possible for the a given bound on e.

Given $0 \leq buf \leq n_b$, and assuming $0 \leq e \leq 4$, using a variant of Eqn. 2 above, namely

$$L(s,n_b) \wedge \forall s', n_b' \centerdot L(s', n_b') \rightarrow n_b \leq n_b' \quad (3)$$

, the solver returns $n_b \geq 20$.

[3]More general linear constraints that are composed of conjunctions and negations are possible, but will require some kind of template from the designer.

There are also situations in which the lower bound m_e is not necessarily 0, as the following example illustrates

Example 4.2. Suppose the packet buffer is supplying packets to a video or audio processor. It is required to maintain the outflow in the range $2 \ldots 4$. Alg. 1 converges after 22 iterations with an invariant from which we extract the requirement that e must also lie between 2 and 4.

4.1 Composition

We have shown how to infer the environment assumptions on a single component. However, a component is rarely used in isolation. In this case, even if the most lenient environment assumptions on a component were known (e.g. a predesigned component) there is still the problem of determining the environment assumption for the composition. In this section we provide some examples that illustrate our approach applied to such situations.

Example 4.3. Consider two of the packet buffers from Example 4.1 in series as in Fig. 1 (ignore *pd* for now). The buffers might represent nodes in a packed switched network, being used to buffer incoming packets and are required to maintain the outflow rate below some maximum (4) so as to prevent network congestion. Assume *outA* is limited to 3 and *outB* to 4 packets/time unit. . We wish to know what the maximum possible range is for the rate of incoming packets. Instead of e we use the more meaningful name *pi*. Applying our approach to this problem returns a maximum range of 0..3 for pi. When the buffer sizes are increased from 20 to 40, the maximum value of pi is 6.

Example 4.4. Consider two of the packet buffers in Example 4.2 connected in series as in Fig. 1 Assume a lossy connection with packets possibly being dropped (*pd*) in the connection between A and B. For simplicity, we assume we assume *pd* is 0 or 1, The specification for this problem is as follows:

Module FC2Lossy
  State
    $bufA, bufB, outA, outB : Integer$
  State Invariant
    $0 \leq bufA \land 0 \leq outA \land 0 \leq bufB \land 0 \leq outB$
  Transitions
    $Update([uA, uB], [pi]) \triangleq$
      $-1{\leq}uA{\leq}1 \land -1{\leq}uB{\leq}1 \land m{\leq}pi{\leq}n \land 0{\leq}pd{\leq}1\land$
      $bufA' = bufA + pi - outA\land$
      $outA' = outA + uA\land$
      $bufB' = bufB + outA - outB - pd\land$
      $outB' = outB + uB$
  Required Properties
    $bufA = 0\land$
    $outA = 0\land$
    $bufB = 0\land$
    $outB = 0\land$
    $\Box\ 0{\leq}bufA \land bufA{\leq}20 \land 0{\leq}outA \land outA + uA{\leq}4\land$
    $0{\leq}bufB \land bufB{\leq}20 \land 0{\leq}outB \land outB + uB{\leq}4$
End Module

(Note that *bufA* and *bufB* need separate controls *uA* and *uB).* Unfortunately, on this example, our implementation times out.

4.2 Handling many variables

The reason for the timeout in Example 4.4 is that Z3's quantifier elimination procedure struggles on large expressions containing many variables. For example Fig. 2 shows part of the symbolic expression for WCP on the 6th iteration To address this, we have modified our approach as follows. Instead of asking the SMT solver for a solution to formula 2 once Alg. 1 converges, we ask the solver to supply a solution at the beginning. We now use these concrete values of m and n in Alg 1 to compute the new invariant. These represent the widest range of possible values for e under the current (weak) invariant. if Alg. 1 converges with a non false invariant using these values then we are done. Typically, however, the algorithm will produce an invariant that is too strong (False). Intuitively, this is because the initial values of m and n allow the environment too much latitude, and there is no starting state from which the invariant can be satisfied. In this case, we reduce the size of the interval $m..n$ and repeat the process until either convergence or there is no solution even with the smallest possible interval (1 element). In the case that one of the bounds m or n is known (e.g. $m = 0$) we can use standard binary search to find the largest possible value of n. However, when both bounds are variable, we resort to a modified form of binary search to find the largest bound. This is shown in the pseudo Python code outline in Alg. 2. A brief description of Alg. 2 is as follows: getBounds() first asks the SMT solver for a solution to constraint 2 using the initial invariant (see initial L in Alg. 1). This produces the most lenient bounds possible (since the initial node label is the weakest). (In the packet buffer example this is 0..20). The environment constraint (*ePred* i.e. $m \leq e \leq n$) is instantiated with these bounds and Alg. 1 is run as normal with these initial bounds. If the bounds are feasible then Alg.1 terminates with success and a new invariant which is used to infer the bounds . If not, calcBounds is called with the current initial interval bisected. calcBounds calls 1 to see if a node label can be determined with these bounds and if so, now tries to grow it, if possible, towards the end of the original bound (the new upper bound is $(u + U)\ div\ 2$). If the current interval is already as large as possible, then it exits with the current bounds. Otherwise, what happens next depends on what phase the algorithm is in. There are two phases: a "growing" phase and "shrinking" (non-growing) phase. Initially calcBounds starts out in the shrinking phase and stays there as it as long as it is reducing the interval to find one that is satisfiable. This is the first

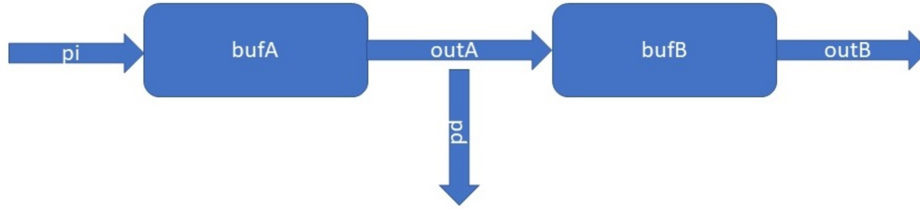

Figure 1. Two packet buffers in series

```
ForAll([pi, pd], Implies(And(m <= pi, pi <= n,
pd >= 0, pd <= 1), ForAll([bufAX, bufBX, outAX,
outBX], Implies(And(bufAX == bufA + pi - outA,
outAX == outA + u, bufBX == bufB + outA - outB -
pd, outBX == outB + u), And(Or(Not(-1*n + m <=
0), Not(-6*outBX + 6*outAX + bufBX <= 5)), 2*n +
-2*m <= 11, Or(Not(-1*n + m <= 0), 6*n + -6*outAX
+ bufAX + 2*outBX <= 34),...(26 more disjuncts)
```

Figure 2. Small part of the symbolic expression for WCP in Packet Flow Control

case in the code (that checks for $\neg growing$), hence the new upper bound is $(l + u)$ div 2. The growing phase begins as soon as a feasible range has been found. Once in the growing phase, CALCBOUNDS remains in that phase, and even when the current bounds fail it only reduces the interval size back towards the last known good upper bound (ie the new upper bound is $(old_u + u)$ div 2). Using this approach we are now able to solve the problem presented in Example 4.4. The result is that pi must lie between 2 and 4.

5 Related Work

There has been a significant amount of work on inferring the environment assumptions on a system that is composed of smaller components going back to Pnueli [8] and more recently the work of Cobleigh et al. [4]. Cobleigh et al. in particular propose a framework in which component based validation can be carried out automatically. To show that a system composition meets its requirement, they first infer an assumption required on the main (or one of the) component(s) for it to meet the requirement (Step 1) and then verify that that assumption is also satisfied by the other components (Step 2). By representing the assumption as a labeled transition system, it can be incrementally constructed by an iterative DFA learning algorithm, L*. We plan to investigate whether we can use our approach to infer the assumption on the component in Step 1.

Regarding the construction of the E predicate, one possibility we considered was inferring E along with the state

ALGORITHM 2: Interval Refinement Algorithm

```
def getBounds():
 #ask SMT solver for soln to (2) using initial L
 (M,N) = findInitialBounds()
 ePred = substitute(ePred, [(m,M), (n,N)])
 (success?,L') = findInvariant(ePred) #call Alg. 1
 if success?:
     (M,N) = inferBoundsFinal(L')
 else:
     (M,N) = calcBounds(M, (M+N) div 2, N, -maxint,
 False)
 return (M,N)

#get the widest possible bounds wrt given inv.
def inferBoundsFinal(L):
 exp = L(m,n) ∧ ∀s,m2,n2: L(s,m2,n2)⟶n2-m2 ≤n-m
 s = SolverFor("LIA")
 s.add(exp)
 if s.check() == sat:
     model = s.model()
     return (model[m], model[n])
 else:
     raise exception("Unable to infer bounds for
 invariant")

def calcBounds(l, u, U, old_u, growing):
 ePred = substitute(ePred, [(m,l), (n,u)])
 (success?,L') = findInvariant(ePred)
 if success: #soln exists with curr bounds, try to
 extend
     if u < U-1:
         u_mid = (u+U) div 2
         calcBounds(l, u_mid, U, u)
     else:
          return (l,u)
 else if u > l: #no soln, need to reduce interval
 size
      if ¬growing:
          u_mid = (l + u) div 2
          calcBounds(l, u_mid, u, -maxint, False)
      else:
          u_mid = (old_u + u) div 2
          calcBounds(l, u_mid, u, old_u, True)
 else print("no valid bounds from ", l)
```

invariant L. In this regard, Dillig et al [6] use abductive reasoning to infer loop invariants. Their approach is quite powerful and seems appealing but will lead to an E that depends on the current state. This will not work for us as the environment represents the uncontrollable aspect so it cannot be made dependent on the current state. One question that arises is whether we can synthesize the E predicate as some Boolean combination of linear constraints, as opposed to the fixed linear range that is currently supported. Horn Clause solvers [2] are a potential solution. We plan to experiment with such solvers in future work.

Acknowledgments

This work has been sponsored in part by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration, by the NSF under contract , and ONR under contract. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] T. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. 2014. A Constraint-Based Approach to Solving Games on Infinite Graphs. In *Symp. on Principles of Programming Languages (POPL)*.

[2] Nikolaj BjÃžrner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. *Horn Clause Solvers for Program Verification.* 24–51.

[3] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Saar. 2012. Synthesis of Reactive(1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938. https://www.sciencedirect.com/science/article/pii/S0022000011000869 In Commemoration of Amir Pnueli.

[4] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin Heidelberg, 331–346.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 238–252.

[6] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. *SIGPLAN Not.* 48, 10 (oct 2013), 14 pages. https://doi.org/10.1145/2544173.2509511

[7] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.

[8] Amir Pnueli. 1985. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, Krzysztof R. Apt (Ed.). Springer Berlin Heidelberg, 123–144.

[9] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of an Asynchronous Reactive Module. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP '89)*. Springer-Verlag, 652–671.

[10] Jakob Rehof and Torben Mogensen. 1999. Tractable Constraints in Finite Semilattices. *Science of Computer Programming* 35 (1999), 191–221.

[11] Matteo Slanina, S. Sankaranarayanan, Henny Sipma, and Zohar Manna. 2007. *Controller Synthesis of Discrete Linear Plants Using Polyhedra.* Technical Report. Technical Report REACT-TR-2007-01, Stanford University.

[12] D.R. Smith and S. Nedunuri. 2021. *Model Refinement.* Tech. Rep. 21.0. Kestrel Institute. https://www.kestrel.edu/people/smith/pub/mr-tr.pdf

[13] Eduardo Sontag. 1998. *Mathematical Control Theory.* Springer.