Model Refinement (Extended) [⋆]

Douglas R. Smith[1] and Srinivas Nedunuri[2]

[1] Kestrel Institute, Palo Alto, California 94304 USA
`smith@kestrel.edu`
[2] Sandia National Laboratories, Livermore, California 94550 USA
`snedunu@sandia.gov`

7 September 2023

Abstract. Model Refinement is a uniform approach to generating correct-by-construction designs for algorithms and systems from formal specifications. Given an overapproximating model $\mathcal{M}$ of system dynamics and a set Φ of required properties, model refinement is an iterative process that eliminates behaviors of $\mathcal{M}$ that do not satisfy the required properties. The result of model refinement is a refined model $\mathcal{M}'$ that satisfies by-construction the required properties Φ. The calculations needed to generate refinements of $\mathcal{M}$ typically involve quantifier elimination and extensive formula/term simplification modulo the underlying domain theories. This paper focuses on the enforcement of basic safety properties in the form of state and action invariants. Several extensions of the safety language are presented and normalization rules are given to reduce them to basic safety properties. We have run a prototype implementation of model refinement based on the Z3 SMT solver over a variety of system and algorithm design problems.

1 Introduction

Program synthesis is the process of transforming a formal specification of requirements to a program that provably satisfies the specification. Historically, program synthesis stems from logical investigations into the connection between provability of formulas of the form

$$\forall x\, \exists z\, \varphi(x, z) \qquad (1)$$

and computable functions; prominently, Kleene's 1945 work on realizability [26]. In modern terms, we say that a witness to the existential in (1) is a computable function f such that $\forall x.\, \varphi(x, f(x))$. The development of resolution and constructive proof tools in the 1960's led computer scientists to develop techniques for extracting functional programs from proofs of (1) [22, 13, 29] . Current support for program extraction from proofs is typically provided in proof environments for constructive logics such as NuPRL [14] and Coq [1].

⋆ This work has been sponsored in part by NSF under contract CCF-0737840, ONR under contract N00014-04-1-0727, and 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 under contract DE-NA0003525. 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.

Going beyond synthesis of functions, in 1957 Church proposed the problem of synthesizing nonterminating computations that react to a stream of inputs from the environment [10, 11] as in digital circuits. This corresponds to generalizing formula (1) to an infinite alternation of quantifiers:

$$\forall x_0 \, \exists x_1 \, \forall x_2 \, \exists x_3 \, \cdots \varphi(x_0, x_1, x_2, x_3, \cdots) \tag{2}$$

Formula (2) is naturally interpreted as a game between the program and its environment with winning condition φ. For every move or choice x_0 that the environment makes, there is choice x_1 that the program can make, such that for every move or choice x_2 that the environment makes, ... such that the formula φ holds, in which case the program wins, otherwise the environment wins. Solutions to Church's problem [7, 33] established a double exponential bound on the complexity of finding a nonterminating program (now called a *reactive system*) for certain classes of games with a finite state space. More recent research focuses on classes of game-like specifications for which the synthesis process has a lower worst-case complexity [4, 3, 6].

Formal approaches to program synthesis start with a logical specification essentially of the form (1) or (2) which decides the desired behaviors of a program. The essence of many synthesis techniques is to eliminate undesired behaviors from a model whose behaviors overapproximate the set of desired behaviors. In this paper we propose a unifying framework, called *model refinement*, for specifying such overapproximating models, together with a constraint system whose solution corresponds to the elimination of undesired behaviors from the model. The framework serves to unify and extend previous work on function/algorithm synthesis with reactive system synthesis. Given a model $\mathcal{M}$ that overapproximates system behaviors and a set Φ of required properties, the goal of model refinement is to generate the least refinement $\mathcal{M}'$ of model $\mathcal{M}$ such that $\mathcal{M}'$ satisfies the specified properties Φ. If the set of legal initial states in $\mathcal{M}'$ differs from the initial states of $\mathcal{M}$, then the difference characterizes the set of initial states from which the system does not have a winning strategy. Model checking [12] is the special case in which refinement of the model is not an option.

Overapproximating models can arise in a variety of ways. For control system problems, the model captures the dynamics of a physical asset (aka the "plant") to be controlled. In information system design, the model captures the APIs and possible operations of a component and perhaps a restricted grammar for expressing programs [2]. In general system design, a model can express a system design pattern [19, 9, 36]. In algorithm design, a model can reflect the imposition of a parametric solution pattern, such as an algorithm theory [44] or a sketch [48].

This paper focuses on enforcement of basic safety properties. In later sections, we introduce a wider fragment of temporal logic that can be reduced to the basic safety fragment. We choose to model the state space logically, which enables representing and reasoning about large or infinite state spaces. Most current work on the synthesis of reactive systems focuses on circuit design and starts with specifications in propositional Linear Temporal Logic (LTL) [6, 24]. Model Refinement allows specifications that are first-order and uses a temporal logic of action that is amenable to refinement, which LTL is not, allowing a broader range of applications to be tackled.

Model refinement is intended to support highly automated refinement-generating tools that produce correct-by-construction designs together with machine-checkable proofs. The essential barrier to full automation is the computational complexity of formula simplification in the application domain theories that support the system specification. When the domain theories are decidable (e.g. by

SMT solvers) and admit quantifier elimination, then model refinement can run fully automatically. We have used our SMT-based prototype to perform model refinement on a variety of examples.

Our contributions include

1. a uniform framework for specifying algorithms and reactive systems by a combination of over-approximating behavioral models and logical specifications of required behavior,
2. a characterization of model refinement via a system of definite constraints that can be efficiently solved by fixpoint-iteration procedures,
3. a variety of examples to show the breadth of the technique,
4. a prototype implementation based on the Z3 SMT-solver [43].

Motivating Example A secure enclave has a door whose latch is controlled by a card reader. A user can Insert or Remove a card from the reader. The system controls the latch and can perform Lock or Unlock actions. When unlocked, the Door can be opened. A model for this enclave has actions or transitions for Insert and Remove a card from the reader (we ignore the aspect of actually authenticating the card). It also has actions for Locking and Unlocking the door. In this simple initial model, many behaviors are allowed, such as the door Unlocking without any card in the reader. As such the model provides a superset of the behaviors that we desire. We specify those desired behaviors by means of safety properties: (1) whenever an Insert action occurs, then the door must Unlock (within k time units), (2) whenever an Unlock action occurs, then there must have been an Insert action (within the previous k time units). The intent of model refinement is to strengthen the four actions of the model so that only behaviors satisfying the two global properties hold. This example is worked in detail in Section 5.2.

We first introduce model refinement over basic safety properties. We then show how safety properties that are expressed using bounded-time past and future temporal operators (Section 5.1) and path properties (Section 7) can be reduced to basic safety properties. Each of our examples runs in a few seconds on our prototype Z3-based model refinement tool.

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 [27]. A *state* is a (type-consistent) map from variables to values. *State predicates* are boolean expressions formed over the variables of a state and the constants (including functions) relevant to an application domain. A state predicate p denotes a relation $[\![p]\!]$ over states, so $p(s)$ denotes the truth value $[\![p]\!](s)$ for state s. *Actions* are boolean 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 or $\Box\varphi$ where φ is a state predicate or an action. The truth of a safety property φ at position n of a trace σ (an infinite sequence of states), written $\sigma, n \models \varphi$, is defined as follows:

– $\sigma, n \models p$, for p a state predicate, if p holds at state $\sigma[n]$, i.e. $[\![p]\!](\sigma[n])$;
– $\sigma, n \models a$, for a an action, if a holds over the states $\sigma[n], \sigma[n+1]$, i.e. $[\![a]\!](\sigma[n], \sigma[n+1])$;
– $\sigma, n \models \Box\varphi$ if $\sigma, i \models \varphi$ for all $i \geq n$.

2.2 Behavioral Models

Formally, a model is a *labeled control flow graph* (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: a finite set of nodes. Associated with each node $m \in N$, we have a finite subset of observable 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.
– A: a finite set of directed arcs, $A \subseteq N \times N$. Each node m has 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 predicate over $V(m)$ representing a node invariant. For each arc $a = \langle m, n \rangle$, label $L_a \in \mathcal{L}$ is an action over $V(m)$, $V(n)$, and auxiliary variables e and u which are discussed below.

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.

A *state* st_m at node m is a type-consistent map from $V(m)$ to values. To simplify notation, we often write $L_m(st_m)$ to denote $st_m \models L_m(V(m))$ (and similarly for arc labels). 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 nondeterministic action, whose nondeterminism may be reduced under refinement. In reactive systems, which have a game-like character, some of the nondeterminism 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 nondeterminism are refinable and which are unrefinable. 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. e is treated as an uncontrollable environment or adversary input that ranges over the unrefinable set $E_a(st_m)$;
2. u is treated as a controllable value that satisfies the refinable constraint $U_a(st_m, u)$;
3. function f_a gives the deterministic response of the action.

The variability of the control value 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 [49]. Let

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

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

We specify an action $a = \langle m, n\rangle$ via a predicate over $V(m) \times V(n)$ that denotes a transition relation

$$[\![a]\!]_{\mathcal{M}} = \{\langle st_m, st_n\rangle \mid L_a(st_m, st_n)\} \subseteq \mathcal{S}_m \times \mathcal{S}_n.$$

When the LCFG is clear from context we omit the subscript.

An action with a degenerate sytem input is called an *environment action* since it is triggered and instantiated solely by the environment. Similarly for *system actions*. In some models it is possible to extract an entire Environment LCFG consisting of environment actions that models the behaviors of the Environment treated as an agent or process. See for example the Card Reader model in Section 5.2.

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. $st_{i+1} \in [\![n]\!]$.

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

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.

2.3 Specification and Refinement

Refinement of LCFG model $\mathcal{M}_1$ to model $\mathcal{M}_2$ is a preorder relation, written $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$, that holds when there exists a *simulation map* $\xi : \mathcal{M}_2 \rightarrow \mathcal{M}_1$ that maps the nodes and arcs of $\mathcal{M}_2$ to the nodes and arcs of $\mathcal{M}_1$; i.e. where $\xi : N^{\mathcal{M}_2} \rightarrow N^{\mathcal{M}_1}$ and $\xi : A^{\mathcal{M}_2} \rightarrow A^{\mathcal{M}_1}$ such that

1. Initial nodes are preserved: $\xi(m_0^{\mathcal{M}_2}) = m_0^{\mathcal{M}_1}$;
2. Observable variables: $V^{\mathcal{M}_2}(m) \supseteq V^{\mathcal{M}_1}(\xi(m))$ for each node $m \in N^{\mathcal{M}_2}$;
3. Node labels: $L_m^{\mathcal{M}_2} \implies L_{\xi(m)}^{\mathcal{M}_1}$ for each node $m \in N^{\mathcal{M}_2}$;
4. Arc labels: $L_a^{\mathcal{M}_2} \implies L_{\xi(a)}^{\mathcal{M}_1}$ for each arc $a \in A^{\mathcal{M}_2}$.

There are several kinds of transformations of models that generate refinements, including (1) strengthening the invariant at a node, and (2) strengthening the action at an arc. These are used in the model refinement procedure in the next section. A third transformation, *structure refinement*, replaces an arc by an arc-like LCFG. This transformation may be used when imposing a design pattern or program scheme as a constraint on how to achieve the action of the arc. An example of this is given in Section 7.1.

A *specification* $\mathcal{S} = \langle \mathcal{M}, \Phi\rangle$ is comprised of a model $\mathcal{M}$ and a set of properties Φ that we require to incorporate or enforce in $\mathcal{M}$. A specification denotes the set of traces generable by $\mathcal{M}$ that also satisfy all properties in Φ:

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

Refinement of specification $\mathcal{S}$ to specification $\mathcal{T}$ is a preorder relation, written $\mathcal{S} \sqsubseteq \mathcal{T}$, that holds when there is a mapping ξ from traces of $\mathcal{T}$ to traces of $\mathcal{S}$ such that

$$\forall \sigma . \sigma \in [\![\mathcal{T}]\!] \implies \xi(\sigma) \in [\![\mathcal{S}]\!]$$
or more succinctly $\xi([\![\mathcal{T}]\!]) \subseteq [\![\mathcal{S}]\!]$.

Theorem 1. If
1. $\mathcal{S}_1 = \langle \mathcal{M}_1, \Phi_1 \rangle$ and $\mathcal{S}_2 = \langle \mathcal{M}_2, \Phi_2 \rangle$ are specifications,
2. $\xi : \mathcal{M}_2 \rightarrow \mathcal{M}_1$ is a simulation map
3. $\Phi_2 \implies \Phi_1$

then $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$.

Proof: The simulation map ξ from $\mathcal{H}$ to $\mathcal{G}$ defines a simulation relation so we can show that every trace of $\mathcal{H}$ maps to (or simulates) a trace of $\mathcal{G}$, as illustrated in Figure 1. Effectively, $\mathcal{H}$ simulates the observable behavior of $\mathcal{G}$ with no more nondeterminism.

$$\begin{tikzcd}
\sigma : & m_0 \arrow[r, dotted] & m \arrow[r, "a"] & m' \arrow[r, dotted] & {} \\
\sigma' : & h_0 \arrow[r, dotted] \arrow[u, "\xi"] & h \arrow[r, "b"] \arrow[u, "\xi"] & h' \arrow[r, dotted] \arrow[u, "\xi"] & {}
\end{tikzcd}$$

Fig. 1: Simulating a trace

Given a trace σ' of $\mathcal{H}$, we show how to construct a trace σ of $\mathcal{G}$. If h_0 is the start node of $\mathcal{H}$, then $\sigma'[0] \in [\![h_0]\!]$. By construction, $\xi(h_0) = m_0$ where m_0 is the start node of $\mathcal{G}$, so we can construct a start state in $[\![m_0]\!]$ by simply forgetting/eliminating those variables of h_0 that are not in m_0 (since $V_{\mathcal{H}}(h_0) \supseteq V_{\mathcal{G}}(\xi(h_0)) = V_{\mathcal{G}}(m_0)$; i.e.

$$\sigma[0] = \{ v \mapsto val \mid v \in V(m_0) \wedge val = \sigma'[0](v) \}.$$

$\sigma'[0] \models L_{h_0}$

$\implies \quad \sigma'[0] \models L_{m_0}$ — since $L_{h_0} \Rightarrow L_{\xi(h_0)} \iff L_{\xi(m_0)}$

$\implies \quad \sigma[0] \models L_{m_0}$ — retracting the model to just the variables of m_0

Inductively, consider the transition $\sigma'[i] \rightarrow \sigma'[i+1]$ where $\sigma'[i] \in [\![h]\!]$ $(\sigma'[i] \models L_h)$ and $\xi(h) = m$. Let $\sigma[i]$ be the state constructed from $\sigma'[i]$ by forgetting of irrelevant variables. Let $b = \langle h, h' \rangle$ be the $\mathcal{H}$ arc such that $\sigma'[i+1] \in [\![h']\!]$. Let $\xi(b) = a = \langle n, n' \rangle$, then since b was enabled in state $\sigma'[i]$ we have

$\langle \sigma'[i], \sigma'[i+1] \rangle \models L_b$

$\implies \quad \langle \sigma'[i], \sigma'[i+1] \rangle \models L_a$ — since $L_b \Rightarrow L_{\xi(b)} \iff L_a$

$\equiv \quad \langle \sigma[i], \sigma[i+1] \rangle \models L_a$ — retracting the models to just the variables of m and n

So, the arc a is enabled in state $\sigma[i]$ and thus $\sigma[i+1]$ is part of a legal trace of $\mathcal{G}$. This shows that $\xi([\![\mathcal{M}_2]\!]) \subseteq [\![\mathcal{M}_1]\!]$. To show specification refinement:

$$\begin{array}{lll}
\xi(\llbracket \mathcal{S}_2 \rrbracket) & & \\
\quad = & \xi(\llbracket \mathcal{M}_2 \rrbracket \cap \llbracket \Phi_2 \rrbracket) & \text{definition} \\
\quad = & \xi(\llbracket \mathcal{M}_2 \rrbracket) \cap \xi(\llbracket \Phi_2 \rrbracket) & \text{distributing} \\
\quad \subseteq & \llbracket \mathcal{M}_1 \rrbracket \cap \llbracket \Phi_2 \rrbracket & \text{since } \xi(\llbracket \mathcal{M}_2 \rrbracket) \subseteq \llbracket \mathcal{M}_1 \rrbracket \\
\quad \subseteq & \llbracket \mathcal{M}_1 \rrbracket \cap \llbracket \Phi_1 \rrbracket & \text{assumption that } \Phi_2 \Rightarrow \Phi_1 \\
\quad = & \llbracket \mathcal{S}_1 \rrbracket. & \text{definition}
\end{array}$$

That is, $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$. QED

3 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 later sections we characterize several situations in which only a near-greatest solution can be found.

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.

We characterize the model refinement transformation by a three-stage constraint system. The first stage enforces general behavioral constraints, and the second stage enforces initial state constraints and frame constraints that arise from protected variables.

wcp is the *weakest controllable predecessor* (WCP) predicate transformer and is defined by

$$wcp(L_a, L_n) \equiv \forall e.\, e \in E(st_m) \implies \exists st_n.\, st_n = f_a(st_m, e, u) \wedge L_n(st_n)$$

or, simply

$$wcp(L_a, L_n) \equiv \forall e.\, e \in E(st_m) \implies L_n(f_a(st_m, e, u))$$

where L_n is a state predicate. *wcp* is the weakest formula over $V(m) \bigcup \{u\}$ such that for any environment input e the transition a is assured to reach a state st_n satisfying the post-state predicate L_n. Its effect is to define the nonblocking states at node m – those states from which there is some control value that forces the transition to a legal state at n regardless of the environment input.

Stage 0.
0. **State Initialization:** Let m_0 be a solution to the constraint-satisfacton 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:** $L_m \implies \varphi$ for each node $m \in N$ if φ is a state predicate expressed over the variables at m;

2. **Arc Localization:** $L_a \implies \varphi$ for each arc $a = \langle m, n \rangle \in A$ if φ is an action expressed over the variables at m and n;
3. **Control Constraint:** $U_a \implies wcp(L_a, L_n)$ for each arc $a = \langle m, n \rangle$
4. **Node Invariant:** $L_m \implies \bigvee_{a=\langle m,n \rangle} \exists u.\, U_a$ for each node $m \in N$.

Stage 2.
5. **Variable Protection:** $L_a \implies unchanged(v)$ for each arc $a = \langle m, n \rangle \in A$ in which there is no mention of v' for protected $v \in V$ in L_a.

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, and then further refines it by solving the stage 2 constraints.

The Localization 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 state transitions in which the environment can force the system to a state not satisfying the safety properties. The Node Invariant constraints (4) serve to eliminate blocking states at a node with respect to all of its outgoing arcs.

```
if $\varphi$ is a state predicate
    then for $m \in N:\ L_m \leftarrow L_m \wedge \varphi$
    else for $a \in A:\ L_a \leftarrow L_a \wedge \varphi$
do
    for $a \in A:\ U_a \leftarrow U_a \wedge wcp(L_a, L_n)$
    for $m \in N:\ L_m \leftarrow L_m \wedge \bigvee_{a=\langle m,n \rangle} \exists u.\, U_a$
until $L_m$ is unchanged for all nodes $m \in N$.
```

Fig. 2: Model Refinement Algorithm

A straightforward algorithm for solving the constraint system over the labels on a model is presented in Figure 2. The iteration converges to a fixpoint when the labels do not change in an iteration. Upon convergence to a refined model $\mathcal{M}'$, we have $[\![\mathcal{M}']\!] \subseteq [\![\mathcal{M}]\!] \cap [\![\Phi]\!]$, and in the case that the algorithm converges to a greatest fixpoint we have $[\![\mathcal{M}']\!] = [\![\mathcal{M}]\!] \cap [\![\Phi]\!]$.

The constraints have definite form[3] and the algorithm in [35] provides a more efficient control strategy that exploits dependencies between the constraints.

The *derived initial condition* is the final refined invariant L_{m_0} which characterizes the set of non-blocking initial states from which the system can ensure that all behaviors satisfy the specified safety properties. 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 counterexamples produced by the model-checker.

Correctness and Complexity

The correctness of this algorithm is a consequence of Tarski's theorem. Each constraint has definite form $v \leq g(v)$, so we can express solutions as fixpoints of $v = g(v)$. As we are looking for the most general (i.e. least refinement of the initial model), the algorithm aims to converge on the greatest fixpoint using a Kleene iteration.

[3] A constraint over a meet semilattice is *definite* or Horn-like if has the form $v \leq g(v)$ for variable v and monotone function g.

If the state space is finite, then the fixpoint iteration process will be finite too. In fact, the number of iterations is linear in the height of the lattice.

There are several challenges that arise in solving the constraint system. First, to aid convergence and improve performance, it is necessary to aggressively simplify expressions at each step. Each iteration generates instances of *wcp* with its universal quantification over environment inputs. Formula simplification techniques, especially quantifier elimination, are needed to eliminate redundancy and keep the intermediate forms of the labels as compact as possible. Second, forcing termination in a fixpoint iteration algorithm is addressed by the concept of widening from abstract interpretation [15]. When computing a greatest fixpoint, the idea is to underapproximate the bounds in the constraint system resulting in a fixpoint that underapproximates the greatest fixpoint. In terms of model refinement, an underapproximation would result in a possibly stronger derived initial condition; that is, it would operate safely but only from a subset of initial states. We can address the challenges of convergence and quantifier elimination by underapproximating a universally quantified formula from *wcp*. The operator $\mathbb{A}\, x.p(x)$ results in the weakest quantifier-free formula such that $\mathbb{A} x.p(x) \implies \forall x.p(x)$ with respect to background theory T. When T admits quantifier elimination, then we have $\mathbb{A}\, x.p(x) \;\equiv\; \forall x.\, p(x)$, otherwise it underapproximates the quantified formula. Replacing $wcp(L_a, L_n)$ in the Control Constraint by $\mathbb{A}\, st_n.wcp(L_a, L_n)$ in the Basic Safety Constraint System provides the possibility of (1) greater formula simplification capability, and (2) more rapid convergence, particularly when fragments of the background domain theory do not admit (tractable) quantifier elimination. The tradeoff is that the resulting model may underapproximate the greatest fixpoint model. Loosening the requirement that $\mathbb{A}$ yields the weakest sufficient condition would further increase the range of applicability of this approach. The techniques of abductive inference [32] and directed inference [38] aim to find a simplest and weakest possible sufficient condition on a given formula. The fact that many theories do not admit quantifier elimination in general and the computational complexity of elimination algorithms has motivated several efforts to define inexpensive underapproximations and overapproximations to quantified formulas, e.g. [23]. Finally, the problem of detecting equivalence between two formulas is, of course, undecidable in first-order and higher-order logics. Practically, we restrict our examples to the decidable theories of current SMT solvers, some of which admit quantifier elimination. More generally, tactic-driven interactive provers/calculators may be necessary to support model refinement.

Example: Packet Flow Control

In this example, based on [37], a buffer is used to control and smooth the flow of packets in a communication system. We model this problem as in discrete control theory with a plant (a buffer of length *buf*), environment/disturbance input e, and control value u. The environment supplies a stream of packets that varies up to 4 packets per time unit. The 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.

This is a classical discrete control problem with a single node and a single linear transition. It can be specified by the following TLA-like notation for an LCFG, which lists the one node with its state variables and their initial invariant, the one arc and its initial action (dependent on environment input e and control value u), and the required safety properties.

Specification FC0
  Node: m_0
    vars: $buf, out : Integer$
    invariant: $0 \leq buf \wedge 0 \leq out$
  Arc: $a = \langle m_0, m_0 \rangle$
    action: $Update(u, e) \triangleq -1 \leq u \leq 1 \wedge 0 \leq e \leq 4 \wedge buf' = buf + e - out \wedge out' = out + u$
  Required Properties
    $buf = 0$
    $out = 0$
    $\square\ 0 \leq buf \wedge buf \leq 20 \wedge 0 \leq out \wedge out \leq 4$
End Specification

The first two required properties set the initial state values. For the last required property, the algorithm in Figure 2 instantiates *wcp* to generate the following formula as an upper bound on the control condition $U(buf, out, u) \equiv -1 \leq u \leq 1$

$$\forall e.\ 0 \leq e \leq 4 \implies 0 \leq buf + e - out \leq 20 \wedge 0 \leq out + u \leq 4.$$

This formula is in the language of integer linear arithmetic which admits quantifier elimination and our Z3-based prototype simplifies it to the equivalent of

$$1 \leq buf - out \leq 16 \wedge 0 \leq out + u \leq 4.$$

According to the algorithm in Figure 1, the control condition $U(buf, out, u)$ strengthens to

$$-1 \leq u \leq 1 \wedge 1 \leq buf - out \leq 16 \wedge 0 \leq out + u \leq 4$$

and the state invariant strengthens to

$$0 \leq buf \wedge 0 \leq out \wedge 1 \leq out - buf \leq 16.$$

Next, our prototype simplifies the control condition with respect to the strengthened state invariant, and the control condition becomes

$$-1 \leq u \leq 1 \wedge 0 \leq out + u \leq 4.$$

Since the control condition for the sole transition has changed, the iteration continues. For this problem convergence happens after four iterations and generates the following refined model, in which the required properties are enforced by-construction and so they are theorems of the model (as can be checked by a model checker).

Specification FC1
  Node: m_0
    vars: $buf, out : Integer = 0$
    invariant: $0 \leq out \leq 4 \wedge 0 \leq buf - out \leq 16 \wedge -3 \leq buf - 3 * out \leq 11 \wedge -6 \leq buf - 4 * out \leq 10$
  Arc: $a = \langle m_0, m_0 \rangle$
    action: $Update(u, e) \triangleq -1 \leq u \leq 1 \wedge\ 0 \leq out + u \leq 4 \wedge\ -6 \leq buf - 4 * u - 5 * out \leq 6$
      $\wedge\ -1 \leq buf - 2 * u - 3 * out \leq 9$
      $\wedge\ 0 \leq e \leq 4 \wedge\ buf' = buf + e - out \wedge\ out' = out + u$
  Theorems
    $buf = 0$
    $out = 0$
    $\square\ 0 \leq buf \wedge buf \leq 20 \wedge 0 \leq out \wedge out \leq 4$
End Specification

The strengthened state invariant on node m_0 is also the derived initial condition and specifies the set the initial states from which we have assurance that the system will keep within the required bounds regardless of environment inputs.

The refined transition now defines a somewhat complex polyhedron around the control values. If there are no more required properties to enforce, then the next step will be to synthesize a control function that selects a specific control value u in each given state. This is also known as extracting a winning strategy for the system game modulo the derived initial conditions.

The version of this problem in which the variables are Reals or Rationals, with an infinite state space, is also solved in a small number of iterations in a few seconds, with a different invariant polytope and derived initial condition defining the safe operating space.

4 Synthesis of Control Functions

In the Flow Control example, model refinement generates a specification for the control constraint $U_a(st, u)$ at each arc a, for state st and control variable u. In words, if the system is in state st (s.t. $L(st)$), and the system chooses control value u such that $U_a(st, u)$, then the arc a is enabled with assurance that taking the transition results in a safe state.

In the Flow Control example, we have the control constraint

$Control(\langle buf, out\rangle, u) \triangleq\ -1{\leq}u{\leq}1 \wedge\ 0{\leq}out + u{\leq}4$
$\wedge\ -6{\leq}buf{-}4{*}u{-}5{*}out{\leq}6 \wedge -1{\leq}buf{-}2{*}u{-}3{*}out{\leq}9$

The control constraint specifies the control function, which has the generic form

$$controlFun(st \mid L(st)) = \{u \mid U_a(st, u)\}.$$

with input condition $L(st)$ and input/output condition $U_a(st, u)$. Implicitly this is formula 1 from Section 1:

$$\forall st.\ L(st) \implies \exists u.\ U(st, u).$$

Algorithm or function synthesis is appropriate for this specification, since the behaviors are specified by a simple input-output relation.

In the Flow Control case, since there are only three possible control values, the synthesis of a case switch function seems appropriate. Our prototype has two variant algorithm synthesis tactics, described next. In outline, given that $\{u \mid U(st, u)\}$ is a finite set $\{u_0, u_1, ...u_{n-1}\}$, the tactic generates the nondeterministic code

```
$controlFun(st \mid L(st)) =$
    **if** $U_a(st, u_0)\ \rightarrow\ u_0$
    $[]\ U_a(st, u_1)\ \rightarrow\ u_1$
    $[]\ \cdots$
    $[]\ U_a(st, u_{n-1})\ \rightarrow\ u_{n-1}$
    **fi**
```

For Flow Control problem, we obtain

```
$controlFun(st \mid L(st)) =$
    if $4{*}out{-}buf{>}{-}5 \ \wedge\ 2{*}out{-}buf{>}{-}12 \ \wedge\ out{\geq}1 \ \wedge\ 5{*}out{-}buf{>}{-}3 \ \rightarrow\ -1$
    [] $2{*}out{-}buf{>}{-}13 \ \wedge\ buf{-}4{*}out{>}{-}4 \ \wedge\ buf{-}2{*}out{>}{-}1 \ \wedge$
        $4{*}out{-}buf{>}{-}8 \ \wedge\ 5{*}out{-}buf + {>}{-}7 \ \wedge\ buf{-}5{*}out{>}{-}7 \ \rightarrow\ 0$
    [] $buf{-}4{*}out{>}{-}1 \ \wedge\ buf{-}2{*}out{>}0 \ \wedge\ buf{-}5{*}out{>}{-}3 \ \wedge\ out \leq 3 \ \rightarrow\ 1$
    fi
```

which can be determinized/simplified by ordering the cases and using context to simplify tests:

```
$controlFun(st \mid L(st)) =$
    if $4{*}out{-}buf > -5 \ \wedge\ 2{*}out{-}buf > -12$
        $\wedge\ out{\geq}1 \ \wedge\ 5{*}out{-}buf > -3$ then $-1$
    else if $buf{-}2{*}out{>}0 \ \wedge\ out{\leq}3$ then $0$
    else $1$
```

The second strategy uses an EF-SMT-solver such as Yices [16]. It uses the SMT solver to find a solution to the EF problem:

$$\exists u.\ \forall st, e.\ L(st) \wedge e \in E(st) \wedge \alpha(st, e, u).$$

This tactic does not find a solution for the Flow Control problem.

5 Property Normalization

The model refinement process defined above enforces basic safety properties. In this section we study two extensions of the property language and how to reduce them to basic safety properties. In Section 5.1 we present a family of refinement-generating transformations that eliminate occurrences of time-bounded operators. In Section 7 we present a refinement-generating transformation that reduces path properties (expressed over a path of arcs in the model) to basic safety properties.

5.1 Properties Expressed Using Time-Bounded Temporal Operators

Time constraints play an essential role in many control programs. We add time to LCFG models by assuming a global variable *start* that records the start time of each state in a trace. Time is assumed to strictly increase without bound along the states of a trace. We leave the action of updating *start* implicit.

For a state or action predicate φ, consider the following time-bounded operators:

1. $\diamondminus_k\, \varphi$ means that φ was true in some state no more than k time units in the past:
   $\sigma, i \models \diamondminus_k\, \varphi$ iff there exists $j < i$ such that $\sigma[i](start) \leq \sigma[j](start) + k$ and $\sigma, j \models \varphi$.
2. $\Diamond_k\, \varphi$ means that φ will be true in some state no more than k time units in the future:
   $\sigma, i \models \Diamond_k\, \varphi$ iff there exists $j > i$ such that $\sigma[j](start) \leq \sigma[i](start) + k$ and $\sigma, j \models \varphi$.
3. $\Box_k\, \varphi$ means that φ holds in each state for the next k time units:
   $\sigma, i \models \Box_k \varphi$ iff for each $j > i$ such that $\sigma[j](start) \leq \sigma[i](start) + k$ we have $\sigma, j \models \varphi$.
4. $\boxminus_k\, \varphi$ means that φ held in each state over the previous k time units:
   $\sigma, i \models \boxminus_k \varphi$ iff for each $j < i$ such that $\sigma[i](start) \leq \sigma[j](start) + k$ we have $\sigma, j \models \varphi$.

We extend the definition of basic safety property as follows: state/action predicates may include predicates of the form $\diamondminus_k\, \varphi$ and $\Diamond_k\, \varphi$ where φ is a state/action predicate.

Transformations to reduce positive occurrences of $\diamondminus_k$, $\Diamond_k$, $\Box_k$, and $\boxminus_k$ are presented next. Negative occurrences of $\diamondminus_k$ and $\Diamond_k$ can be treated by applying the equivalences $\diamondminus_k\, \varphi \equiv \neg\boxminus_k\neg\varphi$ and $\Diamond_k\, \varphi \equiv \neg\Box_k\neg\varphi$, resulting in positive occurrences of $\boxminus_k$ and $\Box_k$. Negative occurrences of $\boxminus_k$ and $\Box_k$ can be treated by applying the equivalences $\boxminus_k\, \varphi \equiv \neg\diamondminus_k\neg\varphi$ and $\Box_k\, \varphi \equiv \neg\Diamond_k\neg\varphi$, resulting in positive occurrences of $\diamondminus_k$ and $\Diamond_k$.

The transformations defined in the next subsections introduce fresh variables to the model and required properties that specify how they evolve. We specify them as *protected* to ensure that they evolve without interference under subsequent refinements. When a variable v is protected, there is a final step to the model refinement transformation that adds the frame or invariance condition *unchanged*(v) (defined by $v' = v$) to all actions that do not mention v'.

4.1.1 Eliminating the Time-Bounded Past Operator $\diamondminus_k$

Transformation 1. Given a specification $\mathcal{S} = \langle M, \Phi\rangle$, transform $\mathcal{S}$ as follows: for each positive occurrence of a subformula of the form $\diamondminus_k\psi$ in Φ:

1.1 add a fresh protected variable $\mathit{lastpsi} : \mathit{Time}$ to $V(m)$ for each $m \in N$, which records the latest time at which ψ held, and add to Φ the new goal formulas

   1.1.1 $\mathit{lastpsi} = -\infty$
   1.1.2 $\Box\, \psi \implies \mathit{lastpsi}' = \mathit{start}$

1.2 replace the occurrence of $\diamondminus_k\, \psi$ with $\mathit{start} \le \mathit{lastpsi} + k$.

Return the modified specification $\mathcal{S}' = \langle M', \Phi'\rangle$.

Theorem 2. If $\mathcal{S}'$ is the result of applying Transformation 1 to specification $\mathcal{S}$, then $\mathcal{S} \sqsubseteq \mathcal{S}'$.

Proof: Since the transformation adds no new nodes or arcs, a simulation map ξ will be the obvious bijection between the old and new nodes and arcs. To apply Theorem 1, it remains to show that $\Phi' \implies \Phi$. Let $\sigma \in [\![\mathcal{M}']\!] \cap [\![\Phi']\!]$ and consider the state $\sigma[i]$ for some $i \ge 0$. If $\sigma[i](\mathit{lastpsi}) = -\infty$ then there has been no prior state that satisfies ψ, so both $(\sigma, i) \models \mathit{start} \le \mathit{lastpsi} + k$ and $(\sigma, i) \models \diamondminus_k\, \psi$ are false. Otherwise, let $j < i$ be the largest index such that $(\sigma, j) \models \psi$, then $\sigma[j+1](\mathit{lastpsi}) = \sigma[j](\mathit{start})$ by the action of property (1.1.2). So we have

$$\begin{aligned}
&\sigma, i \models \mathit{start} \le \mathit{lastpsi} + k \\
&\quad\equiv\quad \sigma[i](\mathit{start}) \le \sigma[i](\mathit{lastpsi}) + k && \text{definition} \\
&\quad\equiv\quad \sigma[i](\mathit{start}) \le \sigma[j+1](\mathit{lastpsi}) + k && \mathit{lastpsi}\text{ is protected and} \\
& && \psi\text{ doesn't hold between indices } j+1 \text{ and } i \\
&\quad\equiv\quad \sigma[i](\mathit{start}) \le \sigma[j](\mathit{start}) + k && \text{by assumption and the action of (1.1.2)} \\
&\quad\equiv\quad \sigma, i \models \diamondminus_k\, \psi. && \text{definition}
\end{aligned}$$

From this inference, we conclude generally that in the context of any trace of $[\![\mathcal{M}']\!] \cap [\![\Phi']\!]$ that the state predicate $\mathit{start} \le \mathit{lastpsi} + k$ holds exactly when the property $\diamondminus_k\, \psi$ holds; i.e. $\mathit{start} \le \mathit{lastpsi} + k \equiv \diamondminus_k\, \psi$. Since we have formed Φ' by modifying positive occurrences of subformulas of the form $\diamondminus_k\psi$, Φ is monotone in those locations, and so we infer $\Phi' \implies \Phi$.

4.1.2 Eliminating the Time-Bounded Future Operator $\Diamond_k$

In the following transformation, we assume a variable $psiDeadlines : Set(Time)$ that represents a set of deadlines and has the operation $min(psiDeadlines)$ which returns the least element of the set, if one exists, otherwise ∞.[4]

Transformation 2. Given a specification $\mathcal{S} = \langle \mathcal{M}, \Phi \rangle$, transform $\mathcal{S}$ as follows: for each positive occurrence of a subformula of the form $\Diamond_k \psi$ in Φ:

2.1 add fresh protected variable $psiDeadlines : Set(Time)$ to each $V(m)$ for node $m \in N$, and add to Φ the new goal formulas:
  2.1.1 $psiDeadlines = \{\}$
  2.1.2 $\Box\, \psi \implies psiDeadlines' = \{\}$
  2.1.3 $\Box\, start \leq min(psiDeadlines)$
2.2 replace the occurrence of $\Diamond_k\, \psi$ in Φ with
  $psiDeadlines' = psiDeadlines \cup \{start + k\}$.

Return the modified specification $\mathcal{S}' = \langle M', \Phi' \rangle$.

Theorem 3. If $\mathcal{S}'$ is the result of applying Transformation 2 to specification $\mathcal{S}$, then $\mathcal{S} \sqsubseteq \mathcal{S}'$.

Proof: Since the transformation adds no new nodes or arcs, a simulation map ξ will be the obvious bijection between the old and new nodes and arcs. To apply Theorem 1, it remains to show that $\Phi' \implies \Phi$. Let $\sigma \in [\![\mathcal{M}']\!] \cap [\![\Phi']\!]$ and consider arbitrary index $p \geq 0$ such that $\sigma[p+1](psiDeadlines) \backslash \sigma[p](psiDeadlines) = \{d\}$ for deadline d. State $\sigma[p+1]$ must have been produced by an action that implies $psiDeadlines' = insert(d, psiDeadlines)$ arising from the replacement of a subformula $\Diamond_k \psi$ where $d = \sigma[p](start) + k$. We now show that $\sigma, p \models \Diamond_k\, \psi$. Let $r > p+1$ be the least index such that $\sigma[r](start) > d$. If $d \in \sigma[r](psiDeadlines)$ then we have $\sigma, r \models start > d \geq min(psiDeadlines)$ which contradicts the state invariant (2.1.3). Consequently, $d \notin \sigma[r](psiDeadlines)$. Therefore d was removed from $psiDeadlines$ at some point $q \in [p{+}2, r{-}1]$. But since $psiDeadlines$ is protected, only the action 2.1.2 can change $psiDeadlines$, specifically setting it to empty. Again by 2.1.2 this means that ψ held at $q-1$, i.e. $\sigma, q-1 \models \psi$. Since $\sigma[q](start) \leq d < \sigma[r](start)$ by choice of r, we have $(\sigma, p) \models \Diamond_k\, \psi$. This shows that whenever the action $psiTimer' = psiDeadlines \cup \{start + k\}$ takes place in an arbitrary trace, then we also have the property $\Diamond_k \psi$; i.e. $psiTimer'{=}psiDeadlines \cup \{start + k\} \implies \Diamond_k \psi$. Since we have only replaced positive occurrences of subformulas of the form $\Diamond_k \psi$, Φ is monotone in those locations, and so we infer $\Phi' \implies \Phi$.

4.1.3 Eliminating the $\boxminus_k$ Operator

Transformation 3. Given a specification $\mathcal{S} = \langle M, \Phi \rangle$, transform $\mathcal{S}$ as follows: for each positive occurrence of a subformula of the form $\boxminus_k \psi$ in Φ:

3.1 add a fresh protected variable $lastnotpsi : Time$ to $V(m)$ for each $m \in N$, which records the latest time at which $\neg\psi$ held, and add to Φ the new goal formulas

[4] While for this specific operator $\Diamond_k$, we could replace the set with just a *nextDeadline* variable, generally a future time requirement will be supported by a priority queue of currently undischarged tasks or obligations, and some kind of scheduler to manage the queue.

3.1.1 $lastnotpsi = -\infty$
3.1.2 $\square \neg\psi \implies lastnotpsi' = start$

3.2 replace the occurrence of $\boxminus_k \psi$ with $start > lastnotpsi + k$.

Return the modified specification $\mathcal{S}' = \langle M', \Phi' \rangle$.

Theorem 4. If $\mathcal{S}'$ is the result of applying Transformation 3 to specification $\mathcal{S}$, then $\mathcal{S} \sqsubseteq \mathcal{S}'$.

Proof: Since the transformation adds no new nodes or arcs, a simulation map ξ will be the obvious bijection between the old and new nodes and arcs. To apply Theorem 1, it remains to show that $\Phi' \implies \Phi$.

Let $\sigma \in [\![\mathcal{M}']\!] \cap [\![\Phi']\!]$ and consider the state $\sigma[i]$ for some $i \geq 0$. If $\sigma[i](lastnotpsi) = -\infty$ then there has been no prior state that satisfies $\neg\psi$, so both $(\sigma, i) \models start > lastnotpsi + k$ and $(\sigma, i) \models \boxminus_k \psi$ are true. Otherwise, let $j < i$ be the largest index such that $(\sigma, j) \models \neg\psi$, then $\sigma[j+1](lastnotpsi) = \sigma[j](start)$ by the action of property (3.1.2). So we have

$$
\begin{array}{lll}
\sigma, i \models start > lastnotpsi + k & & \\
\quad \equiv & \sigma[i](start) > \sigma[i](lastnotpsi) + k & \text{definition} \\
\quad \equiv & \sigma[i](start) > \sigma[j+1](lastnotpsi) + k & lastnotpsi \text{ is protected and} \\
 & & \neg\psi \text{ doesn't hold between indices } j+1 \text{ and } i \\
\quad \equiv & \sigma[i](start) > \sigma[j](start) + k & \text{by assumption and the action of (3.1.2)} \\
\quad \equiv & \sigma, i \models \neg \diamondminus_k \neg\psi & \text{definition} \\
\quad \equiv & \sigma, i \models \boxminus_k \psi. & \text{definition}
\end{array}
$$

From this inference, we conclude generally that in the context of any trace of $[\![\mathcal{M}']\!] \cap [\![\Phi']\!]$ that the state predicate $start > lastnotpsi + k$ holds exactly when the property $\boxminus_k \psi$ holds; i.e.

$$start > lastnotpsi + k \;\equiv\; \boxminus_k \psi.$$

Since we have formed Φ' by modifying positive occurrences of subformulas of the form $\boxminus_k\psi$, Φ is monotone in those locations, and so we infer $\Phi' \implies \Phi$ (or more strongly $\Phi' \equiv \Phi$).

4.1.4 Eliminating the $\square_k$ Operator

In the following transformation, we assume a variable $notpsiESTs : Set(Time)$ that represents a set of Earliest Start Times, and has the operation $max(notpsiESTs)$ which returns the largest element of the set, if one exists, otherwise $-\infty$.

Transformation 4. Given a specification $\mathcal{S} = \langle \mathcal{M}, \Phi \rangle$, transform $\mathcal{S}$ as follows: for each positive occurrence of a subformula of the form $\square_k\psi$ in Φ:

4.1 add fresh protected variable $notpsiESTs : Set(Time)$ to each $V(m)$ for node $m \in N$, and add to Φ the new goal formulas:
4.1.1 $notpsiESTs = \{\}$
4.1.2 $\square \neg\psi \implies notpsiESTs' = \{\}$
4.1.3 $\square\; start \leq max(notpsiESTs) \implies \psi$

4.2 replace the occurrence of $\square_k \psi$ in Φ with
$notpsiESTs' = notpsiESTs \cup \{start + k\}$.

Return the modified specification $\mathcal{S}' = \langle M', \Phi' \rangle$.

Theorem 5. If $\mathcal{S}'$ is the result of applying Transformation 4 to specification $\mathcal{S}$, then $\mathcal{S} \sqsubseteq \mathcal{S}'$.

Proof: Since the transformation adds no new nodes or arcs, a simulation map ξ will be the obvious bijection between the old and new nodes and arcs. To apply Theorem 1, it remains to show that $\Phi' \implies \Phi$. Let $\sigma \in [\![\mathcal{M}']\!] \cap [\![\Phi']\!]$ and consider arbitrary index $p \geq 0$ such that $\sigma[p+1](\mathit{notpsiESTs}) \backslash \sigma[p](\mathit{notpsiESTs}) = \{\mathit{est}\}$ for earliest start time est. State $\sigma[p+1]$ must have been produced by an action that implies $\mathit{notpsiESTs}' = \mathit{notpsiESTs} \cup \{\mathit{est}\}$ arising from the replacement of a subformula $\Box_k \psi$ where $\mathit{est} = \sigma[p](\mathit{start}) + k$. We now show that $\sigma, p \models \Box_k\, \psi$. Let $r > p+1$ be any index such that $\sigma[r](\mathit{start}) \leq \mathit{est}$. If $\mathit{est} \notin \sigma[r](\mathit{notpsiESTs})$, then est was removed from $\mathit{notpsiESTs}$ at some point $q \in [p+2, r-1]$. But since $\mathit{notpsiESTs}$ is protected, only the action 4.1.2 can change $\mathit{notpsiESTs}$, specifically setting it to empty. Again by 4.1.2 this means that $\neg\psi$ held at $q-1$, so we have both $\sigma, q-1 \models \neg\psi$ and $\sigma, q-1 \models \psi$ and there can be no such state. So, we have $\mathit{est} \in \sigma[r](\mathit{notpsiESTs})$ and $\sigma[r](\mathit{start}) \leq \mathit{est} \leq \mathit{max}(\mathit{notpsiESTs})$. By (4.1.3) $\sigma, r \models \psi$, hence $\sigma, p \models \Box_k\, \psi$. This shows that whenever the action $\mathit{notpsiESTs}' = \mathit{notpsiESTs} \cup \{\mathit{start} + k\}$ takes place in an arbitrary trace, then we also have the property $\Box_k \psi$; i.e.

$$\mathit{notpsiESTs}' {=} \mathit{notpsiESTs} \cup \{\mathit{start} + k\} \implies \Box_k \psi.$$

Since we have only replaced positive occurrences of subformulas of the form $\Box_k \psi$, Φ is monotone in those locations, and so we infer $\Phi' \implies \Phi$.

5.2 Example: Secure Enclave

Transformations 1 and 2 normalize a specification by refining it to another specification that only requires basic safety properties. The following example illustrates a two-step process of property refinement transformation followed by model refinement.

A secure enclave has a door whose latch is controlled by a card reader. A user can Insert or Remove a token from the reader. The system controls the latch and can perform Lock or Unlock actions. When unlocked, the Door can be opened. Suppose that we have the following specification for a secure enclave.

```
Specification SecureEnclave0
    Node: $m_0$
        vars: $\mathit{token}, \mathit{lock} : \mathit{Boolean}$
              $k : \mathit{Time}$
        invariant: $\mathit{true}$
    Arc: $a = \langle m_0, m_0 \rangle$
        actions: $\mathit{Insert} \triangleq \neg\mathit{token} \wedge \mathit{token}'$
                 $\mathit{Remove} \triangleq \mathit{token} \wedge \neg\mathit{token}'$
                 $\mathit{Lock} \triangleq \mathit{lock}'$
                 $\mathit{Unlock} \triangleq \neg\mathit{lock}'$
    Required Properties
        $\Box \mathit{Insert} \implies \Diamond_k \mathit{Unlock}$
        $\Box \mathit{Unlock} \implies \diamondminus_k \mathit{Insert}$
End Specification
```

where the required properties specify that (1) whenever an *Insert* action occurs then there will be *Unlock* action no more than k time units in the future, and (2) whenever an *Unlock* action occurs then there was an *Insert* event no longer than k time units in the past.

Applying the property refinement transformations from the previous section, we generate a specification refinement $SecureEnclave0 \sqsubseteq SecureEnclave1$ where $SecureEnclave1$ has no occurrence of time-bounded temporal operators in its required properties.

Specification SecureEnclave1
  Node: m_0
    vars: $token, lock : Boolean$
      $k : Time$
    protected vars: $unlockDeadlines : Set(Time)$
      $lastInsert : Time$
    invariant: $true$
  Arc: $a = \langle m_0, m_0 \rangle$
    actions: $Insert \triangleq \neg token \wedge token'$
      $Remove \triangleq token \wedge \neg token'$
      $Lock \triangleq lock'$
      $Unlock \triangleq \neg lock'$
  Required Properties
    $lastInsert = -\infty$
    $unlockDeadlines = \{\}$
    $\Box Insert \implies lastInsert' = start$
    $\Box Unlock \implies start - k \leq lastInsert$
    $\Box Unlock \implies unlockDeadlines' = \{\}$
    $\Box start \leq min(unlockDeadlines)$
    $\Box Insert \implies unlockDeadlines' = unlockDeadlines \cup \{start + k\}$
  Theorems
    $\Box Insert \implies \Diamond_k Unlock$
    $\Box Unlock \implies \diamondminus_k Insert$
End Specification

The initial required properties are theorems in this refined model (as consequences of Theorems 2 and 3). Applying the model refinement procedure from Section 3, we generate a refined model that satisfies the initial goals by-construction and has no unrealized required properties.

Specification SecureEnclave2
  Node: m_0
    vars:
      $token : Boolean = false$
      $lock : Boolean = true$
      $k : Time$
    protected vars: $unlockDeadlines : Set(Time) = \{\}$
      $lastInsert : Time = -\infty$
    invariant: $start \leq min(unlockDeadlines)$

Arc: $a = \langle m_0, m_0 \rangle$
actions:
$Insert \triangleq \neg token \wedge token' \wedge lastInsert' = start$
$\wedge\ unlockDeadlines' = unlockDeadlines \cup \{start + k\}$
$Remove \triangleq token \wedge \neg token' \wedge unchanged(lastInsert) \wedge unchanged(unlockDeadlines)$
$Lock \triangleq lock' \wedge unchanged(lastInsert) \wedge unchanged(unlockDeadlines)$
$Unlock \triangleq start - k \leq lastInsert \wedge \neg lock' \wedge unlockDeadlines' = \{\}$
$\wedge\ unchanged(lastInsert)$
Required Properties
Theorems
$\Box Insert \implies \Diamond_k Unlock$
$\Box Unlock \implies \diamondminus_k Insert$
End Specification

The refined state invariant implies that when *unlockDeadlines* is nonempty, then the system must execute the Unlock transition before the earliest deadline $min(unlockDeadlines)$. The initial required properties hold by-construction in this refined model and can be verified by a model-checking algorithm.

5.3 Other Examples

In [5], the Cinderella game is introduced and solved using sketches as hints to the solver. The game is parametric on a real value c used to define the Stepmother's (antagonist's) task. It is conjectured that automatic solutions (i.e. without human-provided hints) are "unrealistic" for values of c in a certain range. Our model refinement prototype automatically generates winning strategies in that range using roughly a minute of CPU time.

6 Refinement-Generating Transformations

The previous sections present some refinement-generating transformations for the purpose of reducing syntactic sugar to basic safety properties. Here, we briefly note a few other transformations that are commonly used to improvement the performance of a system or algorithm design.

The *finite differencing* transformation (aka *incrementalization*) [31, 28, 39], can be expressed as a transformation that introduces a safety property to enforce by model refinement. The key idea is to introduce a protected fresh variable, say c, and an invariant safety property $\Box c = e(st)$, where e is some function of state st. The references develop this transformation on both imperative and functional models.

Example: Transformations 1 and 2 above provide implicit examples of finite differencing. Transformation 1 introduces the fresh protected variable *lastpsi* together with the safety property $\Box\Psi \implies lastpsi' = start$ (or equivalently $\Box lastpsi' = if \Psi then start else lastpsi$). The model refinement process then effectively refines the actions of the model to incrementally compute *lastpsi*.

7 Path Properties

Some required properties are naturally expressed over the endpoints of a path in the model, rather than state and action invariants. They express required properties that hold between values that are not near in time or space (as between the prestate and poststate of an action). We express *path properties* as predicates over the variables of states and constants, as with state properties, except that when necessary we prefix the variable with the node at which the value is referenced. If a variable is only accessible at one node (i.e. it is local), the prefix can be omitted.

We define next some normalization rules that can be used to reduce path properties to action properties. The normalization rules work by propagating the path property through the structure of the path, resulting in the strengthening of the labels on particular arcs. The resulting refined path implies the path property by-construction. At that point, the constraint system of Section 3 can be defined and solved.

Path properties may arise by the imposition of model substructure, where an arc is replaced by an arc-like LCFG (i.e. a submodel). This may happen when an action specifies a complex state change that requires, say, an iterative or recursive computation to complete. We call this process *structure refinement*. Suppose that we have a required property $\varphi_{m,p}(st_m, st_p)$ that relates the state at node m to the state at node p, where there exists a path from m to p in the model $\mathcal{M}$. Our strategy is to propagate φ through the structure of $\mathcal{M}$ until we have inferred properties that can be localized to the nodes and arcs of $\mathcal{M}$. For purposes of reasoning about path properties we proceed as if we have path labels in $\mathcal{M}$ for all pairs of nodes; e.g. $L_{m,p}$ is treated as the label expressing properties of the paths from node m to node p.

There are two propagation rules that reduce the scope of a path property, with the goal of reducing the property to localized refinements on actions in the path: either propagate forward from node m toward p, or propagate backward from node p toward m. Rules for both are defined next. Each rule reduces the span of a path predicate by one, so we iterate their application until we generate a path predicate than spans a single arc, whereupon we can enforce it locally.

Forward Propagation: Let $\mathcal{S} = \langle \mathcal{M}, \Phi \rangle$ be a specification and let $\varphi_{m,p} \in \Phi$ be a path property from node m to the state at node p. We can refine $\mathcal{S}$ to reduce a path property as follows: (1) Delete $\varphi_{m,p}$ from Φ, and (2) for each arc $a = \langle m, n \rangle \in Arc$, add the path formula $wcPostSpec(L_a, \varphi_{m,p})$ to Φ where $wcPostSpec(L_a, \theta)$ is the *Weakest Controllable PostSpecification* of action L_a with respect to path formula θ over $V(m) \cup V(p)$ and is defined by

$$wcPostSpec(L_a, \theta) \;\equiv\; \forall st_m, u, e, st_n . L_m(st_m) \wedge U(st_m, u) \wedge e \in E(st_m) \wedge st_n = f_a(st_m, u, e) \implies \theta.$$

$wcPostSpec$ is the weakest path formula over $V(n) \cup V(p)$ such that for any transition instance of a from some state st_m to state st_n, there is some st_p such that $\theta(st_m, st_p)$. We repeat Forward Propagation until all path properties have been reduced to actions (and thus can be enforced by model refinement).

Backward Propagation: Let $\mathcal{S} = \langle \mathcal{M}, \Phi \rangle$ be a system specification and let $\varphi_{m,p} \in \Phi$ be a path formula from node m to the state at node p. We can refine $\mathcal{S}$ to reduce the path property occurrences as follows: (1) Delete $\varphi_{m,p}$ from Φ, and (2) for each arc $a = \langle n, p \rangle \in Arc$ where there exists a path from m to n, add the path formula $wcPreSpec(L_a, \varphi_{m,p})$ to Φ where $wcPreSpec(L_a, \theta)$ is the

Weakest Controllable PreSpecification of action L_a with respect to path formula θ over $V(m) \cup V(p)$ and is defined by

$$wcPreSpec(L_a, \theta) \;\equiv\; \forall u, e, st_p.\, L_n(st_n) \land U(st_n, u) \land e \in E(st_n) \land st_p = f_a(st_n, u, e) \implies \theta.$$

wcPreSpec is the weakest path formula over $V(m) \cup V(n)$ such that for any transition instance of a from some state st_n to state st_p, there is some st_m such that $\theta(st_m, st_p)$. We repeat Backward Propagation until all path properties have been reduced to actions (and thus can be enforced by model refinement).

Both of these propagation rules work by propagating the path property φ through the transition a, whether forward or backwards. To get useful results, there must be some structure in L_a. These rules are often applied after one has chosen a candidate function/operation for transition a and then desires to play out the consequences. This process is analogous to SAT algorithms in which one chooses a variable and a value heuristically and then explores the consequences via boolean propagation and conflict-driven learning in the failure case. The choice of a simple operation that is natural in context, as a structure refinement, enables the propagation to go through. This is a choice and alternative choices lead to different designs, as illustrated in the next section.

7.1 Algorithm Design Example: Sorting

One feature of model refinement is that it subsumes a major part of the automated algorithm design work performed in earlier function synthesis systems such as KIDS [39]. In retrospect, the success of KIDS in algorithm design is partly due to its automated inference system which was designed to propagate output conditions through the structure of a chosen program scheme. To illustrate, consider the design of a sorting algorithm using a binary divide-and-conquer program scheme as a model. In a functional notation, the model can be expressed as

$$F(x : D) : (z : R) = \;\textsf{if}\; primitive(x) \;\textsf{then}\; direct(x) \;\textsf{else}\; compose \circ (F \times F) \circ decompose(x)$$

and the required property is $bag(x) = bag(z) \land ordered(z)$, where x and z are lists of numbers, $bag(x)$ returns the bag or multiset of elements in list x, and $ordered(z)$ holds when list z is in sorted order. The property is simply an input/output predicate since the only observable behavior of an algorithm is its (uncontrollable) input and (controllable) output value. In a functional setting, there are no global variables and hence no global state. The input to each functional component is the environment input and the control value is the output of the action.

There are several common tactics for designing divide-and-conquer algorithms. One is to select a simple *decompose* operation on the input type, and then to calculate a *compose* operator that achieves the correct output. A dual tactic is to select a simple *compose* operation on the output type, and then calculate a *decompose* operator that achieves a decomposition of the input into parts that can be solved and composed to yield a correct solution.

We might represent the key recursive part of the scheme as a dataflow path:

$$\langle x_0 \rangle \xrightarrow{decompose(x_0, x_1, x_2)} \langle x_1, x_2 \rangle \xrightarrow{F(x_1, z_1) \times F(x_2, z_2)} \langle z_1, z_2 \rangle \xrightarrow{compose(z_0, z_1, z_2)} \langle z_0 \rangle$$

where a node represents a state by the variables that exist in it (and their properties), and each arc specifies an action by a predicate over input and output variables. This particular model derives from a functional program, so the abstract "states" actually do not represent stored values, but the value flow at intermediate points in a computation. For simplicity and clarity, we use this graphical representation rather than perform the straightforward translation to the TLA-like notation used in previous examples.

In terms of the dataflow path, the goal constraint is a predicate over x_0 and z_0: $\varphi(x_0, z_0) \equiv bag(x_0) = bag(z_0) \wedge ordered(z_0)$. Suppose that we follow the second tactic and refine the model by choosing list concatenation as our *compose* operator: $compose \mapsto z_0 = z_1 +\!\!+ z_2$. The ultimate effect of this choice is to derive a variant of a quicksort algorithm. Note that in this case the environment input is the pair $\langle z_1, z_2\rangle$ and the control value is the output z_0. The Backward Propagation Rule applies here since the goal property is not expressed over the input and output variables of *compose*, so we calculate:

$$
\begin{array}{ll}
\multicolumn{2}{l}{wcPreSpec(compose, \varphi(x_0, z_0))} \\
\equiv & \forall z_0.\, z_0 = z_1 +\!\!+ z_2 \implies bag(x_0) = bag(z_0) \wedge ordered(z_0) \\
\equiv & bag(x_0) = bag(z_1 +\!\!+ z_2) \wedge ordered(z_1 +\!\!+ z_2) \qquad \text{Quantifier Elimination on } z_0 \\
\equiv & bag(x_0) = bag(z_1) \cup bag(z_2) \wedge ordered(z_1) \wedge ordered(z_2) \wedge bag(z_1) \leq bag(z_2). \quad \text{Simplification}
\end{array}
$$

where we have used domain-specific laws for distributing *bag* and *ordered* over list concatenation, and $b_1 \leq b_2$ holds when each element of bag b_1 is less than or equal to each element of bag b_2. As this remains a path predicate $\varphi(x_0, z_1, z_2)$ (i.e. not localizable to an arc), we continue by propagating this derived goal backward through the recursive calls:

$$
\begin{array}{ll}
\multicolumn{2}{l}{wcPreSpec(F \times F, \varphi(x_0, z_1, z_2))} \\
\equiv & \forall z_1, z_2.\, bag(x_1) = bag(z_1) \wedge ordered(z_1) \wedge bag(x_2) = bag(z_2) \wedge ordered(z_2) \\
 & \implies bag(x_0) = bag(z_1) \cup bag(z_2) \wedge ordered(z_1) \wedge ordered(z_1) \wedge bag(z_1) \leq bag(z_2) \\
\equiv & bag(x_0) = bag(x_1) \cup bag(x_2) \wedge bag(x_1) \leq bag(x_2). \qquad \text{Simplification and Quantifier Elimination}
\end{array}
$$

This last predicate is expressed over the input/output variables of the *decompose* operator, so it can be localized and enforced by strengthening the *decompose* action to

$$bag(x_0) = bag(x_1) \cup bag(x_2) \wedge bag(x_1) \leq bag(x_2).$$

Note that this is a specification for (a version of) the well-known partition subalgorithm of Quicksort. It asserts that if we decompose the input list x_0 into two lists x_1 and x_2 whose collective elements are the same as the elements in x_0, and such that each element of x_1 is less-than-or-equal-to each element of x_2, then when we recursively sort x_1 and x_2, and then concatenate them, the result will be a sorted version of x_0. If we had included a well-founded order in the decompose operator, we would infer a derived initial condition of $length(x_0) > 1$ on *decompose*. This serves as a guard on the recursive path in the algorithm.

In summary, we have used propagation rules to infer a specification on the *decompose* action that, if realized by further refinement, is sufficient to establish the correctness of the whole algorithm. The complete derivation of Quicksort, including the use of divide-and-conquer to synthesize the partition operation, may be found in [38], which also derives several other sorting algorithms. Derivation of several parallel sorting algorithms via divide-and-conquer may be found in [41].

8 Refining Concurrent Models

LGCFs naturally model sequential systems that interact with an external environment. In [30] we explore model refinement on statechart models, which admit concurrent subprocesses. Several interesting results come from this work. Assume for simplicity that a system S has subprocesses A and B. Process A does not know what actions B will take, and so to provide guarantees of enforcing the global safety property $\Box\Phi$, process A must treat the actions of B as environment actions. That is, the constraints that model actions of A must universally quantify over possible actions of B, and conversely. It is easy to construct examples in which this local lack of knowledge on the part of A and B leads to the model refinement fixpoint iteration eliminating some feasible joint behaviors. Generally, given a concurrent system model, the greatest fixpoint of the corresponding constraint system represents only a subset of feasible joint behaviors. The main way around this problem is effectively to add a scheduler to the system model. The corresponding constraint system then has a greatest fixpoint that represents all feasible joint behaviors.

9 Related Work

Our previous work on functional algorithm design used algorithm theories as over-approximating models for various classes of algorithms. Algorithm theories and design tactics [44] were implemented in KIDS [39] and Specware [25]. These synthesis systems used a form of model refinement to instantiate algorithm models for divide-and-conquer [38], global search, dynamic programming [40], and other classes. Synthesized applications include schedulers [8, 45], SAT-solvers [47], and garbage collectors [46].

Sketching [48] is a currently popular program synthesis approach that can be seen as a special case of model refinement. The model is supplied in the form of a program template with holes for missing code. In the case of SyGuS [2], a grammar is given as an over-approximation to the missing code. The property to be enforced may be expressed using the language of an SMT-solver, so that guesses as to how to fill the hole can be verified. While the problem setup is similar to model refinement, the synthesis process is based on generate-and-test rather than predicate transformer-based calculation.

Model refinement is most obviously derived from the extensive literature on controller synthesis [34, 18] and reactive system design [33]. Most current work on the synthesis of reactive systems focuses on circuit design and starts with specifications in propositional Linear Temporal Logic (LTL) or GR(1) [6, 24]. Model Refinement allows specifications that are first-order and uses a temporal logic of action that is amenable to refinement, which LTL is not, allowing a broader range of applications to be tackled.

The algorithm derivation in Section 7 highlights a novel aspect of model refinement: the imposition of a design template rather than a plant or game model as it typical in reactive system design. Design templates in the systems world are often discussed as Design Patterns. The refinement mechanism is structure refinement (see Section 2.3) which refines a model arc by an arc-like LCFG, in effect, replacing the arc with a design pattern. The arc specification becomes a path property and the normalization rules in Section 7 are used to localize the property by strengthening arc labels along the path. It is typical of algorithm derivation that structure refinements are needed to implement arc labels resulting in the top-down synthesis of subalgorithms. Algorithms often have a deeper hierarchy of subcomputations than system control codes.

The model refinement constraints are a kind of Constrained Horn Clause (CHC) and specialized algorithms have been explored for these as a generalization of SMT solving [17, 20, 21]. The main application is finding inductive invariants for program verification. Our approach aims to find a maximal solution whereas CHC tools typically aim to find any solution, since any inductive invariant is sufficient to establish the specified verification condition.

10 Discussion

The concept of model refinement only requires a semilattice of models and a language for expressing required properties. For concreteness, we have presented a Boolean lattice of models defined by labeled control flow graphs with first-order constraints and required properties in the form of basic safety properties. While this provides a fairly general and mechanizable framework for user-guided, yet highly automated design, it also admits the possibility of high computational complexity or undecidability due to the expressiveness of the first-order formulas. By suitably restricting the domain of discourse to decidable theories, we can define a more tractable and automatic design process. Our prototype implementation restricts constraints to the decidable theories in Z3, which is sufficient for a range of applications including the examples presented above. Extension to handle liveness properties ($\Diamond\varphi$) and reactivity properties ($\Box\Diamond\varphi$) can also be handled as definite constraint systems whose fixpoints can be found by Kleene iteration combined with widening. However, for practical purposes, reactive systems typically want guarantees of bounded-time responsiveness, which is a safety property as seen in Section 5.

Model Refinement is intended to be part of a library of refinement-generating transformations that are used to develop complex algorithms and systems. In our view, a practical synthesis environment generates a refinement chain from an initial specification down to compilable code. Each step of the refinement chain is generated by a transformation that is also capable of emitting proofs of the refinement relation between the pre- and post-specification [42]. Model Refinement would tend to be used earlier in the refinement chain since it translates logical requirements into operational designs, by enforcing properties in the model. Other refinement-generating transformations are necessary to improve the performance of the evolving model including expression simplification, finite-differencing or incrementalization, and datatype refinements [28, 39].

Treating a specification as a model plus required properties is a key aspect of model refinement. Models are essentially programs annotated with invariant properties. While temporal logics can be translated into automata (and vice-versa), for complex designs, the models can be much more compact than logic, especially when the nodes have rich properties and the control structure is complex. Initially, models serve to succinctly capture fixed behavioral structure in the problem domain, such as physical plant dynamics and information system APIs. During refinement, the model serves as the accumulation of the design decisions made so far. Another intended use of models is via the imposition of design patterns for algorithms and systems. Patterns from a library capture best-practice designs that might be difficult to find by search; e.g. when there is a delicate tradeoff between “ilities”, such as between precision of output and runtime.

References

1. Coq website. https://coq.inria.fr/.
2. Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2013), pp. 1–17.
3. Alur, R., and La Torre, S. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Logic 5*, 1 (Jan. 2004), 125.
4. Asarin, E., Maler, O., Pnueli, A., and Sifakis, J. Controller synthesis for timed automata. *IFAC Proceedings Volumes 31*, 18 (1998), 447–452. 5th IFAC Conference on System Structure and Control 1998 (SSC'98).
5. Beyene, T., Chaudhuri, S., Popeea, C., and Rybalchenko, A. A constraint-based approach to solving games on infinite graphs. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014), p. 221233.
6. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., and Saar, Y. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences 78*, 3 (2012), 911–938.
7. Buchi, J. R., and Landweber, L. H. Solving sequential conditions by finite-state stategies. *Transactions of the AMS 138* (1969), 295–311.
8. Burstein, M. H., and Smith, D. R. ITAS: A portable interactive transportation scheduling tool using a search engine generated from formal specifications. In *Proceedings of AIPS-96* (Edinburgh, UK, May 1996).
9. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, 1996.
10. Church, A. Application of recursive arithmetic to the problem of circuit synthesis. In *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University* (1957).
11. Church, A. Application of recursive arithmetic to the problem of circuit synthesis. *Journal of Symbolic Logic 28*, 4 (1963), 289–290.
12. Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking.* MIT Press, 2000.
13. Constable, R. L. Constructive mathematics and automatic program writers. In *Information Processing 71* (Ljubljana, Yugoslavia, August 23–28, 1971), IFIP, pp. 229–233.
14. Constable, R. L. *Implementing Mathematics with the NuPrl Proof Development System.* Prentice-Hall, New York, 1986.
15. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977), ACM, pp. 238–252.
16. Dutertre, B. Yices 2.2. In *Computer-Aided Verification (CAV'2014)* (July 2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744.
17. Fedyukovich, G., Prabhu, S., Madhukar, K., and Gupta, A. Solving constrained horn clauses using syntax and data. In *2018 Formal Methods in Computer Aided Design (FMCAD)* (2018), pp. 1–9.
18. Filippidis, I., Dathathri, S., Livingston, S. C., Ozay, N., and Murray, R. M. Control design for hybrid systems with tulip: The temporal logic planning toolbox. In *2016 IEEE Conference on Control Applications (CCA)* (2016), pp. 1030–1041.
19. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
20. Govind, V., Shoham, S., and Gurfinkel, A. Solving constrained horn clauses modulo algebraic data types and recursive functions. *Proc. ACM Program. Lang. 6*, POPL (2022).
21. Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. Synthesizing software verifiers from proof rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012), p. 405416.
22. Green, C. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence* (1969), pp. 219–239.
23. Gulwani, S., and Musuvathi, M. Cover algorithms and their combination. In *Programming Languages and Systems* (2008), Springer, pp. 193–207.
24. Jacobs, S., Klein, F., and Schirmer, S. A high-level ltl synthesis format: Tlsf v1.1. *Electronic Proceedings in Theoretical Computer Science 229* (2016), 112132.
25. Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.
26. Kleene, S. On the interpretation of intuitionistic number theory. *J. Symbolic Logic 10*, 4 (1945), 109–124.
27. Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (1994), 872–923.

28. Liu, Y. *Systematic Program Design: From Clarity to Efficiency.* Cambridge University Press, 2013.
29. Manna, Z., and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 90–121.
30. Nedunuri, S., and Smith, D. R. Automated modular refinement of statecharts and components. Tech. Rep. SAND Technical Report X, Sandia National Laboratory, 2023.
31. Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.
32. Paul, G. Approaches to abductive reasoning - an overview. *Artificial Intelligence Review 7*, 2 (1993), 109–152.
33. Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1989), p. 179190.
34. Ramadge, P., and Wonham, W. The control of discrete event systems. *Proceedings of the IEEE 77*, 1 (1989), 81–98.
35. Rehof, J., and Mogensen, T. Tractable constraints in finite semilattices. *Science of Computer Programming 35* (1999), 191–221.
36. Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* Wiley, 2000.
37. Slanina, M., Sankaranarayanan, S., Sipma, H., and Manna, Z. Controller synthesis of discrete linear plants using polyhedra. Tech. rep., Technical Report REACT-TR-2007-01, Stanford University, 2007.
38. Smith, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).
39. Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.
40. Smith, D. R. Structure and design of problem reduction generators. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 91–124.
41. Smith, D. R. Derivation of parallel sorting algorithms. In *Parallel Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter, Eds. Kluwer Academic Publishers, New York, 1993, pp. 55–69.
42. Smith, D. R. Generating programs plus proofs by refinement. In *Verified Software: Theories, Tools, Experiments* (2008), B. Meyer and J. Woodcock, Eds., Springer-Verlag LNCS 4171, pp. 182–188.
43. Smith, D. R. Model refinement code. https://github.com/KestrelInstitute/modelRefinement, 2021.
44. Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. In *Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375*, L. van de Snepscheut, Ed. Springer-Verlag, Berlin, 1989, pp. 379–398. (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321).
45. Smith, D. R., Parra, E. A., and Westfold, S. J. Synthesis of planning and scheduling software. In *Advanced Planning Technology* (1996), A. Tate, Ed., AAAI Press, Menlo Park, pp. 226–234.
46. Smith, D. R., Westbrook, E., and Westfold, S. J. Deriving Concurrent Garbage Collectors: Final Report. Tech. rep., Kestrel Institute, 2015. http://www.kestrel.edu/home/people/smith/pub/Crash-FR.pdf.
47. Smith, D. R., and Westfold, S. Toward the Synthesis of Constraint Solvers. Tech. rep., Kestrel Institute, 2013. http://www.kestrel.edu/home/people/smith/pub/CW-report.pdf.
48. Solar-Lezama, A. The sketching approach to program synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS* (2009), vol. 5904 of *Lecture Notes in Computer Science*, Springer, pp. 4–13.
49. Sontag, E. *Mathematical Control Theory.* Springer, 1998.

Appendix: Secure Enclave Example in TLA+

The specification SecureEnclave1 in Section 5.2 can be translated into TLA+ as follows. It runs successfully through the TLA model-checker.

```
----------------------------- MODULE SecureEnclave1 -----------------------------

EXTENDS Naturals, Integers, Sequences, FiniteSets, TLC

CONSTANTS
MAX_TIME, K

(*TYPES*)
Time == -MAX_TIME .. (MAX_TIME+1)

VARIABLES
clock,              (* what is the current time? *)
locked,             (* is the door locked? *)
token,              (* is there a token in the reader? *)
lastInsertT,        (* last time that a card was inserted in reader *)
unlockDeadlines,    (* set of deadlines for when to unlock *)
lastUnlockT         (* need to express the Eventually property below *)

vars == <<clock, locked, token, lastInsertT, unlockDeadlines, lastUnlockT>>
varsXclock == << locked, token, lastInsertT, unlockDeadlines, lastUnlockT>>

TypeCheck ==
     /\ locked \in BOOLEAN
     /\ token  \in BOOLEAN
     /\ clock \in Nat
     /\ lastInsertT \in Time
     /\ lastUnlockT \in Time
     /\ \A x \in unlockDeadlines: x \in Nat  \*everything in TLA is a set

\* MAX_TIME+1 so CheckTimeConstraint' below doesn't block when clock reaches MAX_TIME
min(xs) == IF xs={} THEN MAX_TIME+1 ELSE CHOOSE x \in xs : (\A y \in xs: x <= y)

(* --- SYSTEM ACTIONS --- *)
(* 1. Unlock the Latch  *)
Unlock ==
        clock <= lastInsertT + K
    /\  locked'=FALSE /\ unlockDeadlines' = {} /\ lastUnlockT' = clock /\
            UNCHANGED <<token, lastInsertT>>

(* 2. Lock the Latch *)
Lock ==
    locked'=TRUE /\ UNCHANGED <<token, lastInsertT, unlockDeadlines, lastUnlockT>>

(* --- ENV ACTIONS --- *)
(* 3. Insert the Token *)
Insert ==
        ~token
    /\  token'=TRUE /\
        lastInsertT'=clock /\ unlockDeadlines' = unlockDeadlines \union {clock+K} /\
        UNCHANGED <<locked, lastUnlockT>>

(* 4. Remove the Token *)
Remove ==
        token
    /\  token'=FALSE /\ UNCHANGED <<locked, lastInsertT, unlockDeadlines, lastUnlockT>>

(* --------- the specification of behavior  ---------------- *)
Init ==
    /\ locked = TRUE
    /\ token = FALSE
    /\ clock = 0
    /\ lastInsertT = -MAX_TIME (* end of time *)
    /\ lastUnlockT = MAX_TIME
    /\ unlockDeadlines = {}
    /\ TypeCheck

EnvActs ==
    \/  Insert
    \/  Remove

SysActs ==
    \/  Unlock
    \/  Lock

DoNothing == UNCHANGED varsXclock

(* state invariant *)
CheckTimeConstraint == clock <= min(unlockDeadlines)

(* CheckTimeConstraint' performs a "lookahead" to ensure we don't transition to a
   state that doesn't satisfy the invariant *)
SysOrEnvOrSkip == (EnvActs \/ SysActs \/ DoNothing) /\ CheckTimeConstraint'

ClockTick == clock' = clock+1

Next == ClockTick /\ SysOrEnvOrSkip /\ CheckTimeConstraint
```

```
Spec == Init /\ [][Next]_vars

(* any Unlock action must be preceded by a card Insert within k time units
      UnlockPre == Unlock => (<_>_K Insert)
   after prop transform this become  *)
InsertBeforeUnlock == [][Unlock => (clock <= lastInsertT + K)]_vars

UnlockHappened == lastUnlockT \in lastInsertT .. clock
InsertLeadsToUnlock ==
    [][(lastInsertT # -MAX_TIME /\ lastInsertT < clock /\ ~UnlockHappened) =>
        clock <= min(unlockDeadlines)]_vars

THEOREM Spec => []TypeCheck
THEOREM Spec => InsertBeforeUnlock
THEOREM Spec => InsertLeadsToUnlock
=============================================================================
```