Model Refinement (Extended)*

Douglas R. Smith1 and Srinivas Nedunuri2

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

19 November 2021

Abstract. Model Refinement is a uniform approach to generating correct-by-construction designs for algorithms and systems from formal specifications. Given an overapproximating model $M$ of system dynamics and a set $\Phi$ of required properties, model refinement is an iterative process that eliminates behaviors of $M$ that do not satisfy the required properties. The result of model refinement is a refined model $M'$ that satisfies by-construction the required properties $\Phi$. The calculations needed to generate refinements of $M$ typically involve quantifier elimination and extensive formula/term simplification modulo the underlying domain theories. 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

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

and computable functions; prominently, Kleene’s 1945 work on realizability [18]. 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) [15, 11, 21]. Current support for program extraction from proofs is typically provided in proof environments for constructive logics such as NuPRL [12] and Coq [1].

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 [8, 9] as in digital circuits. This corresponds to generalizing formula (1) to an infinite alternation of quantifiers:

$$\forall x_0 \exists x_1 \forall x_2 \exists x_3 \cdots \varphi(x_0, x_1, x_2, x_3, \cdots)$$

* 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.
Formula (2) is naturally interpreted as a game between the program and its environment with winning condition \( \varphi \). 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 \( \varphi \) holds, in which case the program wins, otherwise the environment wins. Solutions to Church’s problem [6, 23] 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, 5].

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 \( \Phi \) 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 \( \Phi \). 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 [10] 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 [14, 7, 25]. In algorithm design, a model can reflect the imposition of a parametric solution pattern, such as an algorithm theory [32] or a sketch [34].

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) [5, 17]. 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 [31].

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 4.1) and path
properties (Section 4.3) 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 [19]. 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 \( \varphi \) or \( \Box \varphi \) where \( \varphi \) is a state
predicate or an action. The truth of a safety property \( \varphi \) at position \( n \) of a trace \( \sigma \) (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) \( M = (V, N, A, L) \) where

- \( V \): 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 \( V(m) \subseteq V \). \( N \) 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.
- \( L \): a set of labels. For each node \( m \in N \), we have a label \( L_m \in L \) that is a state predicate over
\( V(m) \) representing a node invariant. For each arc \( a = \langle m, n \rangle \), label \( L_a \in L \) 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 refnable and which are unrefnable. Accordingly, the label $L_a$ of an action has the general form:

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

where

1. $e$ is treated as an uncontrollable environment or adversary input that ranges over the unrefnable set $E_a(st_m)$;
2. $u$ is treated as a controllable value that satisfies the refnable 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 refnable 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 [35]. Let

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

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 $\mathcal{M} = (\mathcal{N}, \mathcal{A}, L)$ 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 $\xi : \mathcal{M}_2 \to \mathcal{M}_1$ that maps the nodes and arcs of $\mathcal{M}_2$ to the nodes and arcs of $\mathcal{M}_1$; i.e. where $\xi : N^{\mathcal{M}_2} \to N^{\mathcal{M}_1}$ and $\xi : A^{\mathcal{M}_2} \to A^{\mathcal{M}_1}$, such that
1. Initial nodes are preserved: $\xi(m_0^{M_2}) = m_0^{M_1}$;
2. Observable variables: $V^{M_2}(m) \supseteq V^{M_1}(\xi(m))$ for each node $m \in N^{M_2}$;
3. Node labels: $L_m^{M_2} \implies L^{M_1}_\xi(m)$ for each node $m \in N^{M_2}$;
4. Arc labels: $L_a^{M_2} \implies L^{M_1}_{\xi(a)}$ for each arc $a \in A^{M_2}$.

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 4.4.

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

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

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

$$\forall \sigma, \sigma' \in [T] \implies \xi(\sigma) \in [S]$$

or more succinctly $\xi([T]) \subseteq [S]$.

**Theorem 1.** If
1. $S_1 = \langle M_1, \Phi_1 \rangle$ and $S_2 = \langle M_2, \Phi_2 \rangle$ are specifications,
2. $\xi : M_2 \to M_1$ is a simulation map
3. $\Phi_2 \implies \Phi_1$ then $S_1 \sqsubseteq S_2$.

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

![Fig. 1: Simulating a trace](image)

Given a trace $\sigma'$ of $H$, we show how to construct a trace $\sigma$ of $G$. If $h_0$ is the start node of $H$, then $\sigma'[0] \in [h_0]$. By construction, $\xi(h_0) = m_0$ where $m_0$ is the start node of $G$, 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 $V_H(h_0) \supseteq V_G(\xi(h_0)) = V_G(m_0)$; i.e.

$$\sigma[0] = \{ v \mapsto \text{val} \mid v \in V(m_0) \land \text{val} = \sigma'[0](v) \}.$$
\[ \sigma'[0] \models L_{h_0} \]
\[ \implies \quad \sigma'[0] \models L_{m_0} \]
\[ \implies \quad \sigma[0] \models L_{m_0} \]
\[ \text{since } L_{h_0} \Rightarrow L_{\xi(h_0)} \iff L_{\xi(m_0)} \]
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 \( H \) 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

\[ \langle \sigma'[i], \sigma'[i+1] \rangle \models L_b \]
\[ \implies \quad \langle \sigma'[i], \sigma'[i+1] \rangle \models L_a \]
\[ \equiv \quad \langle \sigma[i], \sigma[i+1] \rangle \models L_a \]
\[ \text{since } L_b \Rightarrow L_{\xi(b)} \iff L_a \]
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 \( \xi([M_2]) \subseteq [M_1] \). To show specification refinement:

\[ \xi([S_2]) \]
\[ = \quad \xi([M_2] \cap [\Phi_2]) \quad \text{definition} \]
\[ = \quad \xi([M_2]) \cap \xi([\Phi_2]) \]
\[ \subseteq \quad [M_1] \cap [\Phi_2] \quad \text{distributing} \]
\[ \subseteq \quad [M_1] \cap [\Phi_1] \quad \text{since } \xi([M_2]) \subseteq [M_1] \]
\[ = \quad [S_1]. \quad \text{assumption that } \Phi_2 \Rightarrow \Phi_1 \]

That is, \( S_1 \subseteq S_2 \). QED

3 Model Refinement as Constraint Solving

Model refinement transforms a model \( M \) and required properties \( \Phi \) into a model \( M' \) such that \( M \subseteq M' \wedge M' \models \Phi \). We define now a constraint system whose solutions correspond to refinements of \( M \) that satisfy \( \Phi \). The intent is to find the greatest solution of the constraint system, which corresponds to the minimal refinement of \( M \) that satisfies \( \Phi \). 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.

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 \textit{weakest controllable predecessor} (WCP) predicate transformer and is defined by

\[
\text{wcp}(L_a, L_n) \equiv \forall e. \ e \in E(st_m) \implies \exists st_n. \ st_n = f_a(st_m, e, u) \land L_n(st_n)
\]

or, simply

\[
\text{wcp}(L_a, L_n) \equiv \forall e. \ e \in E(st_m) \implies L_n(f_a(st_m, e, u))
\]

where \( L_n \) is a state predicate. \text{wcp} is the weakest formula over \( V(m) \cup \{ u \} \) 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-satisfaction problem posed by the conjunction \( \Theta \) 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 \( \varphi \) 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 \( \varphi \) is an action expressed over the variables at \( m \) and \( n \);
3. **Control Constraint:** \( U_a \implies \text{wcp}(L_a, L_n) \) 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:** \( L_a \implies \text{unchanged}(v) \) 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 \( S = \langle M, \Phi \rangle \), the model refinement transformation first refines \( S \) 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 \( m \in N : \ L_m \leftarrow L_m \land \varphi \)

else for \( a \in A : \ L_a \leftarrow L_a \land \varphi \)

do

for \( a \in A : \ U_a \leftarrow U_a \land \text{wcp}(L_a, L_n) \)

for \( m \in N : \ L_m \leftarrow L_m \land \bigvee_{a=\langle m, n \rangle} \exists u. \ U_a \)

until \( L_m \) is unchanged for all nodes \( m \in N \).

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 \( M' \), we have \( [M'] \subseteq [M] \cap [\Phi] \), and in the case that the algorithm converges to a greatest fixpoint we have \( [M'] = [M] \cap [\Phi] \).
The constraints have definite form\(^3\) and the algorithm in [24] provides a more efficient control strategy that exploits dependencies between the constraints.

The derived initial condition is the final refined invariant \(L_m\) 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.

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 [13]. 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 \(\forall x.p(x)\) results in the weakest quantifier-free formula such that \(\forall x.p(x) \implies \forall x.p(x)\) with respect to background theory \(T\). When \(T\) admits quantifier elimination, then we have \(\forall x.p(x) \equiv \forall x.p(x)\), otherwise it underapproximates the quantified formula. Replacing \(wcp(L_a, L_n)\) in the Control Constraint by \(\forall st_n.wcp(L_a, L_n)\) 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 \(\forall\) yields the weakest sufficient condition would further increase the range of applicability of this approach. The techniques of abductive inference [22] and directed inference [27] 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. [16]. 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.

**Example: Packet Flow Control**

In this example, based on [26], 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

\[^3\] A constraint over a meet semilattice is *definite* or Horn-like if has the form \(v \leq g(x)\) for variable \(v\) and monotone function \(g\).
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:** $0 \leq buf \land 0 \leq out$

**Arc:** $a = \langle m_0, m_0 \rangle$

**action:** $Update(u, e) \triangleq -1 \leq u \leq 1 \land 0 \leq e \leq 4 \land buf' = buf + e - out \land out' = out + u$

**Required Properties**

$buf = 0$

$out = 0$

$\square 0 \leq buf \land buf \leq 20 \land 0 \leq out \land out \leq 4$

**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 $U(buf, out, u) \equiv -1 \leq u \leq 1 \land 0 \leq e \leq 4 \land buf' = buf + e - out \land out' = out + u$.

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

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

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

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

and the state invariant strengthens to

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

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

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

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:** $0 \leq out \leq 4 \land 0 \leq buf - out \leq 16 \land -3 \leq buf - 3 \ast out \leq 11 \land -6 \leq buf - 4 \ast out \leq 10$

**Arc:** $a = \langle m_0, m_0 \rangle$

**action:** $Update(u, e) \triangleq -1 \leq u \leq 1 \land 0 \leq out + u \leq 4 \land -6 \leq buf - 4 \ast u - 5 \ast out \leq 6$
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 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 4.1 we present a family of refinement-generating transformations that eliminate occurrences of time-bounded operators. In Section 4.3 we present a refinement-generating transformation that reduces path properties (expressed over a path of arcs in the model) to basic safety properties.

4.1 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 \( \text{start} \) 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 \( \text{start} \) implicit.

For a state or action predicate \( \varphi \), consider the following time-bounded operators:

1. \( \Diamond_k \varphi \) means that \( \varphi \) was true in some state no more than \( k \) time units in the past:
   \[ \sigma, i \vdash \Diamond_k \varphi \iff \text{there exists } j < i \text{ such that } \sigma[i](\text{start}) \leq \sigma[j](\text{start}) + k \text{ and } \sigma, j \vdash \varphi. \]
2. \( \Box_k \varphi \) means that \( \varphi \) will be true in some state no more than \( k \) time units in the future:
   \[ \sigma, i \vdash \Box_k \varphi \iff \text{there exists } j > i \text{ such that } \sigma[j](\text{start}) \leq \sigma[i](\text{start}) + k \text{ and } \sigma, j \vdash \varphi. \]
3. \( \square_k \varphi \) means that \( \varphi \) holds in each state for the next \( k \) time units:
   \[ \sigma, i \vdash \square_k \varphi \iff \text{for each } j > i \text{ such that } \sigma[j](\text{start}) \leq \sigma[i](\text{start}) + k \text{ we have } \sigma, j \vdash \varphi. \]
4. \( \lozenge_k \varphi \) means that \( \varphi \) held in each state over the previous \( k \) time units:
   \[ \sigma, i \vdash \lozenge_k \varphi \iff \text{for each } j < i \text{ such that } \sigma[i](\text{start}) \leq \sigma[j](\text{start}) + k \text{ we have } \sigma, j \vdash \varphi. \]
We extend the definition of basic safety property as follows: state/action predicates may include predicates of the form $\Diamond_k \varphi$ and $\Diamond_k \varphi$ where $\varphi$ is a state/action predicate.

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

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

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

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

1.2 replace the occurrence of $\Diamond_k \psi$ with $\text{start} \leq \text{lastpsi} + k$.

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

**Theorem 2.** If $S'$ is the result of applying Transformation 1 to specification $S$, then $S \subseteq S'$.

Proof: Since the transformation adds no new nodes or arcs, a simulation map $\xi$ 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 $\sigma \in [M'] \cap [\Phi']$ and consider the state $\sigma[i]$ for some $i \geq 0$. If $\sigma[i](\text{lastpsi}) = -\infty$ then there has been no prior state that satisfies $\psi$, so both $\sigma[i] \models \text{start} \leq \text{lastpsi} + k$ and $\sigma, i \models \Diamond_k \psi$ are false. Otherwise, let $j < i$ be the largest index such that $\sigma, j \models \psi$, then $\sigma[j + 1](\text{lastpsi}) = \sigma[j](\text{start})$ by the action of property (1.1.2). So we have

$$
\sigma, i \models \text{start} \leq \text{lastpsi} + k
\equiv \sigma[i](\text{start}) \leq \sigma[i](\text{lastpsi}) + k
\equiv \sigma[i](\text{start}) \leq \sigma[j + 1](\text{lastpsi}) + k
$$

lastpsi is protected and

$$
\sigma, i \models \Diamond_k \psi.
$$

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

11
4.1.2 Eliminating the Time-Bounded Future Operator $\diamond_k$

In the following transformation, we assume a variable $\psi\text{Deadlines} : \text{Set}(\text{Time})$ that represents a set of deadlines and has the operation $\text{min}(\psi\text{Deadlines})$ which returns the least element of the set, if one exists, otherwise $\infty$.\(^4\)

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

1. add fresh protected variable $\psi\text{Deadlines} : \text{Set}(\text{Time})$ to each $V(m)$ for node $m \in N$, and add to $\Phi$ the new goal formulas:
   1.1. $\psi\text{Deadlines} = \{\}$
   1.2. $\square \psi \implies \psi\text{Deadlines}' = \{\}$
   1.3. $\square \text{start} \leq \text{min}(\psi\text{Deadlines})$

2. replace the occurrence of $\diamond_k \psi$ in $\Phi$ with
   $\psi\text{Deadlines}' = \psi\text{Deadlines} \cup \{\text{start} + k\}$.

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

**Theorem 3.** If $S'$ is the result of applying Transformation 2 to specification $S$, then $S \subseteq S'$.

Proof: Since the transformation adds no new nodes or arcs, a simulation map $\xi$ 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 $\sigma \in [M'] \cap [\Phi']$ and consider arbitrary index $p \geq 0$ such that $\sigma[p+1](\psi\text{Deadlines}) \setminus \sigma[p](\psi\text{Deadlines}) = \{d\}$ for deadline $d$. State $\sigma[p+1]$ must have been produced by an action that implies $\psi\text{Deadlines}' = \text{insert}(d, \psi\text{Deadlines})$ arising from the replacement of a subformula $\diamond_k \psi$ where $d = \sigma[p](\text{start}) + k$. We now show that $\sigma, p \models \diamond_k \psi$. Let $r > p + 1$ be the least index such that $\sigma[r](\text{start}) > d$. If $d \in \sigma[r](\psi\text{Deadlines})$ then we have $\sigma, r \models \text{start} > d \geq \text{min}(\psi\text{Deadlines})$ which contradicts the state invariant (2.1.3). Consequently, $d \notin \sigma[r](\psi\text{Deadlines})$. Therefore $d$ was removed from $\psi\text{Deadlines}$ at some point $q \in [p+2, r-1]$.

But since $\psi\text{Deadlines}$ is protected, only the action 2.1.2 can change $\psi\text{Deadlines}$, specifically setting it to empty. Again by 2.1.2 this means that $\psi$ held at $q-1$, i.e. $\sigma, q - 1 \models \psi$. Since $\sigma[q](\text{start}) \leq d < \sigma[r](\text{start})$ by choice of $r$, we have $\sigma, p \models \diamond_k \psi$. This shows that whenever the action $\psi\text{Timer}' = \psi\text{Deadlines} \cup \{\text{start} + k\}$ takes place in an arbitrary trace, then we also have the property $\diamond_k \psi$; i.e. $\psi\text{Timer}' = \psi\text{Deadlines} \cup \{\text{start} + k\} \implies \diamond_k \psi$. Since we have only replaced positive occurrences of subformulas of the form $\diamond_k \psi$, $\Phi$ is monotone in those locations, and so we infer $\Phi' \implies \Phi$.

4.1.3 Eliminating the $\Box_k$ Operator

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

1. add a fresh protected variable $\text{lastNotPsi} : \text{Time}$ to $V(m)$ for each $m \in N$, which records the latest time at which $\neg \psi$ held, and add to $\Phi$ the new goal formulas

\(^4\) While for this specific operator $\diamond_k$, we could replace the set with just a $\text{nextDeadline}$ 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 = −∞
3.1.2 □ −ψ =⇒ lastnotpsi′ = start

3.2 replace the occurrence of □kψ with start > lastnotpsi + k.

Return the modified specification S′ = ⟨M′, Ψ′⟩.

Theorem 4. If S′ is the result of applying Transformation 3 to specification S, then S ⊑ 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 Ψ′ =⇒ Ψ.

Let σ ∈ [M′] ∩ [Ψ′] and consider the state σ[i] for some i ≥ 0. If σ[i](lastnotpsi) = −∞ then there has been no prior state that satisfies −ψ, so both (σ, i) ⊨ start > lastnotpsi + k and (σ, i) ⊨ □kψ are true. Otherwise, let j < i be the largest index such that (σ, j) ⊨ −ψ, then σ[j + 1](lastnotpsi) = σ[j](start) by the action of property (3.1.2). So we have

σ, i ⊨ start > lastnotpsi + k
  ≡ σ[i](start) > σ[i](lastnotpsi) + k
  ≡ σ[i](start) > σ[j + 1](lastnotpsi) + k
  ≡ σ[i](start) > σ[j](start) + k
  ≡ σ, i ⊨ □j −ψ

From this inference, we conclude generally that in the context of any trace of [M′] ∩ [Ψ′] that the state predicate start > lastnotpsi + k holds exactly when the property □kψ holds; i.e.

start > lastnotpsi + k ≡ □kψ.

Since we have formed Ψ′ by modifying positive occurrences of subformulas of the form □kψ, Ψ is monotone in those locations, and so we infer Ψ′ =⇒ Ψ (or more strongly Ψ′ ⊑ Ψ).

4.1.4 Eliminating the □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 −∞.

Transformation 4. Given a specification S = ⟨M, Ψ⟩, transform S as follows: for each positive occurrence of a subformula of the form □kψ in Ψ:

4.1 add fresh protected variable notpsiESTs : Set(Time) to each V(m) for node m ∈ N, and add to Ψ the new goal formulas:
  4.1.1 notpsiESTs = {}  
  4.1.2 □ −ψ =⇒ notpsiESTs′ = {}  
  4.1.3 □ start ≤ max(notpsiESTs) =⇒ ψ  

4.2 replace the occurrence of □kψ in Ψ with notpsiESTs′ = notpsiESTs ∪ {start + k}.
Return the modified specification $S' = \langle M', \Phi' \rangle$.

**Theorem 5.** If $S'$ is the result of applying Transformation 4 to specification $S$, then $S \subseteq S'$.

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

$$\text{notpsiESTs}' = \text{notpsiESTs} \cup \{\text{start} + k\} \implies \Box_k \psi.$$ Since we have only replaced positive occurrences of subformulas of the form $\Box_k \psi$, $\Phi$ is monotone in those locations, and so we infer $\Phi' \implies \Phi$.

4.2 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:** token, lock : Boolean

$k : \text{Time}$

**invariant:** true

**Arc:** $a = \langle m_0, m_0 \rangle$

**actions:**
- Insert $\triangleq \neg\text{token} \land \text{token'}$
- Remove $\triangleq \text{token} \land \neg\text{token'}$
- Lock $\triangleq \text{lock'}$
- Unlock $\triangleq \neg\text{lock'}$

**Required Properties**

- $\Box \text{Insert} \implies \Diamond_k \text{Unlock}$
- $\Box \text{Unlock} \implies \Diamond_k \text{Insert}$

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 $\text{SecureEnclave}_0 \sqsubseteq \text{SecureEnclave}_1$ where $\text{SecureEnclave}_1$ has no occurrence of time-bounded temporal operators in its required properties.

**Specification** $\text{SecureEnclave}_1$

**Node:** $m_0$

- **vars:** $\text{token}, \text{lock} : \text{Boolean}$
- $k : \text{Time}$

- **protected vars:** $\text{unlockDeadlines} : \text{Set(Time)}$
  - $\text{lastInsert} : \text{Time}$

- **invariant:** $\text{true}$

**Arc:** $a = (m_0, m_0)$

- **actions:**
  - $\text{Insert} \triangleq \neg\text{token} \land \text{token}'$
  - $\text{Remove} \triangleq \text{token} \land \neg\text{token}'$
  - $\text{Lock} \triangleq \text{lock}'$
  - $\text{Unlock} \triangleq \neg\text{lock}'$

**Required Properties**

- $\text{lastInsert} = -\infty$
- $\text{unlockDeadlines} = \{\}$
- $\square \text{Insert} \implies \text{lastInsert}' = \text{start}$
- $\square \text{Unlock} \implies \text{start} - k \leq \text{lastInsert}$
- $\square \text{Unlock} \implies \text{unlockDeadlines}' = \{\}$
- $\square \text{start} \leq \min(\text{unlockDeadlines})$
- $\square \text{Insert} \implies \text{unlockDeadlines}' = \text{unlockDeadlines} \cup \{\text{start} + k\}$

**Theorems**

- $\square \text{Insert} \implies \Diamond_k \text{Unlock}$
- $\square \text{Unlock} \implies \Diamond_k \text{Insert}$

**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** $\text{SecureEnclave}_2$

**Node:** $m_0$

- **vars:**
  - $\text{token} : \text{Boolean} = \text{false}$
  - $\text{lock} : \text{Boolean} = \text{true}$
  - $k : \text{Time}$

- **protected vars:** $\text{unlockDeadlines} : \text{Set(Time)} = \{\}$
  - $\text{lastInsert} : \text{Time} = -\infty$

- **invariant:** $\text{start} \leq \min(\text{unlockDeadlines})$
Arc: \( a = \langle m_0, m_0 \rangle \)

actions:

- \( \text{Insert} \equiv \neg \text{token} \land \text{token'} \land \text{lastInsert'} = \text{start} \land \text{unlockDeadlines}' = \text{unlockDeadlines} \cup \{ \text{start} + k \} \)
- \( \text{Remove} \equiv \text{token} \land \neg \text{token'} \land \text{unchanged(lastInsert)} \land \text{unchanged(unlockDeadlines)} \)
- \( \text{Lock} \equiv \text{lock'} \land \text{unchanged(lastInsert)} \land \text{unchanged(unlockDeadlines)} \)
- \( \text{Unlock} \equiv \text{start} - k \leq \text{lastInsert} \land \neg \text{lock'} \land \text{unlockDeadlines}' = \{ \} \land \text{unchanged(lastInsert)} \)

Required Properties

Theorems

- \( \square \text{Insert} \implies \Diamond_k \text{Unlock} \)
- \( \square \text{Unlock} \implies \Diamond_k \text{Insert} \)

End Specification

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

4.3 Path Properties

Some required properties are naturally expressed over the endpoints of a path in the model, rather than just state properties and actions. 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.

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 \( M \). Our strategy is to propagate \( \varphi \) through the structure of \( M \) until we have inferred properties that can be localized to the nodes and arcs of \( M \). For purposes of reasoning about path properties we proceed as if we have path labels in \( M \) 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 approaches that naturally arise: 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 \( S = \langle M, \Phi \rangle \) 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 \( S \) to reduce a path property as follows: (1) Delete \( \varphi_{m,p} \) from \( \Phi \), and (2) for each arc \( a = \langle m, n \rangle \in \text{Arc} \), add the path formula \( \text{wcPostSpec}(L_a, \varphi_{m,p}) \) to
\(\phi\) where \(wcPostSpec(L_a, \theta)\) is the Weakest Controllable PostSpecification of action \(L_a\) with respect to path formula \(\theta\) over \(V(m) \cup V(p)\) and is defined by

\[
wcPostSpec(L_a, \theta) \equiv \forall st_m, u, e, st_n. L_m(st_m) \land U(st_m, u) \land e \in E(st_m) \land st_n = f_a(st_m, u, e) \implies \theta.
\]

\(wcPostSpec\) 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 \(S = \langle M, \Phi \rangle\) 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 \(S\) to reduce the path property occurrences as follows: (1) Delete \(\varphi_{m,p}\) from \(\Phi\), 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 \(\Phi\) where \(wcPreSpec(L_a, \theta)\) is the Weakest Controllable PreSpecification of action \(L_a\) with respect to path formula \(\theta\) 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 \(\varphi\) 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.

### 4.4 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 [28]. 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

\[
F(x : D) : (z : R) = \text{if } \text{primitive}(x) \text{ then } \text{direct}(x) \text{ else } \text{compose } \circ (F \times F) \circ \text{decompose}(x)
\]

and the required property is \(bag(x) = bag(z) \land \text{ordered}(z)\), where \(x\) and \(z\) are lists of numbers, \(bag(x)\) returns the bag or multiset of elements in list \(x\), and \(\text{ordered}(z)\) 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:

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

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 \( z_0 \): \( \varphi(x_0,z_0) \equiv \text{\textit{bag}}(x_0) = \text{\textit{bag}}(z_0) \land \text{\textit{ordered}}(z_0) \). Suppose that we follow the second tactic and refine the model by choosing list concatenation as our `compose` operator: \( \text{\textit{compose}} \mapsto z_0 = z_1 \cdot z_2 \). 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:

\[
\text{\textit{wcPreSpec}}(\text{\textit{compose}}, \varphi(x_0,z_0)) =
\begin{align*}
&\equiv \forall z_1, z_2. \text{\textit{bag}}(x_1) = \text{\textit{bag}}(z_1) \land \text{\textit{ordered}}(z_1) \& \text{\textit{bag}}(x_2) = \text{\textit{bag}}(z_2) \land \text{\textit{ordered}}(z_2) \\
&\equiv \text{\textit{bag}}(x_0) = \text{\textit{bag}}(z_1) \cup \text{\textit{bag}}(z_2) \land \text{\textit{ordered}}(z_1) \land \text{\textit{ordered}}(z_2) \land \text{\textit{bag}}(z_1) \leq \text{\textit{bag}}(z_2).
\end{align*}
\]

where we have used domain-specific laws for distributing \text{\textit{bag}} and \text{\textit{ordered}} over list concatenation, and \( b_1 \leq b_2 \) holds when each element of \text{\textit{bag}} \( b_1 \) is less than or equal to each element of \text{\textit{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:

\[
\text{\textit{wcPreSpec}}(F \times F, \varphi(x_0,z_1,z_2)) =
\begin{align*}
&\equiv \forall z_1, z_2. \text{\textit{bag}}(x_1) = \text{\textit{bag}}(z_1) \land \text{\textit{ordered}}(z_1) \land \text{\textit{bag}}(x_2) = \text{\textit{bag}}(z_2) \land \text{\textit{ordered}}(z_2) \\
&\implies \text{\textit{bag}}(x_0) = \text{\textit{bag}}(z_1) \cup \text{\textit{bag}}(z_2) \land \text{\textit{ordered}}(z_1) \land \text{\textit{ordered}}(z_2) \land \text{\textit{bag}}(z_1) \leq \text{\textit{bag}}(z_2) \\
&\equiv \text{\textit{bag}}(x_0) = \text{\textit{bag}}(x_1) \cup \text{\textit{bag}}(x_2) \land \text{\textit{bag}}(x_1) \leq \text{\textit{bag}}(x_2).
\end{align*}
\]

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

\[
\text{\textit{bag}}(x_0) = \text{\textit{bag}}(x_1) \cup \text{\textit{bag}}(x_2) \land \text{\textit{bag}}(x_1) \leq \text{\textit{bag}}(x_2).
\]

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 \( \text{length}(x_0) > 1 \) on \textit{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 \textit{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 [27], which also derives several other sorting algorithms. Derivation of several parallel sorting algorithms via divide-and-conquer may be found in [29].

5 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 (\( \lozenge \varphi \)) and reactivity properties (\( \Box \lozenge \varphi \)) can also be handled as definite constraint systems whose fixpoints can be found by Kleene iteration combined with widening.

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 [30]. 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 [20, 28].

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.
Acknowledgements: We would like to thank Allesandro Coglio and Christoph Kreitz for their comments on this paper.

References


Appendix: Secure Enclave Example in TLA+

The specification SecureEnclave1 in Section 4.2 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)

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

=============================================================================