Deductive Model Refinement

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

Abstract. Deductive model refinement (hereafter simply "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, action, and path invariants. We have run a prototype implementation of model refinement based on the Z3 SMT solver over a variety of system and algorithm design problems.

Keywords: Refinement · model-based design · formal specification · reactive synthesis · function synthesis · program synthesis

1 Introduction

Formal specifications characterize the acceptable behaviors of a desired program. Among the various means for specifying requirements on a desired program are (1) *logical specifications* in which predicates expressed in a suitable logic decide the desired behaviors, and (2) *models* whose computable behaviors are a superset or overapproximation of the desired behaviors. We view logical specifications and models as having complementary strengths and believe that their combination can lead to simpler and more natural specifications of systems and algorithms.

This paper comes from a complementary line of formal development research that explores automatic transformations that map a specification or intermediate design to an equivalent form or to a correct-by-construction refinement. The field of *program synthesis* focuses on automatic generation of programs from formal specifications using such transformations. Obviously, any formal development process will benefit to the extent that refinement steps can be automatically

generated. Human insight is still needed to choose appropriate transformations and to specify how to apply them.

In this paper we propose a unifying synthesis framework, called *deductive model refinement* (or simply model refinement), that starts with a formal specification comprised of models and logical properties. From a given model and logical properties, we define a constraint system that characterizes refinements of the model that satisfy the logical properties. Solving the constraint system corresponds to eliminating undesired behaviors from the model. Model refinement serves to unify and extend previous work on function/algorithm synthesis with reactive system synthesis.

Given a model $\mathcal{M}$ that overapproximates desired 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 (where the refinement relation defines a lattice of models). 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 any acceptable behaviors.

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 software system design, the model captures the APIs and possible operations of a component and perhaps a restricted grammar for expressing programs [1]. In general system design, a model can express a system design pattern [9, 4, 22]. In algorithm design, a model can reflect the imposition of a parametric solution pattern, such as an algorithm theory [29] or a sketch [34].

Model refinement techniques are common in science and engineering. Our approach is called *deductive model refinement* due to the use of deductive techniques to enforce logical requirement properties on a model. Examples of data-driven or inductive model refinement can be found in (1) statistical model estimation techniques that fit, say, a Bayesian Network model to given data, and (2) machine learning techniques for refining an artificial neural network model with training data. In these examples the model provides the abstract computational pattern and the data provides the requirements on the refined model.

This paper focuses on enforcement of basic safety properties. In [30], we introduce a wider fragment of temporal logic that can be reduced to the basic safety fragment. Most current work on the synthesis of reactive systems focuses on circuit design and starts with specifications in propositional Linear Temporal Logic (LTL) [3, 13]. It is therefore limited to finite state models. Our approach to model refinement allows specifications that are first-order and uses a temporal logic of action (similar to TLA [15]) that is amenable to refinement, allowing possibly-infinite state spaces and 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 checkable proofs. One barrier to automation is the computational complexity of formula simplification in the application domain theories that support the 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 Z3-based prototype to perform model refinement on a variety of examples, each solvable in a few seconds or minutes.

Our novel contributions include

1. a uniform framework for specifying algorithms and reactive systems by a combination of overapproximating 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 [28].

After introducing basic concepts, the paper first focuses on reactive system synthesis as constraint-solving via iterated constraint propagation, with examples. Function specifications that arise during reactive system synthesis then provide a natural segue into a treatment of function/algorithm synthesis as generalized iterated constraint propagation over paths.

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 [15]. 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 $\sigma, n \vDash \varphi$, 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, of course, support models that have both global and local variables.

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 [35]. 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.

Semantics. A *trace* is a possibly 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* $\xi : \mathcal{M}_2 \to \mathcal{M}_1$ 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: $\xi(m_0^{\mathcal{M}_2}) = m_0^{\mathcal{M}_1}$;
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, *arc 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 5.1.

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:

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

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) $\xi : \mathcal{M}_2 \to \mathcal{M}_1$ is a simulation map, and (3) $\Phi_2 \implies \Phi_1$, then $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$.

The proof, given in [30], shows how ξ induces a map $\hat{\xi}$ of traces of $\mathcal{S}_2$ such that $\hat{\xi}([\![\mathcal{S}_2]\!]) \subseteq [\![\mathcal{S}_1]\!]$.

3 Model Refinement as Constraint Solving

Model refinement transforms a model $\mathcal{M}$ and required properties Φ into a model $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq \mathcal{M}' \wedge \mathcal{M}' \vDash \Phi$. 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 discuss 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 is typically of infinite height for nonpropositional logics.

For arc $a = \langle m, n \rangle$, arc label L_a, and node label L_n, let $wcp(L_a, L_n)$ be the *weakest controllable predecessor* predicate transformer which 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

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

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

Fig. 1: Model Refinement Algorithm

$wcp(L_a, L_n)$ 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 of 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.

The constraint system is comprised of the following four sets of 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 predicate 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 $m \in N$.

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 of a node with respect to all of its outgoing arcs. Given a specification $\mathcal{S} = \langle M, \Phi\rangle$, the model refinement transformation refines the specification by solving the constraint system. In other words, a solution to the constraints is a model that refines the input model and the solution process generates a refinement.

A straightforward algorithm for solving the constraint system over the labels on a model is presented in Figure 1. The iteration converges to a fixpoint when the labels do not change in an iteration. 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 algorithm in [21] 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 nonblocking 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.

The correctness of this algorithm is a consequence of Tarski's theorem. Each constraint has definite form $v \leq g(v)$ where g is monotone, 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. 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 [21]. Techniques to improve the complexity of the algorithm and to guarantee convergence to a fixpoint are further discussed in [30].

Example: Packet Flow Control

In this example, based on [23], 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 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 (1) the global state variables, (2) their initial invariants, (3) the one node, (4) the one arc and its initial action (dependent on environment input e and control value u), (5) the required safety properties, and (6) currently known theorems, which are empty here but are extended by the model refinement process.

Specification FC0
  Vars: $buf, out : Integer$
  Invariant: $0 \leq buf \wedge 0 \leq out$
  Node: m_0
  Arc: $a = \langle m_0, m_0 \rangle : -1 \leq u \leq 1 \ \wedge \ 0 \leq e \leq 4$
    $\wedge \ buf' = buf + e - out \ \wedge \ out' = out + u$
  Required Properties
    $buf = 0$
    $out = 0$
    $\Box \ 0 \leq buf \ \wedge \ buf \leq 20 \ \wedge \ 0 \leq out \ \wedge \ out \leq 4$
  Theorems
End Specification

The first two required properties determine the initial state values. For the last required property, the algorithm in Figure 1 instantiates *wcp* to generate the following formula as an upper bound on the control condition

$$\begin{aligned} U(buf, out, u) \;\equiv\; & -1{\leq}u{\leq}1 \;\wedge\; 0{\leq}out+u{\leq}4 \\ & \wedge\, \forall e.\, 0{\leq}e{\leq}4 \implies 0{\leq}buf+e-out{\leq}20. \end{aligned}$$

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 become theorems of the model, as can be verified by a model checker.

Specification FC1
  Vars: $buf, out : Integer = 0$
  Invariant: $0{\leq}out{\leq}4 \wedge 0{\leq}buf{-}out{\leq}16$
    $\wedge -3{\leq}buf{-}3*out{\leq}11 \wedge -6{\leq}buf{-}4*out{\leq}10$
  Node: m_0
  Arc: $a = \langle m_0, m_0 \rangle :\; -1{\leq}u{\leq}1 \wedge\; 0{\leq}out+u{\leq}4 \wedge\; 0{\leq}e{\leq}4$
    $\wedge\; -6{\leq}buf-4*u-5*out{\leq}6$
    $\wedge\; -1{\leq}buf-2*u-3*out{\leq}9$
    $\wedge\, buf' = buf{+}e{-}out \wedge\; out' = out{+}u$
  Theorems: $buf = 0 \wedge out = 0 \wedge \Box\, (0{\leq}buf{\leq}20 \wedge 0{\leq}out{\leq}4)$
End Specification

Again, note how the Required Properties of the initial model have been transformed into Theorems of the refined model, by construction. The strengthened state invariant on node m_0 is also the derived initial condition and specifies the set of 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. For game-like problems, this is also known as extracting a winning strategy for the system game modulo the derived initial conditions.

Our prototype model refinement system converges on the model above in a few seconds. 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.

Other Examples

The Cinderella-Stepmother game has been posed as a challenge problem for synthesis systems (cf. [2]). It is a turn-based game between Cinderella and her wicked stepmother. The game centers on a ring of five buckets that can each hold up to c units of water, where initially the buckets are empty. In each round of the game the stepmother adds one unit of water distributed over the buckets, and then Cinderella empties two adjacent buckets. If any of the buckets ever overflow, then the stepmother wins, otherwise Cinderella wins.

Specification Cinderella-Stepmother
Vars: $b_0, b_1, b_2, b_3, b_4, c : NonNegativeReal$
$e_0, e_1, e_2, e_3, e_4 : NonNegativeReal$
Invariant: $\bigwedge_{i=0,4} b_i = 0$
Nodes: m_C, m_S
Arc: $Add = \langle m_S, m_C \rangle : \sum_{i=0,4} b'_i = 1 + \sum_{i=0,4} b_i$
$\wedge \ \bigwedge_{i=0,4} b'_i = b_i + e_i$
Arc: $Empty = \langle m_C, m_S \rangle : b'_u = 0 \wedge b'_{(u+1)\%5} = 0$
Required Properties: $\Box \bigwedge_{i=0,4} b_i \leq c$
End Specification

Our specification for this game-like problem has two nodes, one for the turn or each player. The game is parametric on a real value $c > 0$ used to define the Stepmother's (antagonist's) task. In [2], a controller for the game is found using sketches as hints to the solver. It is conjectured that automatic solutions (i.e. without human-provided hints) are "unrealistic" for values of c in range [1.5,3] (the problem is relatively easy outside that range). Our model refinement prototype automatically generates winning strategies in that range using roughly a minute of CPU time.

Other problems for which we have synthesized code include the classic readers-writers problem, elevator control, model-repair [2], and a reactive controller for a secure enclave. The latter problem has time-bounded responsive requirements and in [30] we introduce transformations that reduce a collection of time-bounded temporal operators to basic safety properties. After that reduction then the model refinement algorithm can be applied.

4 Function Synthesis

Model refinement naturally gives rise to the specification of several functions. For example, in the final model of the Flow Control example, the action constrains the control choice u to satisfy

$$-1 \leq u \leq 1 \ \wedge\ 0 \leq out + u \leq 4 \ \wedge\ -6 \leq buf - 4*u - 5*out \leq 6 \ \wedge\ -1 \leq buf - 2*u - 3*out \leq 9. \tag{1}$$

To make that choice, we must synthesize a control function

$$FlowControl(\langle buf, out \rangle) = u \ such \ that \ (1).$$

Generally, for each arc $a = \langle m, n \rangle$ in the final model, model refinement generates a specification for a control function for a that outputs a satisfying control value. The desired control function may be specified as

$$Control_a(st \mid L_m(st)) \ = \ u \ such \ that \ U_a(st, u)$$

where $L_m(st)$ is the precondition and $U_a(st, u)$ is the postcondition. Algorithm or function synthesis is appropriate for this specification, since the behaviors are specified by a simple input-output relation which we treat as a safety property over traces of length 2 (input state followed by output state). A variety of techniques for function synthesis have been developed, many of which stem from the original work on deductive synthesis [12, 6, 17]. Later approaches to synthesis of functions exploit algorithm design patterns [29], sketches [34], and transformations from high-level function definitions [19].

Here, since the control variable u only takes on three values, a simple transformation to form a conditional function can be applied resulting in the following (see [30] for details).

$$\begin{array}{l} controlFun(\langle buf, out \rangle \mid 0 \leq out \leq 4 \wedge 0 \leq buf - out \leq 16 \\ \qquad\qquad\qquad\qquad \wedge -3 \leq buf - 3*out \leq 11 \wedge -6 \leq buf - 4*out \leq 1)) = \\ \quad \textbf{if } 4*out - buf > -5 \ \wedge \ 2*out - buf > -12 \\ \qquad\qquad\qquad\qquad \wedge out \geq 1 \ \wedge \ 5*out - buf > -3 \textbf{ then } \ -1 \\ \quad \textbf{else if } buf - 2*out > 0 \ \wedge \ out \leq 3 \textbf{ then } \ 0 \\ \quad \textbf{else } 1 \end{array}$$

5 Path Properties

In the previous section we discussed how function specification naturally arises during model refinement. In this section we present a general approach to function synthesis that generalizes the model refinement approach. Reactive synthesis tends to generate control systems with relatively flat structure whereas

algorithm/function design generates smaller programs with deeper structure (via a hierarchy of subfunctions). Our intent is to have model refinement as the unifying framework for synthesizing both reactive systems and functions.

Some required properties are naturally expressed over the nodes of a path in the model, rather than being localized to a node (state invarient) or arc (action invariants). They express required properties that hold between values that are not near in time or space. We define *path properties* to be predicates over the variables of nodes along some path in the model. An action property is a special case of a path property since it is expressed over a path of length one. When necessary we prefix a variable with the node at which the value is referenced. If a variable is only accessible at one node or arc (i.e. it is local), the prefix can be omitted.

We define next some refinement rules that can be used to reduce path properties to action properties. The refinement 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 via arc refinement, 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. 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 action properties 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 node p. We can refine $\mathcal{S}$ to reduce the path property $\varphi_{m,p}$ 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

$$\forall st_m, u, e.\, L_m(st_m) \ \wedge\ U(st_m, u) \ \wedge\ e \in E(st_m) \implies \theta(f_a(st_m, u, e)).$$

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 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

$$\forall u, e.\, L_n(st_n) \ \wedge\ U(st_n, u) \ \wedge\ e \in E(st_n) \implies \theta(st_m, f_a(st_n, u, e)).$$

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, L_a must express a nontrivial constraint. These rules are often applied after one has chosen a candidate function/operation for transition a and then desires to infer 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 an arc refinement, enables the propagation to go through. This is a choice and alternative choices lead to different designs, as illustrated in the next section.

5.1 Algorithm Design Example: Sorting

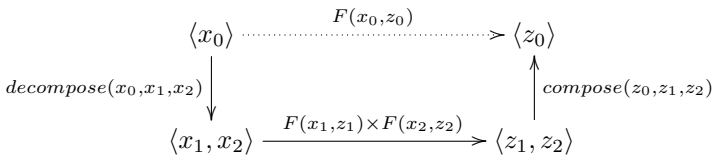
One feature of model refinement is that it subsumes a major part of the automated algorithm design work performed in earlier function synthesis systems such as KIDS [25]. In retrospect, the success of KIDS in algorithm design is partly due to its automated inference system which was designed to propagate output conditions through the structure of a chosen program scheme (i.e. an overapproximating model representing an algorithm class). 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

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

and the required property is $bag(x){=}bag(z) \wedge ordered(z)$, 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:



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:

$wcPreSpec(compose, \varphi(x_0, z_0))$
$\equiv \quad \forall z_0.\ z_0 = z_1{+\!+}z_2 \implies bag(x_0) = bag(z_0) \wedge ordered(z_0)$
$\equiv \quad$ { quantifier elimination on z_0 }

$$
\begin{array}{ll}
 & bag(x_0) = bag(z_1 {+\!+} z_2) \wedge ordered(z_1 {+\!+} z_2) \\
\equiv & \quad \{ \text{ distributivity laws and simplification} \} \\
 & bag(x_0) = bag(z_1) \cup bag(z_2) \wedge \ ordered(z_1) \wedge ordered(z_2) \wedge bag(z_1) \leq bag(z_2).
\end{array}
$$

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:

$$
\begin{array}{ll}
 & wcPreSpec(F \times F,\ \varphi(x_0, z_1, z_2)) \\
\equiv & \quad \{ \text{ unfold } \} \\
 & \forall z_1, z_2.\ bag(x_1) = bag(z_1) \wedge ordered(z_1) \wedge bag(x_2) = bag(z_2) \wedge ordered(z_2) \\
 & \qquad \Longrightarrow\ bag(x_0) = bag(z_1) \cup bag(z_2) \\
 & \qquad\quad \wedge\ ordered(z_1) \wedge ordered(z_1) \wedge bag(z_1) \leq bag(z_2) \\
\equiv & \quad \{ \text{ simplification and quantifier elimination} \} \\
 & bag(x_0) = bag(x_1) \cup bag(x_2) \wedge bag(x_1) \leq bag(x_2).
\end{array}
$$

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 sub-algorithm 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.

6 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 [29] were implemented in KIDS [25] and Specware [14]. These synthesis systems used a form of model refinement to instantiate algorithm models for divide-and-conquer [24], global search, dynamic programming [26], and other classes. Synthesized applications include schedulers [31], SAT-solvers [33], and garbage collectors [32].

Sketching [34] 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 [1], 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 [20, 8] and reactive system design [18]. 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) [3, 13]. Model refinement allows specifications that are first-order and uses a temporal logic of action that is amenable to refinement, allowing a broader range of applications to be tackled.

The algorithm derivation in Section 5 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 arc 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 refinement rules in Section 5 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/action specifications, 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 [7, 10, 11]. 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.

Model checking [5] can be viewed as a special case of model refinement in which refinement of the model is not an option. Counter-example-driven model refinement is performed in CEGAR with the goal to prove a specific property on a model of a fixed underlying program. The goal of CEGAR is not to synthesize a correct program from properties but to verify properties of a given program.

7 Concluding Remarks

The starting point of this work is the observation that logical properties and computational models have complementary strengths for purposes of formally stating requirements on desired computer behavior. The key questions then are (1) how to combine these strengths into a coherent formalism for specifying requirements, and (2) how to calculate programs that are consistent with specifications stated in the formalism.

In this paper we have presented an instance of these general ideas, using (1) a first-order temporal logic of actions to specify logical properties and (2) labelled transition systems to express concrete and abstract computational models and their refinement order. For illustration purposes we have further focused on basic safety properties.

Physical plants (as in the Flow Control problem) and game-like problems (as in the Cinderella-Stepmother problem) provide concrete models upon which model refinement can propagate logical properties over actions. The imposition of abstract designs as abstract models (as in the Sorting example and more generally in the form of design patterns, algorithm schemas, sketches, etc.) naturally transforms property specifications into path predicates over abstract models. This paper has presented model refinement as a unified treatment of reactive and functional design using iterated constraint propagation of (1) path predicates over abstract models and (2) state/action predicates over concrete model steps/arcs.

While this paper 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 automated 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 (and therefore amenable to the techniques in this paper).

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 [27, 32]. 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 [16, 25].

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.

We are currently working on the design of a processor (model) that asynchronously receives and processes tasks for which we want to enforce capacity and timeliness properties. To enforce fairness and timeliness the design composes in an abstract scheduler (design pattern). We hope to report on this work in a future paper.

Acknowledgments. The authors would like to thank Alessandro Coglio, Grant Jurgenson, Christoph Kreitz, and the reviewers for their comments on this paper. 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.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthe-

sis. In: Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD). pp. 1–17 (2013)
2. Beyene, T., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 221233 (2014)
3. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012)
4. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley (1996)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
6. Constable, R.L.: Constructive mathematics and automatic program writers. In: Information Processing 71. pp. 229–233. IFIP, Ljubljana, Yugoslavia (August 23–28, 1971)
7. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8603011
8. Filippidis, I., Dathathri, S., Livingston, S.C., Ozay, N., Murray, R.M.: Control design for hybrid systems with tulip: The temporal logic planning toolbox. In: 2016 IEEE Conference on Control Applications (CCA). pp. 1030–1041 (2016)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
10. Govind, V., Shoham, S., Gurfinkel, A.: Solving constrained horn clauses modulo algebraic data types and recursive functions. Proc. ACM Program. Lang. **6**(POPL) (2022)
11. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 405416 (2012)
12. Green, C.: Application of theorem proving to problem solving. In: Proceedings of the First International Joint Conference on Artificial Intelligence. pp. 219–239 (1969)
13. Jacobs, S., Klein, F., Schirmer, S.: A high-level ltl synthesis format: Tlsf v1.1. Electronic Proceedings in Theoretical Computer Science **229**, 112132 (2016)
14. Kestrel Institute: Specware System and documentation (2003), http://www.specware.org/
15. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16**(3), 872–923 (1994)
16. Liu, Y.: Systematic Program Design: From Clarity to Efficiency. Cambridge University Press (2013)
17. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems **2**(1), 90–121 (January 1980)
18. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 179190 (1989)
19. Püschel, M., Moura, J.M.F., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: Spiral: A generator for platform-adapted libraries of signal processing algorithms. Int. J. High Perform. Comput. Appl. **18**(1), 2145 (feb 2004)
20. Ramadge, P., Wonham, W.: The control of discrete event systems. Proceedings of the IEEE **77**(1), 81–98 (1989)
21. Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. Science of Computer Programming **35**, 191–221 (1999)

22. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. Wiley (2000)
23. Slanina, M., Sankaranarayanan, S., Sipma, H., Manna, Z.: Controller synthesis of discrete linear plants using polyhedra. Tech. rep., Technical Report REACT-TR-2007-01, Stanford University (2007)
24. Smith, D.R.: Top-down synthesis of divide-and-conquer algorithms. Artificial Intelligence **27**(1), 43–96 (September 1985), (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.)
25. 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), 1024–1043 (1990), citeseer.ist.psu.edu/article/smith90kids.html
26. Smith, D.R.: Structure and design of problem reduction generators. In: Möller, B. (ed.) Constructing Programs from Specifications, pp. 91–124. North-Holland, Amsterdam (1991)
27. Smith, D.R.: Generating programs plus proofs by refinement. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments. pp. 182–188. Springer-Verlag LNCS 4171 (2008)
28. Smith, D.R.: Model refinement code (2021), https://github.com/KestrelInstitute/modelRefinement
29. Smith, D.R., Lowry, M.R.: Algorithm theories and design tactics. In: van de Snepscheut, L. (ed.) Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375, pp. 379–398. Springer-Verlag, Berlin (1989), (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321)
30. Smith, D.R., Nedunuri, S.: Model refinement. Tech. Rep. Technical Report 21.0, Kestrel Institute (2021), https://www.kestrel.edu/people/smith/pub/MR-TR.pdf
31. Smith, D.R., Parra, E.A., Westfold, S.J.: Synthesis of planning and scheduling software. In: Tate, A. (ed.) Advanced Planning Technology. pp. 226–234. AAAI Press, Menlo Park (1996)
32. Smith, D.R., Westbrook, E., 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
33. Smith, D.R., Westfold, S.: Toward the Synthesis of Constraint Solvers. Tech. rep., Kestrel Institute (2013), http://www.kestrel.edu/home/people/smith/pub/CW-report.pdf
34. Solar-Lezama, A.: The sketching approach to program synthesis. In: Programming Languages and Systems, 7th Asian Symposium, APLAS. Lecture Notes in Computer Science, vol. 5904, pp. 4–13. Springer (2009)
35. Sontag, E.: Mathematical Control Theory. Springer (1998)