Incremental Realization of Safety Requirements: Non-determinism vs. Modularity

Ali Ebnenasir

▶ **To cite this version:**

Ali Ebnenasir. Incremental Realization of Safety Requirements: Non-determinism vs. Modularity. 6th Fundamentals of Software Engineering (FSEN), Apr 2015, Tehran, Iran. pp.159-175, 10.1007/978-3-319-24644-4_11 . hal-01446598

HAL Id: hal-01446598

https://inria.hal.science/hal-01446598v1

Submitted on 26 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Incremental Realization of Safety Requirements: Non-Determinism vs. Modularity*

Ali Ebnenasir

Department of Computer Science
Michigan Technological University, Houghton MI 49931, U.S.A.
aebnenas@mtu.edu

Abstract. This paper investigates the impact of non-determinism and modularity on the complexity of incremental incorporation of safety requirements while preserving liveness (a.k.a. the problem of *incremental synthesis*). Previous work shows that realizing safety in non-deterministic programs under limited observability is an NP-complete problem (in the state space of the program), where *limited observability* imposes read restrictions on program components with respect to the local state of other components. In this paper, we present a surprising result that synthesizing safety remains an NP-complete problem even for deterministic programs! The results of this paper imply that non-determinism is not the source of the hardness of synthesizing safety in concurrent programs; instead, limited observability has a major impact on the complexity of realizing safety. We also provide a roadmap for future research on exploiting the benefits of modularization while keeping the complexity of incremental synthesis manageable.

Keywords: Program Synthesis, Safety Specifications, Non-Determinism, Modularity

1 Introduction

Understanding the complexity of realizing new (safety/liveness) properties is of paramount importance since today's systems often have to adapt to new requirements while preserving some existing functionalities. *Safety* stipulates that nothing bad ever happens (e.g., at most one process/thread accesses shared resources at any moment), and *liveness* states that something good will eventually occur (e.g., each process eventually gets access to shared resources). New requirements are raised due to changes in platform, environmental faults, design flaws, new user requirements (e.g., non-functional concerns), porting, etc. Thus, it is important to enhance our understanding of what complicates behavioral changes. Towards this end, this paper investigates the complexity of *redesigning* finite-state programs towards capturing new safety requirements while preserving liveness, called the *problem of incremental synthesis*.

* This work was sponsored in part by the National Science Foundation grant CCF-1116546.

Several approaches exist for capturing safety most of which lack a thorough complexity analysis. For example, aspect-oriented approaches [5, 17, 16, 10, 13, 6] provide a method for capturing and verifying cross-cutting functionalities. Control-theoretic techniques [25, 12] realize new safety requirements by generating controllers that implement safety in different components of a system. Techniques based on transformation automata [24] enforce safety and/or security policies. Our previous work [7, 3] shows that incremental synthesis for non-deterministic programs can be done in polynomial time (in the size of program state space) if we consider *unlimited observability*, where program components/processes can atomically read the state of other processes. Nonetheless, the authors of [4] demonstrate that incremental synthesis of safety in non-deterministic programs under limited observability is NP-complete (in the size of the program state space). *Limited observability* imposes restrictions on processes with regard to reading the state of other processes. Vechev *et al.* [27] present an exponential algorithm for synthesizing synchronization mechanisms under limited observability, but they provide no results on the general case hardness of synthesizing synchronization mechanisms. Now, the open questions are: *What role do non-determinism and observability/modularization play in the complexity of incremental synthesis*? *Is incremental synthesis of safety easier for deterministic programs*?

In this paper, we prove that non-determinism is not the major source of the complexity of incremental synthesis; rather it is modularization constraints that complicate the incremental synthesis of safety properties. We consider Alpern and Schneider's [1] definition of safety/liveness properties, where a *property* is a set of sequences of states. Their definition of a *safety* property $\mathcal{P}$ can be represented as a set of finite sequences that cannot be extended to be in $\mathcal{P}$, which we call them *bad sequences*. We also investigate a special case of safety properties that can be specified as a set of *Bad Transitions* (BT) (introduced in [18]). The BT model is more general than the usual notion of *Bad States* (BS) in that every transition reaching a bad state is considered to be a bad transition, whereas not every bad transition reaches a bad state [20]. Previous work [7, 3] shows that incremental synthesis of safety (for deterministic and non-deterministic programs) can be done in polynomial time (in the size of the state space) under unlimited observability. Nonetheless, we show that under limited observability the general case complexity of synthesizing safety in deterministic programs increases to NP-complete! Our results imply that limited observability has a major impact on the complexity of incremental synthesis of safety (see Figure 1) regardless of non-determinism/determinism. While modularity is a powerful design concept, *design for change* [22] is also an important goal. To achieve this goal, research should be focused on identifying the kind of modularization techniques that facilitate incremental synthesis.

Organization. Section 2 presents preliminary concepts. Section 3 formulates the problem of incremental synthesis of safety. Section 4 shows that the general case complexity of incremental synthesis of safety in deterministic programs increases to NP-complete if we assume limited observability. Finally, in Section 5, we make concluding remarks and present a roadmap for future research.

	Unlimited Observability	**Limited Observability**
Deterministic Programs	P	**NP-complete***
Non-Deterministic Programs	P	NP-complete

Fig. 1. The impact of non-determinism and limited observability on the complexity of incremental synthesis (* depicts the contribution of this paper).

2 Preliminaries

In this section, we present formal definitions of finite-state programs, program components[1], computations and safety specifications. The definition of specification is adapted from Alpern and Schneider [1]. We use a read/write model from [2, 19].

Programs. A *program* $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ is a tuple of a finite set V_p of variables and a finite set $\mathcal{C}_p$ of computing components $C_1, \cdots, C_k$, where $k \geq 1$. Each variable $v_i \in V_p$, for $1 \leq i \leq N$, has a finite non-empty domain D_i. A *state* s of p is a valuation $\langle d_1, d_2, \cdots, d_N \rangle$ of program variables $\langle v_1, v_2, \cdots, v_N \rangle$, where $d_i \in D_i$. $\mathcal{I}_p$ denotes a finite set of initial states, and $\mathcal{F}_p$ represents a finite set of accepting/final states. For a variable v and a state s, $v(s)$ denotes the value of v in s. The *state space* $\mathcal{S}_p$ is the set of all possible states of p and $|\mathcal{S}_p|$ denotes the size of $\mathcal{S}_p$. A *state predicate* is a subset of $\mathcal{S}_p$. A *transition* is an ordered pair (s, s'), where s and s' are program states. A *component* C_j is a triple $\langle \delta_j, r_j, w_j \rangle$, where $1 \leq j \leq k$ and $\delta_j \subseteq \mathcal{S}_p \times \mathcal{S}_p$ denotes the *set of transitions* of C_j. We shall define r_j and w_j below. The set of transitions of a program p, denoted δ_p, is the union of the sets of transitions of its components; i.e., $\delta_p = \bigcup_{j=1}^{k} \delta_j$. A program p is *non-deterministic* iff (if and only if) the transition function δ_p is defined as $\mathcal{S}_p \rightarrow 2^{\mathcal{S}_p}$. A *deterministic* program is a special case where from each state there is *at most* one outgoing transition. That is, the set δ_p defines a *partial* (transition) function from $\mathcal{S}_p$ to $\mathcal{S}_p$. Thus, given a state $s \in \mathcal{S}_p$, δ_p returns *at most* one state $s' \in \mathcal{S}_p$. This is a property that each δ_j inherits from δ_p.

Notation. For simplicity, we shall misuse p and δ_p interchangeably.

Read/Write Model. In order to model the access rights of each component C_j $(1 \leq j \leq k)$ with respect to program variables, we define a set of variables that C_j is allowed to read, denoted r_j, and a set of variables that C_j can write, denoted w_j. Notice that, if a variable v of some component C_i is read/written through the *interface* of C_i, then v is considered readable/writable for the component that invokes the interface of C_i. We assume that $w_j \subseteq r_j$; i.e., a component cannot blindly write a variable it cannot read. The write restrictions of a component C_j identify a set of transitions $\{(s, s') \mid \exists v : v \notin w_j : v(s) \neq v(s')\}$ that δ_j excludes, where v denotes a variable. For example, consider a program p_r with two components C_1 and C_2 and two binary variables v_1 and v_2. The component C_1 (respectively, C_2) can read and write the variable v_1 (respectively, v_2), but it

[1] The term *component* can capture objects in object-oriented program, threads/processes in concurrent programming, nodes of network protocols, etc.

cannot read (or write) v_2 (respectively, v_1). Let $\langle v_1, v_2 \rangle$ denote the state of the program p_r. A transition t_1, represented as $(\langle 0,0 \rangle, \langle 1,1 \rangle)$, does not belong to C_1 because $v_2 \notin w_1$ and the value of v_2 is being updated. For similar reasons, t_1 does not belong to C_2 either.

The effect of read restrictions is that, during the synthesis of safety, δ_j of a component C_j includes (respectively, excludes) a transition t iff a *group* of transitions associated with t is included (respectively, excluded) in δ_j [19, 2]. In the aforementioned example, consider the transition t_2 as $(\langle 0,0 \rangle, \langle 1,0 \rangle)$. If C_1 includes only t_2, then the execution of t_2 can be interpreted as the atomic execution of the following if statement: 'if $(v_1 = 0) \wedge (v_2 = 0)$ then $v_1 := 1$'; i.e., C_1 needs to read v_2. Including both transitions $(\langle 0,0 \rangle, \langle 1,0 \rangle)$ and $(\langle 0,1 \rangle, \langle 1,1 \rangle)$ makes the value of v_2 irrelevant, thereby eliminating the need for reading v_2 by C_1. Thus, the component C_1 must either include or exclude both transitions as a group. Formally, a component C_j can include a transition (s, s') if and only if C_j also includes any transition (s_g, s'_g) such that for all variables $v \in r_j$, we have $v(s) = v(s_g)$ and $v(s') = v(s'_g)$, and for all variables $u \notin r_j$, we have $u(s) = u(s')$ and $u(s_g) = u(s'_g)$.

Computations. A *computation* of a program $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ is a sequence of states $\sigma = \ll s_0, s_1, \cdots \gg$, where each transition (s_i, s_{i+1}) in σ $(i \geq 0)$ belongs to some component C_j, $1 \leq j \leq k$, i.e., C_j *executes* the transition (s_i, s_{i+1}), and σ is *maximal*. That is, either σ is infinite, or if σ is finite and terminates in a state s_f, then there is no component of p that executes a transition (s_f, s) for any state s. A *computation prefix* of p is a finite sequence $\sigma = \ll s_0, s_1, \cdots, s_m \gg$ of states in which every transition (s_i, s_{i+1}), for $0 \leq i < m$, is executed by some component C_j, $1 \leq j \leq k$.

Properties and Specifications. Intuitively, a safety property/requirement states that nothing bad ever happens. Formally, we follow Alpern and Schneider [1] in defining a property as a set of sequences of states. A *safety* property $\mathcal{P}$ can be represented by a set of *finite* sequences of states, denoted $\mathcal{B}$, that cannot be extended to be in $\mathcal{P}$. Each sequence in $\mathcal{B}$ represents a scenario of the occurrence of something bad. For example, a safety property of a program with an integer variable x could stipulate that an increment of x must not *immediately* be followed by a decrement in the value of x. Such a safety property can be represented by a set of bad sequences of three states (i.e., two immediate transitions; one that increments x and the subsequent one that decrements x). In this paper, a liveness property states that something good eventually happens, including good things that occur infinitely often. Formally, a *liveness property*, denoted $\mathcal{L}$, is a set of sequences of states, where each sequence in $\mathcal{L}$ either terminates in a state belonging to a state predicate $\mathcal{F}$ (representing the good thing that should happen), or infinitely often visits some states in $\mathcal{F}$. This notion of liveness is sufficiently general to capture stutter-invariant Linear Temporal Logic (LTL) [8] properties [11, 23]. Following Alpern and Schneider [1], we define a specification, denoted *spec*, as a set of safety and liveness properties. A computation $\sigma = \ll s_0, s_1, \cdots \gg$ of a program $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ *satisfies a specification spec from* $\mathcal{I}_p$ iff (1) $s_0 \in \mathcal{I}_p$, (2) no sequence of the safety of *spec*, denoted $\mathcal{B}$, appears

in σ, and (3) if σ terminates in a state s_f, then $s_f \in \mathcal{F}_p$; otherwise, some states in $\mathcal{F}_p$ are reached infinitely often in σ. A *program p satisfies its specification spec from $\mathcal{I}_p$* iff all computations of p satisfy *spec* from any state in $\mathcal{I}_p$. Given a finite computation $\sigma = \ll s_0, s_1, \cdots, s_d \gg$, if no program component executes from s_d and $s_d \notin \mathcal{F}_p$, then σ is a *deadlocked computation* and s_d is a *deadlock state*. A computation $\sigma = \ll s_0, s_1, \cdots \gg$ of p is a *non-progress computation* iff σ does not infinitely often reach some state in $\mathcal{F}_p$ nor does it terminate in a state in $\mathcal{F}_p$. A deadlocked computation is an instance of a non-progress computation. Another example is the case where a computation of p includes a cycle in which no state belongs to $\mathcal{F}_p$, called a *non-progress cycle*. If a computation σ includes a sequence in $\mathcal{B}$, then σ is a *safety-violating computation*. A computation σ of p *violates spec* from a state s_0 iff σ starts at s_0 and σ is either a non-progress computation or a safety-violating computation. A program p *violates spec* from $\mathcal{I}_p$ iff there exists a computation of p that violates *spec* from some state s_0 in $\mathcal{I}_p$.

Notation. Whenever it is clear from the context, we abbreviate '*p satisfies spec from $\mathcal{I}_p$*' as '*p satisfies spec*'.

Bad Transitions (BT) model. Consider a special case in which safety specifications are modeled as a set of finite sequences of length 2; i.e., a sequence has only two states. That is, the safety specification rules out a set of transitions that must not appear in program computations, called *bad transitions* [18]. For instance, the safety specification of a program with an integer variable x may stipulate that x can be decremented only if its value is positive. Consider a state s_0, where $x(s_0) = -2$. State s_0 can be reached either by incrementing x from s_1, where $x(s_1) = -3$, or by decrementing x from s_2, where $x(s_2) = -1$. Observe that the transition (s_2, s_0) is a bad transition, whereas (s_1, s_0) is not. In this model, reaching s_0 does not necessarily violate safety and it depends on how s_0 is reached. Such a model of safety specification is a restricted version of the general model of safety specifications presented by Alpern and Schneider [1], but it is more general than the usual model of *bad states* often specified in the literature in terms of the *always* operator (or invariance properties) in temporal logic [8].

Remark. We investigate incremental synthesis under "no fairness".

3 Problem Statement

In this section, we formally define the problem of incremental synthesis of safety. Let $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ be a program that satisfies a specification *spec*. Moreover, let $\mathcal{B}$ denote the safety of *spec*, and $\mathcal{B}_{new}$ represent a new safety property (e.g., data race-freedom) that p does not satisfy. Our goal is to redesign p to a program $p_r = \langle V_p^r, \mathcal{C}_p^r, \mathcal{I}_p^r, \mathcal{F}_p^r \rangle$, such that p_r satisfies $\mathcal{B} \wedge \mathcal{B}_{new}$ and the liveness of *spec* from $\mathcal{I}_p^r$. For simplicity, during such redesign, we do not expand the state space of p; i.e., no new variables are added to V_p. Thus, we have $V_p^r = V_p$ and $\mathcal{S}_p^r = \mathcal{S}_p$. Since p_r must still satisfy *spec* from all states in $\mathcal{I}_p$, we should preserve all initial states of p. Thus, we require that $\mathcal{I}_p = \mathcal{I}_p^r$. We state the problem as follows:

Problem 1. **Incremental Synthesis of Safety**

- **Input**: A program $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ with its specification *spec*, its set of safety properties $\mathcal{B}$, a new safety property $\mathcal{B}_{new}$ and a set of read/write restrictions for all components in $\mathcal{C}_p$.
  - *Input Assumptions*: The program p satisfies *spec* from $\mathcal{I}_p$, but p may not satisfy $\mathcal{B}_{new}$ from $\mathcal{I}_p$.
- **Output**: A *redesigned program* $p_r = \langle V_p^r, \mathcal{C}_p^r, \mathcal{I}_p^r, \mathcal{F}_p^r \rangle$.
- **Constraints**:
  1. $V_p^r = V_p$ (i.e., $\mathcal{S}_p^r = \mathcal{S}_p$)
  2. $\mathcal{I}_p^r = \mathcal{I}_p$
  3. $\mathcal{F}_p^r \subseteq \mathcal{F}_p$ and $\mathcal{F}_p^r \neq \emptyset$
  4. The number of components and the read/write restrictions of each component in $\mathcal{C}_p$ remain the same in $\mathcal{C}_p^r$, but δ_j of each component C_j may change in $\mathcal{C}_p^r$
  5. $\delta_{p_r} \neq \emptyset$, and p_r satisfies $\mathcal{B} \wedge \mathcal{B}_{new}$ from $\mathcal{I}_p$
  6. Starting from any initial state $s_0 \in \mathcal{I}_p$, the revised program p_r satisfies its liveness specifications.

We now define the decision problem of incremental synthesis in the BT model.

Problem 2. **Decision Problem of Incremental Synthesis**

- **Instance**: A program $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ with its specification *spec*, its set of safety properties $\mathcal{B}$, a new safety property $\mathcal{B}_{new}$ and a set of read/write restrictions for all components in $\mathcal{C}_p$, where $\mathcal{B}$ and $\mathcal{B}_{new}$ are specified in the BT model of safety.
- **Question**: Does there exist a program $p_r = \langle V_p^r, \mathcal{C}_p^r, \mathcal{I}_p^r, \mathcal{F}_p^r \rangle$ that meets the constraints of Problem 1?

Significance of Problem 1. Several activities (e.g., debugging, porting, composition) during software development are instances of Problem 1. A few examples are as follows:

- **Debugging**: Consider the debugging of concurrent programs for data races between multiple threads. That is, multiple threads access shared data where at least one of them performs a write operation. Data race-freedom is a safety property whereas ensuring that each thread eventually gains access to the shared data (i.e., makes *progress*) is a liveness property. Eliminating data races while preserving the progress of each thread is a clear example of synthesizing safety [27].
- **Porting**: Consider a scenario in which a distributed application designed to be deployed on a traditional wireless network is considered for deployment on a wireless sensor network. The sensor nodes have a power-saving mode in which a node automatically turns off its radio (i.e., sleep mode) if no network activities are detected for a specific time interval. To port an existing wireless application to this new platform, one has to revise the application considering the sleep mode. That is, some of the activities would be forbidden in the sleep mode (e.g., sending messages). This is an additional safety constraint that should be met while preserving all other safety/liveness properties.

– **Composition and incremental development**: While in this paper we investigate Problem 1 in the same state space (i.e., no new variables are added while synthesizing safety), we will investigate the synthesis of safety under more relaxed conditions where the redesigned program may include new variables. Such a generalized formulation of Problem 1 captures software composition. Several approaches in software engineering (e.g. aspect-oriented programming [15]) rely on incremental design of software where modules/components that implement features/aspects are incrementally composed with an existing base system [15, 14, 26, 16]. Moreover, there are numerous applications where software behaviors should evolve by composing plug-in components (e.g., Mozilla extensions) or by integrating mobile code (e.g., Java applets, composable proxy filters [21]). While some researchers have investigated type safety of compositions [26], dynamic safety properties of compositions are also of paramount importance.

4 Hardness of Incremental Synthesis

In this section, we illustrate that the general case complexity of synthesizing safety in deterministic programs increases significantly if program components have limited observability with respect to the state of other components. The intuition behind our complexity result lies in the difficulty of redesigning program computations towards capturing a new safety property while preserving liveness. Consider a computation $\sigma = \ll s_0, \cdots, s_{i-1}, s_i, \cdots \gg$, where $i > 0$, in a program p and a new safety property $\mathcal{B}_{new}$ that forbids the execution of the transition $t = (s_{i-1}, s_i)$; i.e., t must not be executed in the redesigned program. As such, we have to remove t. If $s_{i-1} \notin \mathcal{F}_p$, then σ becomes a deadlocked computation. To resolve the deadlock state s_{i-1}, we systematically synthesize a new sequence of states $\sigma_r = \ll s'_0, \cdots, s'_k \gg$, for $k \geq 0$, that is inserted between s_{i-1} and s_i to satisfy $\mathcal{B}_{new}$ (i.e., avoid executing t). Note that while a direct transition from s_{i-1} to s_i violates $\mathcal{B}_{new}$, there may be another sequence of states that can be traversed from s_{i-1} to s_i without violating $\mathcal{B}_{new}$. The same argument holds about building a computation prefix between any predecessor of s_{i-1} and any successor of s_i; there is nothing particular about s_{i-1} and s_i.

Additionally, the synthesized sequence σ_r must not preclude the reachability of the accepting states in σ. To meet this requirement, no transition (s, s') in σ_r should be grouped with a transition (s_g, s'_g) such that s_g is reachable in some computation of p_r and the execution of the transition (s_g, s'_g) causes one of the following problems: (1) s'_g is a deadlock state, (2) (s_g, s'_g) is a safety-violating transition that must not be executed, and (3) s'_g is a predecessor state of s_g in σ, thereby creating a non-progress cycle[2]. To ensure that the above cases do not occur, we should examine (i) all transitions selected to be in σ_r, (ii) all transitions used to connect σ_r to s_{i-1} and s_i along with their associated transition groups, and (iii) the transitions of other computations. Intuitively, this leads to exploring an exponential number of possible combinations of the

[2] If s'_g is a successor of s_g in σ or is reachable in a different computation, then s_g should not have any other outgoing transition due to the determinism constraints.

safe transitions (and their transition groups) that can potentially be selected to be in σ_r. With this intuition, we prove that synthesizing safety in deterministic programs in the BT model of safety under limited observability is NP-complete (using a reduction from the 3-SAT problem [9]).

Problem 3. **The 3-SAT Decision Problem**

- **Instance**: A set of propositional variables, $x_1, x_2, ..., x_n$, and a Boolean formula $\Phi = C_1 \wedge C_2 \wedge ... \wedge C_m$, where $m, n > 1$ and each clause C_j $(1 \leq j \leq m)$ is a disjunction of exactly three literals. Wlog, we assume that the literals x_i and $\neg x_i$ do not simultaneously appear in the same clause $(1 \leq i \leq n)$.

- **Question**: Does there exist an assignment of truth values to $x_1, x_2, ..., x_n$ such that Φ is satisfiable?

In Subsection 4.1, we present a polynomial mapping from an arbitrary instance of 3-SAT to an instance of Problem 2. In Subsection 4.2, we illustrate that the instance of 3-SAT is satisfiable iff safety can be synthesized in the instance of Problem 2.

4.1 Polynomial Mapping from 3-SAT

In this section, we illustrate how for each instance of 3-SAT, we create an instance of Problem 2, which includes a program $p = \langle V_p, \mathcal{C}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$, and its safety specification $\mathcal{B}$ and the new safety property $\mathcal{B}_{new}$ that should be captured by the redesigned program. We build the instance of Problem 2 by considering states and transitions corresponding to each propositional variable and each clause in the instance of 3-SAT.

States and transitions. Corresponding to each propositional variable x_i, where $1 \leq i \leq n$, we consider the states $a_i, a'_i, b_i, b'_i, c_i, c'_i$ and d_i (as illustrated in Figure 2). We also include an additional state a_{n+1}. For each variable x_i, the instance of Problem 2 includes the transitions $(a_i, d_i), (a_i, b_i), (b_i, a_i), (a_i, b'_i), (b_i, c_i), (b_i, c'_i), (b'_i, c'_i), (c'_i, b'_i), (c_i, a'_i), (c'_i, a'_i)$ and (d_i, a'_i).

Corresponding to each clause $C_r = x_i \vee \neg x_j \vee x_k$, where $1 \leq r \leq m$ and $1 \leq i, j, k \leq n$, program p includes 8 states $z_r, z_{ri}, z'_{ri}, z_{rj}, z'_{rj}, z_{rk}, z'_{rk}$ and z'_r, and 7 transitions $(z_r, z_{ri}), (z_{ri}, z'_{ri}), (z'_{ri}, z_{rj}), (z_{rj}, z'_{rj}), (z'_{rj}, z_{rk}), (z_{rk}, z'_{rk})$ and (z'_{rk}, z'_r) depicted in Figure 3. Notice that the transitions in Figures 2 and 3 belong to different program components, which we shall explain later.

Input program. The input program p includes the transitions (a_i, d_i) and (d_i, a'_i), for $1 \leq i \leq n$ and the transitions (a_n, a_{n+1}) and (a_{n+1}, a_1) (see Figure 4). Starting from a_1, the input program p executes transitions $(a_i, d_i), (d_i, a'_i)$ and (a'_i, a_{i+1}), where $1 \leq i \leq n$. From a_{n+1}, the program returns to a_1.

Initial and final states. The states a_1 and z_r $(1 \leq r \leq m)$ are initial states, and a_{n+1} is an accepting/final state. Moreover, the states z_{ri}, z_{rj}, z_{rk} and z'_r are accepting states, where $1 \leq r \leq m$ and $1 \leq i, j, k \leq n$. Starting from a_1, the final state a_{n+1} is infinitely often reached. Further, if the program starts at z_r then it will halt in the accepting state z'_r. Since all transitions $(a_i, d_i), (d_i, a'_i), (a'_i, a_{i+1})$ and (a_{n+1}, a_1) satisfy $\mathcal{B}$, the program p satisfies its safety and liveness specifications from a_1. In summary, we have

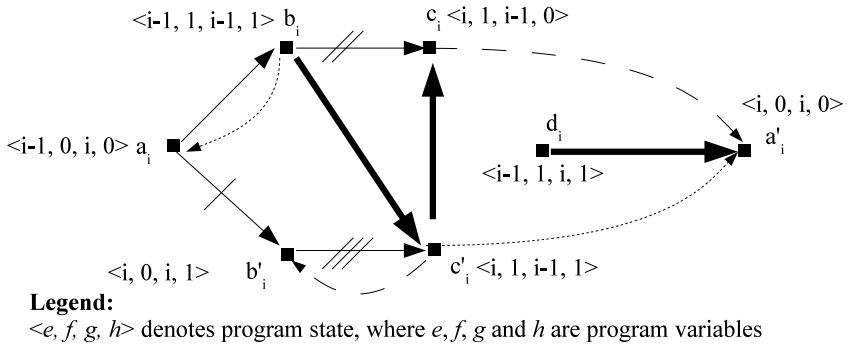

Fig. 2. The set of states and safe transitions corresponding to each propositional variable x_i in the instance of 3-SAT. Each state is annotated with the values assigned to program variables in that state.

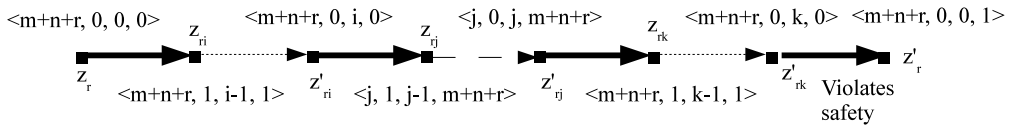

Fig. 3. States and transitions considered in the instance of Problem 2 corresponding to each clause $C_r = x_i \vee \neg x_j \vee x_k$ along with the values of variables.

- $\mathcal{I}_p = \{a_1\} \cup \{z_r | 1 \leq r \leq m\}$
- $\mathcal{F}_p = \{a_{n+1}\} \cup \{z_{ri}, z_{rj}, z_{rk}, z'_r |$ for each clause $C_r = x_i \vee \neg x_j \vee x_k$ in Φ, where $(1 \leq r \leq m) \wedge (1 \leq i, j, k \leq n)\}$

Safety specifications. The safety specification $\mathcal{B}$ rules out any transition other than the transitions in Figures 2 and 3. Notice that while these transitions are permitted by the safety specification $\mathcal{B}$, the input program p does not necessarily include all of them. The new safety property $\mathcal{B}_{new}$ rules out the transitions (a_i, d_i) and (z'_{rk}, z'_r), where $1 \leq i, k \leq n$ and $1 \leq r \leq m$. Thus, in the revised version of the instance of Problem 2, denoted p_r, these transitions must not be executed.

Program variables. The instance of Problem 2, denoted p, has four variables e, f, g and h. We denote a program state by $\langle e, f, g, h \rangle$. Figure 2 illustrates the values of variables in the states included corresponding to each variable x_i. Figure 3 presents the values of variables in the states included corresponding to each clause $C_r = x_i \vee \neg x_j \vee x_k$. As such, the domains of the variables are as follows:

- The variable e has the domain $\{0, \cdots, n\} \cup \{m + n + 1, \cdots, 2m + n\}$.

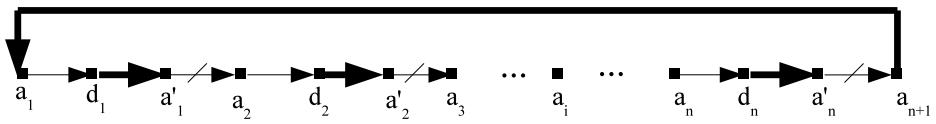

Fig. 4. States and transitions considered in the instance of Problem 2 corresponding to each propositional variable x_i.

- The domain of variable f is equal to $\{0, 1\}$.
- The variable g has a domain of $\{0, \cdots, n+1\}$.
- The domain of the variable h is $\{0, 1\} \cup \{m+n+1, \cdots, 2m+n\}$.

Program components. The program p includes seven components C_1-C_7 whose transitions have been depicted in Figure 5. The read and write restrictions of each component are as follows:

- The first component C_1 includes the transitions (a_i, d_i) and (a_i, b_i), for all $1 \leq i \leq n$ (see Figures 2 and 5). The set of readable variables of C_1, denoted r_1, is equal to $\{e, f, g, h\}$ and its set of writable variables is $w_1 = \{f, g, h\}$.
- The set of transitions (a_i, b'_i) and (a'_i, a_{i+1}) comprises the component C_2 (see the arrow with a crossed line on it in Figures 2 and 5). We have $r_2 = \{e, f, g, h\}$ and $w_2 = \{e, g, h\}$.
- The component C_3 includes the transitions (b_i, c_i) for $1 \leq i \leq n$ (see the arrow with two parallel lines on it in Figures 2 and 5). We have $r_3 = \{e, f, g, h\}$ and $w_3 = \{e, h\}$.
- The fourth component, denoted C_4, includes transitions (b'_i, c'_i) for $1 \leq i \leq n$, $r_4 = \{e, f, g, h\}$ and $w_4 = \{f, g\}$ (see the arrow with three parallel lines on it in Figures 2 and 5).
- For component C_5, we have $r_5 = w_5 = \{e, f, g\}$; i.e., C_5 cannot read h. The component C_5 includes transition (c_i, a'_i), which is grouped with (c'_i, b'_i) and (z_{qi}, z'_{qi}), due to inability of reading h, where (z_{qi}, z'_{qi}) corresponds to a clause C_q in which the literal $\neg x_i$ appears (see the dashed arrow (z_{rj}, z'_{rj}) in Figure 3). Notice that in these three transitions, the values of the readable variables e, f and g are the same in the source states (and in the destination states) and the value of h does not change during these transitions because it is not readable for C_5.
- The sixth component C_6 can read/write $r_6 = w_6 = \{f, g, h\}$, but cannot read e. Its set of transitions includes $(c'_i, a'_i), (b_i, a_i)$ and (z_{ri}, z'_{ri}) (see Figures 2 and 3) that are grouped due to inability of reading e, where (z_{ri}, z'_{ri}) corresponds to a clause C_r in which the literal x_i appears.
- The component C_7 can read and write all variables and its set of transitions includes (d_i, a'_i), (a_{n+1}, a_1), (b_i, c'_i) and (c'_i, c_i) for $1 \leq i \leq n$. Moreover, for each clause $C_r = x_i \vee \neg x_j \vee x_k$, where $1 \leq r \leq m$ and $1 \leq i, j, k \leq n$, component C_7 includes the following transitions: $(z_r, z_{ri}), (z'_{ri}, z_{rj}), (z'_{rj}, z_{rk})$ and (z'_{rk}, z'_r) (see Figure 3).

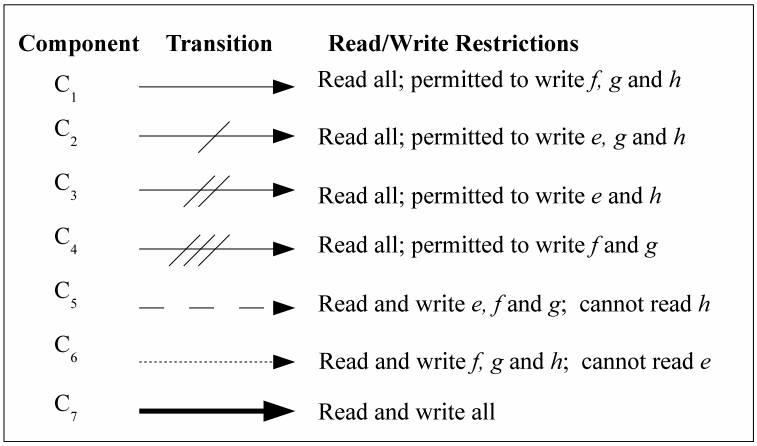

Fig. 5. Program components, their read/write restrictions and the annotation of their transitions.

Theorem 1. *The complexity of the mapping is polynomial. (Proof is straightforward; hence omitted.)*

4.2 Correctness of Reduction

In this section, we show that the instance of 3-SAT is satisfiable iff the instance of Problem 2 (created by the mapping in Section 4.1) can be redesigned to meet the safety properties $\mathcal{B}$ and $\mathcal{B}_{new}$ while preserving its liveness.

Lemma 1. *If the instance of 3-SAT is satisfiable, then the instance of Problem 2, denoted p, can be redesigned to another program p_r for the safety property $\mathcal{B}_{new}$ such that p_r meets the requirements of Problem 1.*

Proof. If the 3-SAT instance is satisfiable, then there must exist a value assignment to the propositional variables $x_1, \cdots, x_n$ such that all clauses C_r, for $1 \leq r \leq m$, evaluate to true. Corresponding to the value assignment to a variable x_i, for $1 \leq i \leq n$, we include a set of transitions in the redesigned program as follows:

- If x_i is assigned *true*, then we include transitions $(a_i, b_i), (b_i, c_i), (c_i, a'_i)$. Thus the computation prefix $\ll a_i, b_i, c_i, a'_i, a_{i+1} \gg$ is synthesized between a_i and a_{i+1}. Since we have included the transition (c_i, a'_i), and transition (c_i, a'_i) is grouped with (z_{qi}, z'_{qi}), where $1 \leq q \leq m$ for any clause C_q in which $\neg x_i$ appears, we must include (z_{qi}, z'_{qi}) as well (see the dashed arrow (z_{rj}, z'_{rj}) in Figure 3).
- If x_i is assigned *false*, then we include transitions $(a_i, b'_i), (b'_i, c'_i), (c'_i, a'_i)$, thereby synthesizing the computation prefix $\ll a_i, b'_i, c'_i, a'_i, a_{i+1} \gg$ between a_i and a_{i+1}. Due to the inability of reading e, including the transition (c'_i, a'_i)

results in the inclusion of the transitions (z_{li}, z'_{li}), where $1 \leq l \leq m$, for any clause C_l in which x_i appears (see the dotted arrows (z_{ri}, z'_{ri}) and (z_{rk}, z'_{rk}) in Figure 3).

– For each clause $C_r = x_i \vee \neg x_j \vee x_k$, the transition (z_{ri}, z'_{ri}) (respectively, (z_{rk}, z'_{rk})) is included iff x_i (respectively, x_k) is assigned *false*. The transition (z_{rj}, z'_{rj}) is included iff x_j is assigned *true*.

Figure 6 depicts a partial structure of a redesigned program for the value assignment $x_1 = false$, $x_2 = true$ and $x_3 = true$ in an example clause $C_5 = x_1 \vee \neg x_2 \vee x_3$. Note that the bad transition (z'_{53}, z'_5) is not reached because $x_3 = true$ and the transition (z_{53}, z'_{53}) is excluded.

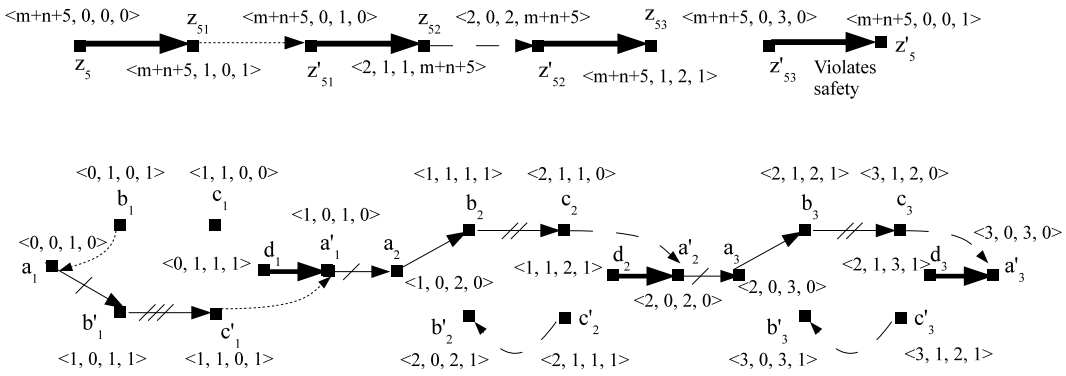

Fig. 6. A partial structure of the redesigned program corresponding to the value assignment $x_1 = false$, $x_2 = true$ and $x_3 = true$ for an example clause $C_5 = x_1 \vee \neg x_2 \vee x_3$.

Now, we illustrate that the redesigned program in fact meets the requirements of Problem 1. The state space remains obviously the same as no new variables have been introduced; i.e., $\mathcal{V}_p = \mathcal{V}_p^r$. During the selection of transitions based on value assignment to propositional variables, we do not remove any initial states. Thus, we have $\mathcal{I}_p = \mathcal{I}_p^r$.

Satisfying safety properties. Since the new safety property rules out transitions (a_i, d_i) and (z'_{rk}, z'_r), we have to ensure that the redesigned program does not execute them. From a_i, the program either transitions to b_i or to b'_i. Thus, safety is not violated from a_i. Moreover, since all clauses are satisfied, at least one literal in each clause $C_r = x_i \vee \neg x_j \vee x_k$ must be true. Thus, at least one of the three transitions $(z_{ri}, z'_{ri}), (z_{rj}, z'_{rj})$ or (z_{rk}, z'_{rk}) is excluded, thereby preventing the reachability of z'_{rk}; i.e., the safety-violating transition (z'_{rk}, z'_r) will not be executed.

Reachability of accepting states (satisfying liveness specifications). While the accepting state z'_r is no longer reachable, the redesigned program halts in one of the accepting state z_{ri}, z_{rj} or z_{rk}. Moreover, the accepting state a_{n+1} is reached infinitely often due to the way we have synthesized the sequences of

states $\ll a_i, b_i, c_i, a'_i, a_{i+1} \gg$ or $\ll a_i, b'_i, c'_i, a'_i, a_{i+1} \gg$ between a_i and a_{i+1}. That is, (non-)terminating computations remains (non-)terminating. Thus, starting from any initial state, some accepting states will be visited infinitely often; i.e., $\mathcal{F}_p^r \subseteq \mathcal{F}_p$. Therefore, if the instance of 3-SAT is satisfiable, then there exists a redesigned program for the instance of Problem 2 that satisfies the requirements of Problem 1.

Lemma 2. *If there exists a redesigned version of the instance of Problem 2 that meets the requirements of Problem 1, then the instance of 3-SAT is satisfiable.*

Proof. Let p_r be a redesigned version of the instance of Problem 2 that meets all the requirements of Problem 1. As such, the set of initial states $\mathcal{I}_p^r$ must be equal to the set $\{a_1\} \cup \{z_r | 1 \leq r \leq m\}$. Starting from a_1, p_r must execute a safe transition. Otherwise, we reach a contradiction; i.e., either a_1 is a deadlock state or the transition (a_1, d_1), which violates the new safety specification is executed. Thus, p_r either includes (a_1, b_1) or (a_1, b'_1), but not both (because p_r is a deterministic program). If p_r includes (a_1, b_1), then we set x_1 to *true* in the 3-SAT formula. If p_r includes (a_1, b'_1), then we set x_1 to *false*.

We assign truth values to each x_i, for $1 \leq i \leq n$, depending on the presence of (a_i, b_i) or (a_i, b'_i) at state a_i (similar to the way we assign a value to x_1). Such a value assignment strategy results in a unique truth-value assigned to each variable x_i. If p_r includes (a_i, b_i), then, from b_i, p_r includes either (b_i, c_i) or (b_i, c'_i) (see Figure 2), but not both (because of determinism). If p_r includes (b_i, c'_i), then, from c'_i, p_r must include either (c'_i, c_i) or (c'_i, a'_i). If p_r includes (c'_i, a'_i), then it must include (b_i, a_i) since these two transitions are grouped due to inability of C_6 in reading e. As such, the two transitions (a_i, b_i) and (b_i, a_i) make a non-progress cycle in p_r (see Figure 2), which is unacceptable as it violates liveness. Now, we show that, from c'_i, p_r cannot include (c'_i, c_i) either. If p_r includes (c'_i, c_i), then it must include (c_i, a'_i), which is grouped with (c'_i, b'_i) due to inability of C_5 in reading h (see Figure 2). Thus, p_r may reach b'_i from c'_i and deadlock in b'_i. Thus, if p_r includes (a_i, b_i) from a_i, then it must include (b_i, c_i) and (c_i, a'_i). In case where p_r includes (a_i, b'_i) from a_i, the transition (b'_i, c'_i) must also be included; otherwise p_r deadlocks in b'_i (Figure 2). From c'_i, p_r cannot include (c'_i, c_i) because it has to include (c_i, a'_i) that is grouped with (c'_i, b'_i), which creates a non-progress cycle. Thus, p_r must include (c'_i, a'_i) from c'_i.

We also illustrate that each clause in the 3-SAT formula evaluates to *true*. Consider a clause $C_r = x_i \vee \neg x_j \vee x_k$. Starting from the initial state z_r, the transition (z_r, z_{ri}) must be present in p_r; otherwise z_r is a deadlock state. Moreover, from z_r, the safety-violating transition (z'_{rk}, z'_r) must not be executed. Thus, at least one of the transitions $(z_{ri}, z'_{ri}), (z'_{ri}, z_{rj}), (z_{rj}, z'_{rj}), (z'_{rj}, z_{rk})$ or (z_{rk}, z'_{rk}) (see Figure 3) must be excluded in p_r. However, if one of the transitions (z_r, z_{ri}), $(z'_{ri}, z_{rj}), (z'_{rj}, z_{rk})$ or (z'_{rk}, z'_r) is excluded, then a reachable deadlock state could be created as their source states are not accepting states. Thus, if either z'_{ri} or z'_{rj} is reached from z_r, then the corresponding transition (z'_{ri}, z_{rj}) or (z'_{rj}, z_{rk}) must be present in p_r. Hence, at least one of the transitions (z_{ri}, z'_{ri}), (z_{rj}, z'_{rj}) or (z_{rk}, z'_{rk}) must be excluded in p_r; i.e., at least one literal in C_r must be *true*, thereby satisfying C_r.

Theorem 2. *Synthesizing safety (under limited observability) in deterministic programs in the BT model of safety specifications is NP-hard in $|\mathcal{S}_p|$. (Proof follows from Lemmas 1 and 2.)*

Theorem 3. *Synthesizing safety (under limited observability) in deterministic programs in the BT model of safety specifications is NP-complete (in $|\mathcal{S}_p|$).*

Proof. The proof of NP-hardness follows from Theorem 2. The proof of membership in NP is straightforward; given a revised program one can verify the constraints of Problem 1 (in the BT model) in polynomial time.

Theorem 4. *Synthesizing safety (under limited observability) in deterministic programs in the Bad State (BS) model of safety specifications is also NP-complete (in $|\mathcal{S}_p|$).*

Proof. The proof of NP-hardness works for the case where the safety specification rules out the reachability of states d_i and z'_r in the instance of Problem 2. The proof of NP membership is straightforward.

5 Conclusions and Future Work

This paper investigates the problem of capturing new safety requirements/properties while preserving existing safety and liveness properties, called *the problem of incremental synthesis*. Previous work [7, 3] shows that incremental synthesis for non-deterministic programs can be done in polynomial time (in the size of program state space) under unlimited observability. Moreover, it is known [4] that the complexity of incremental synthesis of safety for non-deterministic programs would increase to NP-complete under limited observability. In this paper, we illustrated that even for deterministic programs the complexity of incremental synthesis of safety is NP-complete (in program state space). Our NP-hardness proof illustrates that the read inabilities of each component with respect to the local state of other components is a major cause of complexity. Such read inabilities are mainly created because of encapsulation/modularization of functionalities at early stages of design. On one hand, encapsulation/modularity enables designers to create good abstractions while capturing different functionalities. On the other hand, encapsulation exacerbates the complexity of behavioral changes [22] when new crosscutting requirements have to be realized across the components of an existing program. To facilitate change while reaping the benefits of modularization in design, we will extend the work presented in this paper in the following directions:

– *Sound polynomial-time heuristics.* We will concentrate on devising polynomial-time *heuristics* that reduce the complexity of synthesizing safety at the expense of completeness. That is, if heuristics succeed in generating a redesigned program, then the generated program will capture the new safety property while preserving liveness. However, such heuristics may fail to generate a redesigned program while one exists.

- *Sufficient conditions.* We will identify conditions under which safety can be synthesized in polynomial time. Specifically, we would like to address the following questions: (i) *What kinds of inter-component topologies (i.e., read/write restrictions) should a program have such that a new safety requirement can be captured in it efficiently?* (ii) *For which types of programs and safety specifications the complexity of synthesizing safety is polynomial?*
- *Backtracking.* We will implement a backtracking algorithm for synthesizing safety under limited observability. While we showed that it is unlikely that safety can efficiently be synthesized under limited observability, in many practical contexts the worst case exponential complexity may not be experienced. Thus, we expect that a backtracking algorithm can explore the entire state space in a reasonable amount of time. Moreover, we will implement a parallel version of the backtracking algorithm that will benefit from randomization for search diversification.
- *An extensible software framework.* We will develop a framework that provides automated assistance in synthesizing safety. Such a framework will include a repository of *reusable* heuristics that facilitate the synthesis of safety in an automated fashion. Two categories of users can benefit from our extensible framework, namely, (1) *developers of heuristics* who will focus on designing new heuristics and integrating them into our framework, and (2) *mainstream programmers* who will use the built-in heuristics to capture new safety properties in programs.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2), March 2001. An extended abstract appeared at the ACM Symposium on Principles of Distributed Computing - 1996.
3. B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):1–28, 2009.
4. B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems (OPODIS)*, pages 408–427, 2008.
5. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *POPL'00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–66, 2000.
6. S. D. Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 135–145, 2008.
7. A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.
8. E. A. Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16: Temporal and Modal Logics, pages 995–1067. Elsevier Science Publishers B. V., 1990.

9. M. Garey and D. Johnson. *Computers and Interactability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.
10. M. Goldman and S. Katz. Maven: Modular aspect verification. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 308–322, 2007.
11. H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. *Electronic Notes in Theoretical Computer Science*, 66(2):178–193, 2002.
12. M. V. Iordache, J. O. Moody, and P. J. Antsaklis. Synthesis of deadlock prevention supervisors using Petri Nets. *IEEE Transactions on Robotics and Automation*, 18(1):59–68, 2002.
13. R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing behavioral constraints in evolving aspect-oriented programs. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages (FOAL)*, pages 19–28, 2008.
14. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
15. G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.
16. S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):7, 2007.
17. S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. *ACM SIGSOFT Software Engineering Notes*, 29(6):137–146, 2004.
18. S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, OH, USA, 1999.
19. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.
20. S. S. Kulkarni and A. Ebnenasir. The effect of the safety specification model on the complexity of adding masking fault-tolerance. *IEEE Transaction on Dependable and Secure Computing*, 2(4):348–355, 2005.
21. P. K. McKinley, U. I. Padmanabhan, N. Ancha, and S. M. Sadjadi. Composable proxy services to support collaboration on the mobile internet. *IEEE Transactions on Computers*, 52(6):713–726, 2003.
22. D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, 1979.
23. A.-E. B. Salem, A. Duret-Lutz, and F. Kordon. Model checking using generalized testing automata. *Transactions on Petri Nets and Other Models of Concurrency*, 6:94–122, 2012.
24. D. Smith. Requirement enforcement by transformation automata. In *Sixth Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 5–15, 2007.
25. R. S. Sreenivas. On the existence of supervisory policies that enforce liveness in discrete-event dynamic systems modeled by controlled Petri Nets. *IEEE Transactions on Automatic Control*, 42(7):928–945, 1997.
26. S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In *6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104, 2007.
27. M. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 139–154. Springer, 2009.