# Computer Science Technical Report

## How Hard Is Aspect-Oriented Programming?

Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-08-04
December 2008


Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

# How Hard Is Aspect-Oriented Programming?

Ali Ebnenasir

December 2008

**Abstract**

In aspect-oriented programming, it is desirable to algorithmically design *correct* advised programs that meet a new crosscutting concern while preserving the properties of the base program. Such an automated design eliminates the need for after-the-fact verification, thereby potentially reducing development costs. In this paper, we present a formalization of the problem of designing safety aspects for finite-state programs while ensuring the reachability of states from where liveness is met, where safety stipulates that nothing bad ever happens, and liveness states that something good will eventually happen in program computations. Subsequently, we illustrate that, for deterministic sequential programs, it is impossible to efficiently design safety aspects and reach states from where liveness is satisfied unless P = NP; i.e., the problem is NP-hard in program state space. This is a counterintuitive result as previous work illustrates that this problem is NP-hard for *distributed* programs in the presence of faults.

# Contents

# 1 Introduction

We concentrate on the problem of designing safety aspects for deterministic sequential (finite-state) programs while ensuring the reachability of states from where liveness is met. The composition of a crosscutting aspect with an existing program, called the *base* program, that does not meet that aspect is realized in terms of composing new pieces of code, called *advices*, in specific places in the code of the base program (e.g., method calls), called the *joinpoints* [1–4]. Given a base program and a crosscutting concern specified as a safety property (e.g., invariance properties [5,6]), developers often manually identify joinpoints and the advices that should be woven with the base program (in those joinpoints) so that the safety property is met by the resulting program, called the *advised program*. Moreover, the advised program should guarantee that it eventually reaches states from where the liveness specification of the base program is satisfied. To verify that the advised program is *correct*, i.e., it meets the crosscutting concern while preserving the correctness of the base program with respect to its specification, researchers have proposed numerous verification approaches [4,7–9]. However, when such verification fails, programmers have to manually fix the advised program using the feedback of the verification process (e.g., counterexamples in model checking). To facilitate the design of aspect-oriented programs, it is desirable to automatically identify the join points and the advices that should be woven. Several researchers have proposed formalisms and approaches for such an automated design of aspects [2,6,10–12]. We focus our attention on investigating the computational complexity of the automated design of safety aspects.

In this paper, we first formulate the problem of Automated Design of Safety Aspects (ADSA) in the context of deterministic sequential programs. Then we illustrate that the ADSA problem is in fact NP-hard in program state space. The significance of our hardness result is multi-fold. First, we illustrate that, in general, it is unlikely that the design of *correct* aspect-oriented programs for safety concerns can be done efficiently (unless P = NP). Second, our hardness result establishes a lower bound complexity for the design of aspect-oriented programs for other types of crosscutting concerns. For instance, we conjecture that, in general, the design of crosscutting *liveness* concerns (e.g., guaranteed service) cannot be easier than designing safety aspects. Third, our NP-hardness proof will help identify sufficient conditions under which the automated design of aspects can be done in polynomial time (e.g., for a special class of programs and aspects). Fourth, our result is somewhat counterintuitive as in our previous work [13], we have shown that designing safety concerns for distributed programs in the presence of faults is a hard problem, whereas in this paper, we show that designing safety aspects remains difficult even for deterministic programs in the absence of faults!

**Organization.** The organization of the paper is as follows: We present preliminary concepts in Section 2. In Section 3, we formulate the problem of automated design of safety aspects. Subsequently, in Section 4, we show that designing safety aspects is NP-complete by a reduction from the 3-SAT problem [14]. Finally, we make concluding remarks and discuss future work in Section 5.

# 2 Preliminaries

In this section, we present formal definitions of programs, objects, computations and safety specifications. The definition of specification is adapted from Alpern and Schneider [15]. We represent our read/write model from [16, 17]. We define advices, joinpoints, pointcuts and weaving techniques in the level of states and transitions. Several formalizations of the corresponding concepts in the aspect-oriented programming literature [1–4] have inspired our formal definitions.

**Programs and objects.** A *program* $P = \langle V_p, \mathcal{O}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ is a tuple of a finite set $V_p$ of variables, a finite set $\mathcal{O}_p$ of objects, a finite set of initial states $\mathcal{I}_p$ and a finite set of accepting states. 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 a program $p$ 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 $d_i \in D_i$. The *state space* $\mathcal{S}_p$ is the set of all possible states of $p$. A *transition* of $p$ is of the form $(s, s')$, where $s$ and $s'$ are program states. An *object* $O_j$ , where $1 \leq j \leq k$ and $k \geq 1$, is defined in terms of a set of transitions captured by a partial transition function $\delta_j : \mathcal{S}_p \rightarrow \mathcal{S}_p$ that takes a program state and determines what the next state is. From any state, at most one object executes. The set of transitions of a program is the union of the sets of transitions of its objects.

**Computations.** A *computation* of a program $P = \langle V_p, \mathcal{O}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ is a sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of

states with length $len(\sigma)$ that satisfies the following conditions: (1) for each transition $(s_i, s_{i+1})$, where $0 \leq i < len(\sigma)$, in $\sigma$, there exists an object $O_j$, $1 \leq j \leq k$, that includes $(s_i, s_{i+1})$; i.e., $O_j$ *executes* the transition $(s_i, s_{i+1})$; (2) if $\sigma$ is finite and terminates in a state $s_f$, then there does not exist a state $s$ such that $(s_f, s)$ is executed by some program object. A *computation prefix* of $p$ is a finite sequence $\sigma = \langle s_0, s_1, \cdots, s_m \rangle$ of states in which every transition $(s_i, s_{i+1})$, for $0 \leq i < m$, is executed by some object in $\mathcal{O}_j$, $1 \leq j \leq k$.

**Specifications.** Intuitively, a safety specification states that nothing bad ever happens. Formally, we define a safety specification in terms of a set of transitions [18], denoted $\mathcal{B} \subseteq \mathcal{S}_p \times \mathcal{S}_p$, that must not appear in program computations. That is, the set $\mathcal{B}$ denotes the *bad* things that must not occur.[1] A program computation $\sigma = \langle s_0, s_1, \cdots \rangle$ *satisfies its specification from* $\mathcal{I}_p$ iff (if and only if) (1) $s_0 \in \mathcal{I}_p$, (2) no transition $(s_i, s_{i+1})$, $i \geq 0$, is in $\mathcal{B}$ and (3) some states in $\mathcal{F}_p$ are reached infinitely often. If a computation $\sigma$ includes a transition in $\mathcal{B}$, then $\sigma$ is a *safety-violating computation*. The intuition behind the third requirement is that the set of states $\mathcal{F}_p$ defines a condition for satisfying liveness specifications that state something good must happen infinitely often. Thus reaching some states in $\mathcal{F}_p$ infinitely often (respectively, halting in a state in $\mathcal{F}_p$) would satisfy the liveness specification. A *program $P$ satisfies its specification from* $\mathcal{I}_p$ iff all computations of $p$ satisfy its specification from $\mathcal{I}_p$. Given a finite computation $\sigma = \langle s_0, s_1, \cdots, s_d \rangle$, if there does not exist a state $s$ such that $(s_d, s)$ is executed by some program object and $s_d \notin \mathcal{F}_p$, then $\sigma$ is a *deadlocked computation* and $s_d$ is a *deadlock state*. A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ that does not reach some state in $\mathcal{F}_p$ infinitely often is a *non-progress computation*. A deadlocked computation is an instance of a non-progress computation. Another example is the case where a computation includes a cycle in which no state belongs to $\mathcal{F}_p$, called a *non-progress cycle*. A computation $\sigma$ *violates the specification* (i.e., does not satisfy the specification) from a state $s_0$ iff $\sigma$ starts at $s_0$ and $\sigma$ is either a non-progress computation or a safety-violating computation. A program $P$ *violates the specification* (i.e., does not satisfy the specification) from $\mathcal{I}_p$ iff there exists a computation of $P$ that violates the specification from some state $s_0$ in $\mathcal{I}_p$.

**Advices.** An *advice* is a finite sequence of states $\langle as_0, \cdots, as_m \rangle$, where $m \geq 0$, $as$ denotes a state of an advice and every transition $(as_i, as_{i+1})$, $0 \leq i < m$, is executed by some object in $\mathcal{O}_p$.

**Advice composition (weaving).** There are three ways of composing an advice at a specific joinpoint, namely *Before, After* and *Around* [1–4]. For instance, if the joinpoint is a call to a method m(), then an advice can be executed *before* m(), *after* m() or *around* m(); i.e., bypassing the execution of m().[2] Next, we present the before, after and around compositions at the level of program computations.

Let $c = \langle s_0, \cdots, s_b, \cdots s_e, \cdots \rangle$ be a program computation, the state $s_b$ be a joinpoint and $a = \langle as_0, \cdots, as_m \rangle$ be an advice. (A *pointcut* is a set of states/joinpoints) In a concrete sense, $s_b$ could be the starting point of a method of some object and the advice $a$ is supposed to be executed before that method. In the Before weaving (see Figure 1), the transition $(s_{b-1}, s_b)$ is replaced by the sequence of transitions $s_{b-1}, as_0, \cdots, as_m, s_b$.

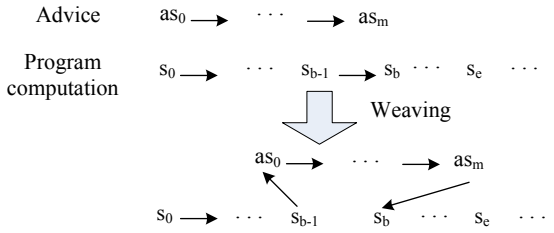


Figure 1: Before advice composition.

If the advice $a$ were to be woven After a method call ending in $s_e$, then the transition $(s_e, s_{e+1})$ would be replaced by $s_e, as_0, \cdots, as_m, s_{e+1}$ (see Figure 2).

Weaving advice $a$ Around the method call $s_b, \cdots, s_e$ generates the sequence $s_{b-1}, as_0, \cdots, as_m, s_{e+1}$ (see Figure 3).

[1] The notion of bad transitions is more general than the concept of bad states often specified in the literature in terms of the *always* operator (or invariance properties) in temporal logic [19]. That is, any transition reaching a bad state is considered a bad transition, however, a state having one outgoing (or incoming) bad transition may not necessarily be a bad state.

[2] There are more complicated ways for around composition (e.g., inserting copies of m() in the advice [4]), which we omit for simplicity.

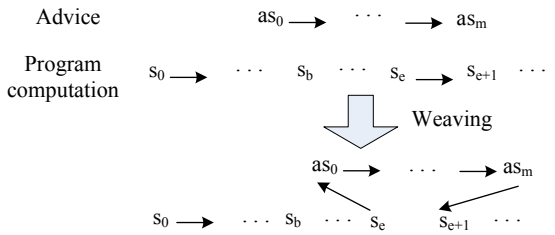

Figure 2: After advice composition.

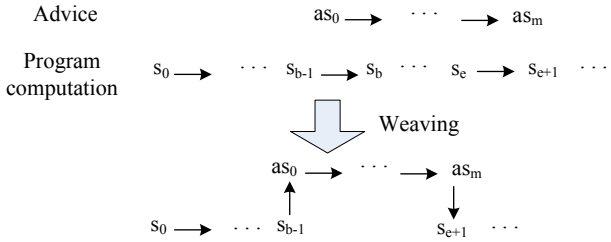

Figure 3: Around advice composition.

While the above weaving methods are different from the point of view of their order with respect to a method call $s_b, \cdots, s_e$, in principle, the advice $a$ is inserted between two states (be it between $s_{b-1}$ and $s_b$, between $s_e$ and $s_{e+1}$ or between $s_{b-1}$ and $s_{e+1}$). We use this fact to define the problem of automated design of advices for a given safety aspect in Section 3. Moreover, in Section 4, we shall construct our proof of NP-completeness based on deciding about the sequences of transitions that can be woven between two specific states so a safety aspect is captured.

**Read/write model.** In order to model the access rights of the objects with respect to program variables, for each object $O_j$ $(1 \leq j \leq k)$, we define a set of variables that $O_j$ is allowed to read, denoted $r_j$, and a set of variables that $O_j$ can write, denoted $w_j$. We assume that $w_j \subseteq r_j$; i.e., an object cannot blindly write a variable it cannot read. The write restrictions identify a set of transitions $\{(s, s') | \exists v : v \notin w_j : v(s) \neq v(s')\}$ that $O_j$ does not include, where $v$ denotes a variable and $v(s)$ denotes the value of $v$ in the state $s$. For example, consider a program $P_e$ with two objects $O_1$ and $O_2$ and two binary variables $v_1$ and $v_2$. The object $O_1$ (respectively, $O_2$) can read and write the variable $v_1$ (respectively, $v_2$), but it cannot read (and write) $v_2$ (respectively, $v_1$). Let $\langle v_1, v_2 \rangle$ denote the state of the program $P_e$. In this case, a transition $t_1$, represented as $\langle 0, 0 \rangle \rightarrow \langle 1, 1 \rangle$, does not belong to $O_1$ because $v_2 \notin w_1$ and the value of $v_2$ is being updated.

The effect of read restriction is that when an object includes (respectively, excludes) a transition, a *group* of transitions is included (respectively, excluded) [16, 20]. Consider the transition $t_2$ as $\langle 0, 0 \rangle \rightarrow \langle 1, 0 \rangle$. If $O_1$ 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., $O_1$ needs to read $v_2$. Including both transitions $\langle 0, 0 \rangle \rightarrow \langle 1, 0 \rangle$ and $\langle 0, 1 \rangle \rightarrow \langle 1, 1 \rangle$ makes the value of $v_2$ irrelevant, thereby eliminating the need for reading $v_2$ by $O_1$. Thus, the object $O_1$ must include both transitions as a group. Likewise, if we were to remove transition $t_2$ from the set of transitions of $O_1$, we would have to exclude the group associated with $t_2$ (due to read restrictions). Formally, an object $O_j$ can include a transition $(s, s')$ if and only if $O_j$ also includes the 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 $u(s_g) = u(s'_g)$. In Section 4, we shall use the notion of grouping of transitions (due to read inabilities) to illustrate that the problem of designing correct aspect-oriented programs is NP-complete.

# 3 Problem Statement

In this section, we formally define the problem of designing advices for safety aspects. Let $P = \langle V_p, \mathcal{O}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$ be the base program and $\mathcal{B}$ denote the safety specification of $P$. We assume that $P$ satisfies its specification before weaving the advices; i.e., $P$ executes no transitions in $\mathcal{B}$ and all computations of $P$ that start from a

state in $\mathcal{I}_p$ infinitely often reach a state in $\mathcal{F}_p$. Moreover, let $\mathcal{B}_{new}$ denote a new safety specification that $P$ does not satisfy. Our goal is to automatically design an advised version of $P$, denoted $P_a = \langle V_p^a, \mathcal{O}_p^a, \mathcal{I}_p^a, \mathcal{F}_p^a \rangle$, such that $P_a$ satisfies $\mathcal{B} \wedge \mathcal{B}_{new}$ and any computation of $P_a$ starting from a state in $\mathcal{I}_p$ infinitely often reaches some state in $\mathcal{F}_p$. Note that, for simplicity, during such transformation, we do not expand the state space of $P$; i.e., no new variables are added to $V_p$ and thus $\mathcal{S}_p^a = \mathcal{S}_p$.

In order to simplify the design of $P_a$, we benefit from the fact that $P$ satisfies its specification from $\mathcal{I}_p$. In particular, in the design of $P_a$, we focus on weaving new advices with existing computations of $P$. For this reason, we do not consider new initial states since we do not have any guarantees about the behaviors of $P$ from new initial states. Moreover, since $P_a$ must still satisfy the specification of $P$ from all states in $\mathcal{I}_p$, we should preserve all initial states of $P$. From a practical point of view, the initial states are valuable and we do not want to eliminate them during the automated design of the advised program. Thus, we require that $\mathcal{I}_p = \mathcal{I}_p^a$. Furthermore, starting from $\mathcal{I}_p$, we require the computations of $P_a$ to infinitely often reach a non-empty subset of the accepting states that $P$ would reach; i.e., $\mathcal{F}_p^a \subseteq \mathcal{F}_p$. Hence, we define the requirements of the problem of automated design of safety aspects as follows:

**The Problem of Automated Design of Safety Aspects (ADSA).**
Given is a program $P = \langle V_p, \mathcal{O}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$, its safety specification $\mathcal{B}$ and a new safety specification $\mathcal{B}_{new}$. The program $P$ satisfies its specification from $\mathcal{I}_p$; i.e., $P$ satisfies its safety specification $\mathcal{B}$ and $\mathcal{F}_p$ is reached infinitely often in all computations of $P$ from $\mathcal{I}_p$. However, $P$ may not satisfy $\mathcal{B}_{new}$ from $\mathcal{I}_p$.

Identify an *advised program* $P_a = \langle V_p^a, \mathcal{O}_p^a, \mathcal{I}_p^a, \mathcal{F}_p^a \rangle$ such that the following conditions hold:

1. $\mathcal{S}_p^a = \mathcal{S}_p$
2. $\mathcal{I}_p^a = \mathcal{I}_p$
3. $\mathcal{F}_p^a \subseteq \mathcal{F}_p$ and $\mathcal{F}_p^a \neq \emptyset$
4. $P_a$ satisfies $\mathcal{B} \wedge \mathcal{B}_{new}$ from $\mathcal{I}_p$
5. Starting from $\mathcal{I}_p$, computations of $P_a$ reach $\mathcal{F}_p^a$ infinitely often. That is, starting from $\mathcal{I}_p$, no computation of $P_a$ becomes a non-progress computation. □

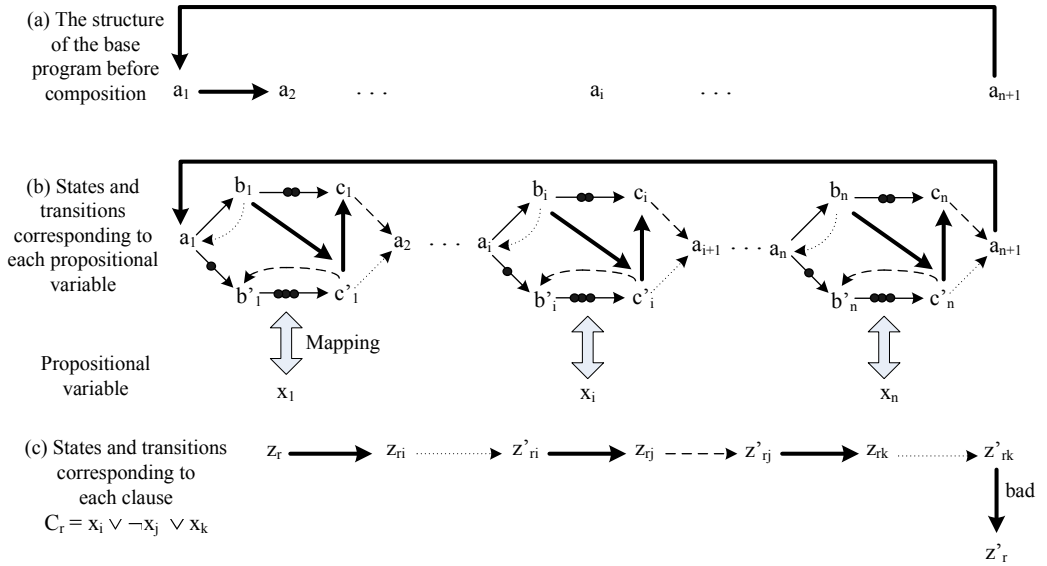


Figure 4: States and transitions corresponding to propositional variables and clauses.

# 4 Hardness Result

In this section, we illustrate that the complexity of automated design of safety aspects is NP-complete in program state space. The intuition behind our complexity result lies in the difficulty of building an acceptable advice that can be woven between two specific states in a computation. In particular, consider a computation $\sigma = \langle s_0, \cdots, s_i, s_{i+1}, \cdots \rangle$ in a program $P$ ($i > 0$) and a new safety specification $\mathcal{B}_{new}$ that forbids the execution of the transition $t = (s_i, s_{i+1})$; i.e., $t$ must not be executed in the advised program. As such, we have to remove $t$. If $s_i \notin \mathcal{F}_p$, then $\sigma$ becomes a deadlocked computation, which is not desirable. To resolve the deadlock state $s_i$, we systematically synthesize an advice $a = \langle sa_0, \cdots, sa_k \rangle$, for $k \geq 0$, that is woven between $s_i$ and $s_{i+1}$. Such an advice must satisfy $\mathcal{B} \wedge \mathcal{B}_{new}$; i.e., no transition in advice $a$ violates $\mathcal{B} \wedge \mathcal{B}_{new}$. Additionally, the advice $a$ must not preclude the reachability of the accepting states in $\sigma$. To meet this requirement, no transition $(s, s')$ in advice $a$ should be grouped with a transition $(s_g, s'_g)$ such that $s_g$ is reachable in some computation of $P$. If $s_g$ is reached in some program computation $\sigma$, then the execution of the transition $(s_g, s'_g)$ may cause three problems: (1) $s'_g$ may not have any outgoing transition, thus creating a reachable deadlock state in some program computation, (2) $s'_g$ may be an ancestor state of $s_g$ in $\sigma$, thereby creating a non-progress cycle, and (3) $(s_g, s'_g)$ may be a safety-violating transition that must not be executed. To ensure that the above cases do not occur, we should re-examine all transitions selected to be in advice $a$ and all transitions used to weave $a$ along with their associated groups. This may lead to verifying an exponential number of possible combinations of the safe transitions that can potentially be selected to be in an advice. Next, we prove that the ADSA problem is NP-complete using a reduction from the 3-SAT problem [14] to ADSA. First, we reiterate the decision problem of 3-SAT. We also state the ADSA problem as a decision problem. Then, in Subsection 4.1, we present a polynomial reduction from an arbitrary instance of 3-SAT to an instance of the ADSA decision problem. Finally, we illustrate that the instance of 3-SAT is satisfiable iff the answer to the ADSA decision problem is affirmative.

**The 3-SAT decision problem.**

Given is a set of propositional variables, $x_1, x_2, ..., x_n$, and a Boolean formula $F = C_1 \wedge C_2 \wedge ... \wedge C_m$, where each $C_j$ ($1 \leq j \leq m$) is a disjunction of exactly three literals. Without loss of generality, we assume that $x_i$ and $\neg x_i$ do not simultaneously appear in the same clause ($1 \leq i \leq n$).

Does there exist an assignment of truth values to $x_1, x_2, ..., x_n$ such that $F$ is satisfiable?

**The ADSA decision problem.**

Given is a program $P = \langle V_p, \mathcal{O}_p, \mathcal{I}_p, \mathcal{F}_p \rangle$, its safety specification $\mathcal{B}$ and a new safety specification $\mathcal{B}_{new}$. The program $P$ satisfies its specification from $\mathcal{I}_p$, but may not satisfy $\mathcal{B}_{new}$ from $\mathcal{I}_p$.

Does there exist an *advised program* $P_a = \langle V_p^a, \mathcal{O}_p^a, \mathcal{I}_p^a, \mathcal{F}_p^a \rangle$ that meets the requirements of the ADSA problem (stated in Section 3)?

## 4.1 Polynomial Reduction from 3-SAT

In this section, we present a polynomial-time reduction from 3-SAT to ADSA. In particular, we illustrate how for each instance of 3-SAT, we create an instance of ADSA, which includes a program, its set of initial states, its set of accepting states, its objects and their read/write restrictions, its safety specification and the new safety specification that should be captured by the advised program.

**Program variables.** The instance of ADSA has four variables $e, f, g$ and $h$ with the following domains:

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

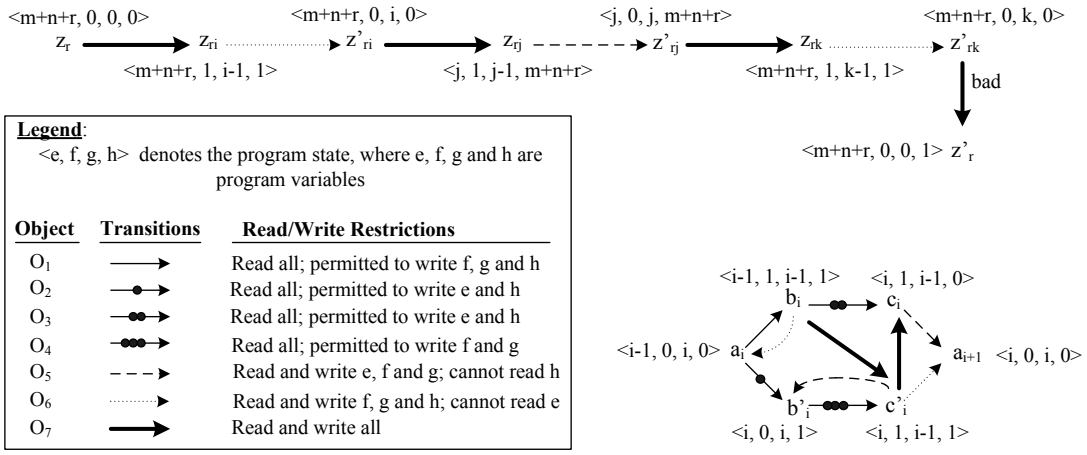


Figure 5: Value assignment to program variables.

**States and transitions.** For each propositional variable $x_i$ ($1 \leq i \leq n$), we consider two program states $a_i$ and $a_{i+1}$ (see part (a) of Figure 4). The state $a_1$ is an initial state and the state $a_{n+1}$ is an accepting state. The program also includes a transition from $a_{n+1}$ back to $a_1$. Thus, starting from $a_1$, the base program simply transitions to $a_2, a_3, \cdots, a_{n+1}$ and back to $a_1$. Part (b) of Figure 4 illustrates the structure of the base program along with a set of safe transitions that can be selected in an advice woven between $a_i$ and $a_{i+1}$. These transitions are as follows (see Figure 4): $(a_i, b_i)$, $(b_i, a_i)$, $(b_i, c_i)$, $(b_i, c'_i)$, $(c_i, a_{i+1})$, $(a_i, b'_i)$, $(b'_i, c'_i)$, $(c'_i, b'_i)$, $(c'_i, c_i)$ and $(c'_i, a_{i+1})$.

For each clause $C_r = x_i \vee \neg x_j \vee x_k$, where $1 \leq r \leq m$ and $1 \leq i, j, k \leq n$, we consider an initial state $z_r$ and seven states reachable from $z_r$, namely states $z_{ri}, z'_{ri}, z_{rj}, z'_{rj}, z_{rk}, z'_{rk}$ and $z'_r$ (see part (c) of Figure 4). Corresponding to the literals in $C_r$, we consider the transitions $(z_{ri}, z'_{ri})$, $(z_{rj}, z'_{rj})$ and $(z_{rk}, z'_{rk})$. We also consider one transition from $z_r$ to $z_{ri}$ and a transition from $z'_{rk}$ to $z'_r$. The states $z_{ri}, z_{rj}, z_{rk}$ and $z'_r$ reachable from $z_r$ are accepting states. Thus, we have

- $\mathcal{I}_p = \{a_1\} \cup \{z_r | 1 \leq r \leq m\}$
- $\mathcal{F}_p = \{a_{n+1}\} \cup \{z_{ri}, z_{rj}, z_{rk}, z'_r |$ for each clause $(C_r = x_i \vee \neg x_j \vee x_k)$ in the 3-SAT formula, 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 except the aforementioned transitions (see Figure 4). The new safety specification $\mathcal{B}_{new}$ prohibits the execution of transitions $(a_i, a_{i+1})$ and $(z'_{rk}, z'_r)$ (see Figure 4). Observe that $\mathcal{B}$ permits the execution of transitions $(a_i, a_{i+1})$ and $(z'_{rk}, z'_r)$, whereas $\mathcal{B}_{new}$ forbids them. Thus, the candidate advices that can be woven between $a_i$ and $a_{i+1}$ are as follows: (i) $\langle b_i, c_i \rangle$, (ii) $\langle b'_i, c'_i \rangle$, and (iii) $\langle b_i, c'_i, c_i \rangle$.

**Objects and their read/write restrictions.** The base program includes seven objects (see Figure 5) whose transitions and read/write restrictions are as follows:

- The first object $O_1$ includes the transitions $(a_i, b_i)$, for all $1 \leq i \leq n$ (see Figure 5). The set of readable variables of $O_1$, denoted $r_1$, is equal to $\{e, f, g, h\}$ and its set of writeable variables is $w_1 = \{f, g, h\}$.
- The set of transitions $(a_i, b'_i)$ comprises the object $O_2$ and $r_2 = \{e, f, g, h\}$ and $w_2 = \{e, h\}$.
- The object $O_3$ includes the transitions $(b_i, c_i)$ (see the arrow with two dots on it in Figure 5). We also have $r_3 = \{e, f, g, h\}$ and $w_3 = \{e, h\}$.
- The fourth object, denoted $O_4$, includes transitions $(b'_i, c'_i)$, and $r_4 = \{e, f, g, h\}$ and $w_4 = \{f, g\}$ (see the arrow with three dots on it in Figure 5).
- For object $O_5$, we have $r_5 = w_5 = \{e, f, g\}$; i.e., $O_5$ cannot read $h$. The object $O_5$ includes transition $(c_i, a_{i+1})$, 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. 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 $O_5$ (see Figure 5).

- The sixth object $O_6$ can read $r_6 = w_6 = \{f, g, h\}$, but cannot read $e$. Its set of transitions includes $(c'_i, a_{i+1}), (b_i, a_i)$ and $(z_{ri}, z'_{ri})$ 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 object $O_7$ can read and write all variables and its set of transitions includes $(a_{n+1}, a_1), (b_i, c'_i)$ 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 $1 \leq i, j, k \leq n$, object $O_7$ includes the following transitions: $(z_r, z_{ri}), (z'_{ri}, z_{rj}), (z'_{rj}, z_{rk})$ and $(z'_{rk}, z'_r)$.

Figure 5 illustrate how we assign different values to the variables in each state of the instance of the ADSA problem. We denote a program state by $\langle e, f, g, h \rangle$.

**Theorem 4.1** The complexity of the reduction is polynomial.

**Proof.** For each propositional variable $x_i$, we consider six states and 10 transitions. Moreover, for each clause in the 3-SAT formula, we have 8 states and seven transitions. Note that since the safety specifications rule out any other transitions except the ones included in the instance of ADSA, we need not be concerned about them. Thus, the size of the instance of the ADSA problem is polynomial in the size of the instance of 3-SAT. □

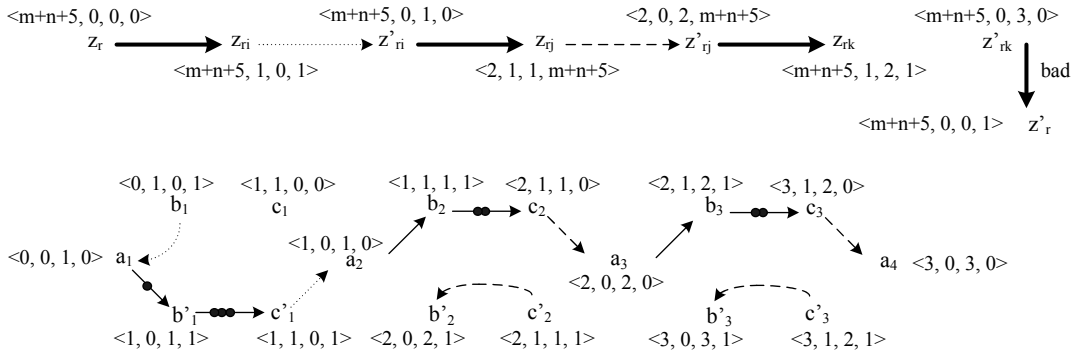


Figure 6: A partial structure of the advised program corresponding to the value assignment $x_1 = false$, $x_2 = true$ and $x_3 = true$ for an example clause $C_5 = x_1 \vee \neg x_2 \vee x_3$.

## 4.2 Correctness of Reduction

In this section, we show that 3-SAT is satisfiable iff the answer to the instance of ADSA (created by the reduction in Section 4.1) is affirmative.

**Lemma 4.2** If the instance of 3-SAT is satisfiable, then there exists an advised version of the instance of ADSA that meets all the requirements of the ADSA problem (defined in Section 3).

**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 advised program as follows:

- If $x_i$ is assigned *true*, then we include transitions $(a_i, b_i), (b_i, c_i), (c_i, a_{i+1})$. Thus the advice $\langle b_i, c_i \rangle$ is woven between $a_i$ and $a_{i+1}$. Since we have included $(c_i, a_{i+1})$, and $(c_i, a_{i+1})$ is grouped with the transitions $(z_{qi}, z'_{qi})$, where $1 \leq q \leq m$, (see the dashed arrow $(z_{rj}, z'_{rj})$ in Figure 5) for any clause $C_q$ in which $\neg x_i$ appears, we must include $(z_{qi}, z'_{qi})$ as well.

- If $x_i$ is assigned *false*, then we include transitions $(a_i, b'_i), (b'_i, c'_i), (c'_i, a_{i+1})$, thereby weaving the advice $\langle b'_i, c'_i \rangle$ between $a_i$ and $a_{i+1}$. Due to the inability of reading $e$, including $(c'_i, a_{i+1})$ results in the inclusion of the transitions $(z_{li}, z'_{li})$, where $1 \leq l \leq m$, (see the dotted arrows $(z_{ri}, z'_{ri})$ and $(z_{rk}, z'_{rk})$ in Figure 5) for any clause $C_l$ in which $x_i$ appears.

- 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 example partial structure of an advised 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 $(z'_{rk}, z'_r)$ is not reached because $x_3 = true$ and the transition $(z_{rk}, z'_{rk})$ is excluded.

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

**Satisfying safety specifications.** Since the new safety specification rules out transitions $(a_i, a_{i+1})$ and $(z'_{rk}, z'_r)$, we have to ensure that the advised 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 bad 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, starting from $z_r$, at least one of the accepting states $z_{ri}, z_{rj}$ or $z_{rk}$ can be reached. Thus, the advised program halts in an accepting state. Moreover, the accepting state $a_{n+1}$ is reached infinitely often due to the way we have woven either aspect $\langle b_i, c_i \rangle$ or aspect $\langle b'_i, c'_i \rangle$ between $a_i$ and $a_{i+1}$. Thus, starting from any initial state, some accepting states will be visited infinitely often; i.e., $\mathcal{F}_p^a \subseteq \mathcal{F}_p$.

Therefore, if 3-SAT is satisfiable, then there exists an advised program (for the instance of the ADSA problem) that satisfies the requirements of the ADSA problem. □

**Lemma 4.3** If there exists an advised version for the instance of ADSA that meets all the requirements of the ADSA problem, then the instance of 3-SAT is satisfiable.

**Proof.** Let $P_a$ be an advised version for the instance of ADSA that meets all the requirements of the ADSA problem. As such, the set of initial states $\mathcal{I}_p^a$ must be equal to the set $\{a_1\} \cup \{z_r | 1 \leq r \leq m\}$. Starting from $a_1$, $P_a$ must execute a safe transition. Otherwise, we reach a contradiction; i.e., either $a_1$ is a deadlock state or the transition $(a_1, a_2)$, which violates the new safety specification is executed. Thus, $P_a$ either includes $(a_1, b_1)$ or $(a_1, b'_1)$, but not both. If $P_a$ includes $(a_1, b_1)$, then we set $x_1$ to *true* in the 3-SAT formula. If $P_a$ 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 $x_i$ since $P_a$ is a deterministic program having at most one outgoing transition from each state. If $P_a$ includes $(a_i, b_i)$, then, from $b_i$, $P_a$ includes either $(b_i, c_i)$ or $(b_i, c'_i)$, but not both. If $P_a$ includes $(b_i, c'_i)$, then, from $c'_i$, $P_a$ must include either $(c'_i, c_i)$ or $(c'_i, a_{i+1})$. If $P_a$ includes $(c'_i, a_{i+1})$, then it must include $(b_i, a_i)$ since these two transitions are grouped due to inability of $O_6$ in reading $e$. As such, the two transitions $(a_i, b_i)$ and $(b_i, a_i)$ make a non-progress cycle in $P_a$. Now, we show that, from $c'_i$, $P_a$ cannot include $(c'_i, c_i)$ either. If $P_a$ includes $(c'_i, c_i)$, then it must include $(c_i, a_{i+1})$, which is grouped with $(c'_i, b'_i)$ due to inability of $O_5$ in reading $h$. Thus, $P_a$ may reach $b'_i$ from $c'_i$ and deadlock in $b'_i$. Thus, if $P_a$ includes $(a_i, b_i)$ from $a_i$, then it must include $(b_i, c_i)$ and $(c_i, a_{i+1})$. In case where $P_a$ includes $(a_i, b'_i)$ from $a_i$, the transition $(b'_i, c'_i)$ must also be included; otherwise $P_a$ deadlocks in $b'_i$. From $c'_i$, $P_a$ cannot include $(c'_i, c_i)$ because it has to include $(c_i, a_{i+1})$ that is grouped with $(c'_i, b'_i)$, which creates a non-progress cycle. Thus, $P_a$ must include $(c'_i, a_{i+1})$ 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; otherwise $z_r$ is a deadlock state, which is a contradiction with $P_a$ being a correct advised program. 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 5) must not be in $P_a$. 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 will 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 $P_a$. Hence, at least one of the transitions $(z_{ri}, z'_{ri})$, $(z_{rj}, z'_{rj})$ or $(z_{rk}, z'_{rk})$ must be excluded in $P_a$; i.e., at least one literal in $C_r$ must be *true*, thereby satisfying $C_r$. For instance, let $\neg x_j$ be the literal that evaluates to *true* in $C_r$; i.e., $x_j = false$. Thus, the

transition $(z_{rj}, z'_{rj})$, and the transitions $(c_j, a_{j+1})$ and $(c'_j, b'_j)$ that are grouped with $(z_{rj}, z'_{rj})$ are excluded. Observe that the assignment of *false* to $x_j$ is consistent with our truth-value assignment. Otherwise, $P_a$ must include $(a_j, b_j)$ and $(b_j, c_j)$, thereby reaching $c_j$, which is a deadlock state because $(c_j, a_{j+1})$ has been excluded. This is a contradiction with $P_a$ being a correct advised program. Therefore, the 3-SAT formula is satisfiable since every clause evaluates to *true*. □

**Theorem 4.4** The problem of automated design of safety aspects is NP-complete in program state space.

**Proof.** The proof of NP-hardness follows from Lemmas 4.2 and 4.3. The proof of NP membership is straightforward, hence omitted. Therefore, the ADSA problem is NP-complete. □

# 5 Concluding Remarks and Future Work

We focused on the problem of designing aspect-oriented programs for crosscutting safety concerns. Specifically, we formulate the problem of designing a *correct* advised version of a base program that satisfies a safety concern while preserving the correctness of the base program. We then illustrated that designing correct advised programs is NP-complete in program state space. In other words, in general, it is unlikely that the design of correct aspect-oriented programs can be done efficiently (unless $P = NP$). This is a counterintuitive result as previous work [13,21] illustrates that designing safety aspects for non-deterministic programs is NP-complete. Based on our hardness result in this paper, we believe that the complexity of design is not due to non-determinism, instead it is due to the inability of one object in reading the state of other objects.

As an extension of this work, we will focus on the design of sound *heuristics* that reduce the complexity of design at the expense of losing completeness. That is, if such heuristics generate an advised program, then the advised program is correct by construction, however, the heuristics may fail to create an advised program while there exists one. As such, for different programs and aspects of different nature, we may need different heuristics. To provide tool support for designers, we will create an extensible software framework that integrates sound heuristics for automated design of aspects. We foresee two categories of users for our framework, namely, (1) *developers of heuristics* who will focus on designing new heuristics and integrating them in our framework, and (2) *designers of aspect-oriented programs* who will use the built-in heuristics in our framework to automatically design advised programs.

# References


[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[2] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION'01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209, 2001.

[3] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):890–910, 2004.

[4] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):7, 2007.

[5] D.R. Smith. Aspects as invariants. In *Automatic Program Development: a Tribute to Robert Paige, O. Danvy, F. Henglein, H. Mairson, and A. Pettorosi, Eds., Springer-Verlag LNCS. (earlier version in Proceedings of GPCE-04, LNCS 3286).*, pages 39–54. 2006.

[6] Slim Kallel, Anis Charfi, Mira Mezini, and Mohamed Jmaiel. Combining formal methods and aspects for specifying and enforcing architectural invariants. In *9th International Conference on Coordination Models and Languages*, pages 211–230, 2007.

[7] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 89–98, 2002.

[8] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. *ACM SIGSOFT Software Engineering Notes*, 29(6):137–146, 2004.

[9] Dianxiang Xu, Izzat Alsmadi, and Weifeng Xu. Model checking aspect-oriented design specification. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 491–500, 2007.

[10] Thomas Colcombet and Pascal 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.

[11] D.R. Smith. Requirement enforcement by transformation automata. In *Sixth Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 5–15, 2007.

[12] Slim Kallel, Anis Charfi, and Mohamed Jmaiel. Using aspects for enforcing formal architectural invariants. *Electronic Notes on Theoretical Computer Science*, 215:5–21, 2008.

[13] Sandeep S. Kulkarni and Ali Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, 2005.

[14] M.R. Garey and D.S. Johnson. *Computers and Interactability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.

[15] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[16] 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.

[17] Ali Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, May 2005.

[18] 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.

[19] 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.

[20] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS*, 23(2), March 2001.

[21] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems (OPODIS)*, 2008.