On Atomicity Enforcement in Concurrent Software via Discrete Event Systems Theory

Yin Wang, Peng Liu, Terence Kelly, Stéphane Lafortune, Spyros Reveliotis, and Charles Zhang

***Abstract*— Atomicity violations are among the most severe and prevalent defects in concurrent software. Numerous algorithms and tools have been developed to detect atomicity bugs, but few solutions exist to automatically fix such bugs. Some existing solutions add locks to enforce atomicity, which can introduce deadlocks into programs. Our recent work avoids deadlock bugs in concurrent programs by adding control logic synthesized using Discrete Event Systems theory. In this paper, we extend this control framework to address single-variable atomicity violation bugs. We use the same class of Petri net models as in our prior work to capture program semantics, and handle atomicity violations by control specifications in the form of linear inequalities. We propose two methodologies for synthesizing control logic that enforces these linear inequalities without causing deadlocks; the resulting control logic is embedded into the program's source code by program instrumentation. These results extend the scope of concurrency bugs in software systems that can be handled by techniques from control engineering. Case studies involving two real Java programs demonstrate our solution procedure.**

I. Introduction

Computer and software engineering is a rich application area for control systems technology. Indeed, classical control theory, based on continuous-variable and time-driven models, has been applied to computer systems problems, such as throughput stabilization, and has achieved commercial success [11]. Many important problems in computer and software engineering, however, are discrete in nature and naturally fall into the realm of discrete-variable and event-driven systems, i.e., Discrete Event Systems (DES). The growth of software defects has paralleled and shadowed the rapid advancement in computer technology. Relevant issues such as safety and consistency are inherently discrete-event problems. Recently, there has been a growing interest in the application of concepts and techniques from DES to software systems and embedded systems; see, e.g., [6], [8], [10], [13], [20], [31]. In particular, the paradigm of *controlling software execution to avoid software defects at runtime* has received significant attention.

Multicore architectures are pervasive in today's computer systems. They have forced programmers to write concurrent programs, a task known to be time-consuming and error-prone. Deadlocks, races, and atomicity violations are the three major types of concurrency bugs [22]. (These terms are defined in Section II-A.) There has been a substantial amount of work on practical tools that automatically detect these types of software bugs. While techniques that detect races and deadlocks are relatively mature [9], [27], atomicity violations are known to be difficult to detect and they have been the focus of several studies; see, e.g., the recent work [12], [28] and the references therein. While numerous studies are dedicated to bug detection, *fixing* detected bugs largely remains a manual operation in practice. A few exceptions exist, and they are often based on heuristic techniques that may not be provably correct. This is because solutions that fix deadlock [15] and atomicity violation [14] bugs may introduce new deadlocks. Another approach uses rules to delay thread execution when there is a potential atomicity violation [5].

In our prior work in the *Gadara project* [1], we have addressed the avoidance of *circular-mutex-wait deadlocks* in concurrent programs using Petri net models and controller synthesis techniques from DES theory; see [30], [31]. This important problem is prevalent in today's multicore computer architectures and it occurs when two or more concurrent threads holding exclusive access to shared data, through mutual exclusion locks, or *mutexes*, are caught in a circular wait situation. Exploiting the fact that Petri nets modeling mutex-based concurrent programs, called Gadara nets and formally defined in [32], are *one-safe* (i.e., their places contain either zero or one token), we have developed two specialized control algorithms for provably avoiding circular-mutex-wait deadlocks. The first technique is called Iterative Control of Gadara nets, or ICOG, and it was first presented in [18]. ICOG is an iterative procedure based on the detection of a special class of *siphons* in the Gadara net model of a program, coupled with the the Petri net control technique of Supervision Based on Place Invariants (SBPI) [23] for enforcement of linear inequalities upon the net marking. ICOG guarantees finite convergence and correctness of the resulting control logic, implemented in the form of control places (also called monitor places) that control the original Gadara net. In parallel, we have developed an alternative and complementary technique called Marking Separation using Linear Classifiers, or MSCL, which linearly separates unsafe and safe markings using Mixed Integer Programming (MIP) in a single iteration [24]. The control logic is also implemented in the form of control places. Both ICOG and MSCL synthesize maximally permissive control logic. In its current implementation, MSCL requires explicit state space enumeration but it provably minimizes the number of

Y. Wang and T. Kelly are with HP Labs, Palo Alto, CA 94304, USA, {yin.wang,terence.p.kelly}@hp.com.

P. Liu and C. Zhang are with the Department of CSE, the University of Science and Technology, Hong Kong, {lpxz,charlesz}@ust.hk.

S. Lafortune is with the Department of EECS, the University of Michigan, Ann Arbor, MI 48109, USA, stephane@eecs.umich.edu.

S. Reveliotis is with the School of ISyE, Georgia Institute of Technology, Atlanta, GA 30332, USA, spyros@isye.gatech.edu.

This work was partially supported by Hong Kong RGC GRF grants, US NSF grants CCF-0819882 and CCF-1138860 (Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering), and the HP Labs Innovation Research Program.

```
Xfer(i,j,x) {
  a[i]-=x;
  a[j]+=x;
}
```
(a) Race

```
Xfer(i,j,x) {
  lock(l[i]);
  a[i]-=x;
  unlock(l[i]);
  lock(l[j]);
  a[j]+=x;
  unlock(l[j]);
}
```
(b) Atomicity violation

```
Xfer(i,j,x) {
  lock(l[i]);
  lock(l[j]);
  a[i]-=x;
  a[j]+=x;
  unlock(l[j]);
  unlock(l[i]);
}
```
(c) Deadlock

Fig. 1: Transfer balance `x` from account `i` to account `j`

control places added, an important consideration for the run-time overhead of the control logic. Other authors have also addressed the correctness of concurrent software using DES techniques; see, e.g., [8], [13].

This paper extends the Gadara control framework to address other classes of concurrency bugs beyond circular-mutex-wait deadlocks. Specifically, we consider a class of atomicity violation bugs called "single-variable atomicity violations." We employ Gadara nets as defined in [32] to capture this class of bugs by control specifications expressed as linear inequalities on the net marking. We show how to adjust the construction of the Gadara net at modeling time to make this possible. After adding one monitor place to enforce each linear inequality using SBPI, ICOG is then directly applied on the resulting controlled Gadara net to eliminate potential deadlocks introduced by these control places. If one is interested in obtaining the minimum-size controller (in terms of number of added control places), then MSCL can be employed, albeit the process is more involved. For this purpose, we develop a novel state space exploration algorithm to identify so-called "boundary" atomicity violation markings. Next, the standard automaton-based supervisory control algorithm for the supremal nonblocking controllable sublanguage is applied to identify unsafe markings (due to uncontrollability or blocking) in the remaining set of markings. Finally, we separate both the original atomicity violation markings and the above unsafe markings from the set of safe markings using the MIP formulation of MSCL.

This paper is organized as follows. First, background discussion on concurrency bugs is given in Section II, then, necessary background on the control framework adopted in this paper is presented in Section III. Section IV presents the approach that we propose for tackling single-variable atomicity violation bugs by DES control techniques, leveraging both ICOG and MSCL. Experimental results are given in Section V and concluding remarks follow in Section VI.

II. Background on Concurrency Bugs

A. Races, Atomicity Violations, and Deadlocks

Multiple threads in concurrent programs communicate via shared memory for the sake of efficiency. A *data race* occurs if two threads access the same memory location simultaneously, and at least one of them is a *write* operation [3]. Figure 1 is an example of a balance transfer function. There is an array of accounts, denoted by `a[1]`, `a[2], ...`. The function moves amount `x` from account `a[i]` to account `a[j]`. A race occurs if two threads call this function simultaneously to transfer money to or from the same account. Programmers should avoid data races because the output is unpredictable due to different execution interleavings. Furthermore, code optimization techniques employed by modern computer architectures and compilers can lead to results "out-of-thin-air" for programs with races [2]. A data race can be trivially fixed by protecting each memory location with a mutex lock. Since at most one thread can hold a mutex at any time, there is no simultaneous access to the same memory location. In Figure 1b, we use a lock array whose size is identical to the account array to protect the latter. The accesses to `a[i]` and `a[j]` are protected by locks `l[i]` and `l[j]`, respectively. The program is now race free.

However, adding locks to protect merely the problematic statements does not guarantee the *atomicity* requirements of the program as a whole. The concept of atomicity originates from database systems where it intuitively means "all or nothing": either a transaction completes as a whole or there is no effect at all to the database state. For programming languages with transaction constructs (sometimes called *atomic sections*), as in transactional memory programs [16], an execution is atomic if the computation result is equivalent to some serial execution, i.e., transactions are executed one after another; this is referred to as "single-global-lock semantics" and corresponds to a notion called *serializability* [25]. For most programming languages, the notion of transaction is not defined. Existing atomicity violation detection tools use heuristics to determine the boundary of transactions, e.g., a *critical section* protected by some lock is often considered as the scope of a transaction. Assuming function `Xfer` must be atomic, the race-free version in Figure 1b may violate atomicity. For example, if someone queries both accounts `a[i]` and `a[j]` between statements `unlock(l[i])` and `lock(l[j])`, the result is incorrect and not equivalent to any serial execution.

On the other hand, moving the acquisition of `l[j]` to the beginning of `Xfer` enforces atomicity, but a deadlock occurs if two threads concurrently transfer money between two accounts in opposite directions. This is an instance of the circular-mutex-wait deadlock mentioned in Section I, where a set of threads is holding a set of locks and waiting to acquire locks in the same set. In prior work in the Gadara project, we developed two new control-theoretic approaches to eliminate circular-mutex-wait deadlocks in concurrent programs: see [24] and [18], [19]. This paper focuses on extending these solution paradigms to certain classes of atomicity violation bugs.

Detecting atomicity violations is substantially more difficult than deadlock detection. First, unlike a deadlock, an atomicity violation does not stop the program from running. The bug may propagate and eventually crash the program at a location distant from the atomicity violation, or the bug may silently cause incorrect output. Second, concurrent programs are nondeterministic. Inputs, thread scheduling, and hardware characteristics all affect the execution, which makes it difficult to reproduce an atomicity violation. Simply monitoring and debugging the code can make the bug disappear, a phenomenon called a *Heisenbug*. In addition, it is computationally expensive to determine whether an execution is serializable [25]. There are actually several notions of serializability that have been proposed and studied; a discussion of these is beyond the scope of this paper. We note that testing for the notion of view serializability is NP-complete. The notion of conflict serializability can be verified in quadratic time using the *conflict graph*, but this complexity is still too high for runtime instrumentation. Consequently, atomicity violation detection has been one of the main research themes in the programming language community for the past decade; we refer the reader to two recent papers and the references therein [12], [28].

Finally, we mention that after detection, fixing atomicity violation bugs remains largely manual. A recent proposal adds locks to protect atomic sections, but it may introduce deadlocks that eventually require manual inspection and fix [14].

B. Single-variable Atomicity Violations

In its most general form, an atomicity violation may involve multiple variables with different combinations of read/write operations [29]. Detecting multi-variable atomicity violations is very challenging. Most existing atomicity violation detection tools focus on the *single-variable* case, which is also the focus of this paper.

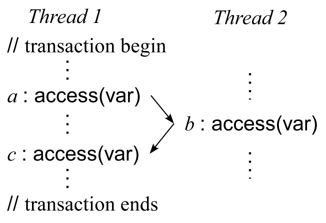


Fig. 2: Atomicity violation

Figure 2 is the unique pattern of execution traces exposing a single-variable atomicity violation [29]. More complicated single-variable atomicity violations are composed from this pattern. The three statements a, b, c all access the same variable. The trace of Thread 1, a to c, is executed within a transaction, e.g., within a critical section protected by a lock. Statement b may or may not be inside a transaction. Intermediate values within a transaction should not be accessed by other threads. Therefore, an atomicity violation can occur if Thread 2 executes b while Thread 1 is in between a and c, denoted as $a \to b \to c$.

The precise definition of an atomicity violation differentiates read and write accesses to `var`. Since each access is either read or write, there are $2^3 = 8$ combinations in total. If b is a write, all four combinations of a and c violate atomicity. If b is a read, an atomicity violation occurs only when both a and c are writes. Therefore, there are five combinations that violate atomicity [29]. Most atomicity violation detection tools differentiate read/write accesses, and detect only the violating combinations. In this paper, however, we use a uniform control specification for all five combinations.

Note that Figure 2 is an execution trace instead of the actual code. The trace between a and c is only one possible path in the code that goes from a to c. Furthermore, the two statements may not be distinct, e.g., they can be the same statement executed in different iterations of a loop. Statement b need not be distinct from a or c either, which implies the same body of code is executed by two threads. Overall, for a single variable, there can be multiple atomicity violation traces that correspond to different code sections. Different variables may result in different violation traces, for which the corresponding code sections may overlap. The situation is different from multi-variable atomicity violations. The solution procedure we present in this paper handles the former but not the latter. For example, Figure 1b is a multi-variable atomicity violation with two variables; considering `Xfer` as the scope of the transaction, there is no single-variable atomicity violation in this case.

III. Control Framework for Concurrent Programs

Before we can present the specifics of the contribution of this paper, it is necessary to review the methodology adopted in the Gadara project for online control of concurrent software based on offline synthesis of control logic. In the offline phase, a suitable model of the program is extracted from the source code at compile time in the form of an enhanced *Control Flow Graph* (or CFG). The enhanced CFG is translated into a Petri net model. Subsequently, techniques from DES control theory are used to synthesize control logic that will enforce the given specifications, namely, avoidance of circular-mutex-wait deadlocks in the prior work in the Gadara project. This synthesis process must also enforce the optimality criterion of maximal permissiveness. Furthermore, a crucial synthesis constraint is that the control flow of the program cannot be altered; in other words, the control logic cannot force the program along a given branch (of an "if/then" statement for instance), since this would alter structurally the behavior of the program. This is ensured by modeling such transitions as *uncontrollable*. The control logic is allowed to delay a given step of the program, e.g., the acquisition of a mutex lock. The last step of the offline phase is the instrumentation of the synthesized logic into the program's source code. In the online phase, the program executes with its embedded control logic. Since the offline synthesis step is based on formal theories, it is provably correct (with respect to the model), and, thus, the execution of the program is guaranteed to satisfy the given specification. For performance reasons, the overhead of executing the embedded control logic should be minimized as well.

A. Program Modeling

A CFG is a high-level graphical representation of all code execution paths that might be traversed by the program. It is constructed during compilation of the program. We extract the CFG and augment it with additional information about lock variables and lock functions. The enhanced CFG is a directed graph. It is therefore straightforward to convert it into a Petri net model. In the resulting Petri net, lock acquisition and release statements are modeled as transitions. So are control transfers between the CFG's *basic blocks*, i.e., branch-free sets of consecutive instructions. Places in the Petri net represent statements in the program other than lock operations. In [32], we formally defined the class of Petri nets that arises from the above modeling process when considering circular-mutex-wait deadlock-avoidance as the given specification; we have called them *Gadara nets*. (We assume the reader is familiar with standard Petri net concepts and notations.)

Definition 1: [32] Let $I_{\mathcal{N}} = \{1, 2, ..., n\}$ be a finite set of process subnet indices. A Gadara net is an ordinary, self-loop-free Petri net $\mathcal{N}_G = (P, T, A, m_0)$ where

1) $P = P_0 \cup P_S \cup P_R$ is a partition such that: a) $P_S = \bigcup_{i \in I_{\mathcal{N}}} P_{S_i}, P_{S_i} \neq \emptyset$, and $P_{S_i} \cap P_{S_j} = \emptyset$, for all $i \neq j$; b) $P_0 = \bigcup_{i \in I_{\mathcal{N}}} P_{0_i}$, where $P_{0_i} = \{p_{0_i}\}$; and c) $P_R = \{r_1, r_2, ..., r_k\}$, $k > 0$.
2) $T = \bigcup_{i \in I_{\mathcal{N}}} T_i, T_i \neq \emptyset, T_i \cap T_j = \emptyset$, for all $i \neq j$.
3) For all $i \in I_{\mathcal{N}}$, the subnet $\mathcal{N}_i$ generated by $P_{S_i} \cup \{p_{0_i}\} \cup T_i$ is a strongly connected state machine. There are no direct connections between the elements of $P_{S_i} \cup \{p_{0_i}\}$ and T_j for any pair (i, j) with $i \neq j$.
4) $\forall p \in P_S$, if $|p \bullet| > 1$, then $\forall t \in p\bullet, \bullet t \cap P_R = \emptyset$.
5) For each $r \in P_R$, there exists a unique minimal-support P-semiflow, Y_r, such that $\{r\} = \|Y_r\| \cap P_R$, $(\forall p \in \| Y_r \|)(Y_r(p) = 1)$, $P_0 \cap \| Y_r \| = \emptyset$, and $P_S \cap \|Y_r\| \neq \emptyset$.
6) $\forall r \in P_R, m_0(r) = 1, \forall p \in P_S, m_0(p) = 0$, and $\forall p_0 \in P_0, m_0(p_0) \geq 1$.
7) $P_S = \bigcup_{r \in P_R} (\|Y_r\| \setminus \{r\})$.

Due to space limitations, we refer the reader to [32] for a discussion of the elements of this definition in relation to multithreaded software using mutex locks. Briefly, each process subnet represents one thread (which could be instantiated more than once), P_0 is the set of idle (or initial) places for these threads, P_S the set of *operation* places of the process subnets, representing the basic blocks in the CFGs of the respective threads, and P_R the set of *resource* places, namely mutex locks, that are shared among the process subnets. Conditions 5) and 6) are key conditions that arise from the fact that a mutex is an *exclusive lock* that is acquired then released, resulting in the semiflow condition 5) with initial marking as stated in condition 6). Condition 7), *which will be relaxed later in this paper*, refers to the fact that for the purpose of circular-mutex-wait deadlock-avoidance, the Petri net model of the program can be *pruned* so that the process subnets only model the critical sections of the program, where one or more mutex locks are being held.

B. Supervision Based on Place Invariants

Supervision Based on Place Invariants (SBPI) is a general control framework for characterizing forbidden markings and synthesizing control logic that avoids them. The control specification of SBPI is a linear constraint (l, b) such that

$$l^T m \leq b \tag{1}$$

where l is an integer weight (column) vector, m is the Petri net marking, and b is an integer scalar. Constraint (1) states that the weighted sum of the number of tokens in each place should be less than or equal to the given constant b. We want to avoid all markings that violate the linear constraint in (1), while allowing all other markings. This is achieved by adding a new *control place* to the net with arcs connecting to existing transitions. This added control place follows the same Petri net dynamics, i.e., it blocks an output transition when its token count is insufficient, and allows it otherwise. Specifically, we solve the following equation for the connectivity of the new control place, denoted by c, in terms of the Petri net incidence matrix D: $D_c = -l^T D$. A solution exists iff $m_0(c) = b - l^T m_0 \geq 0$. We refer the reader to [23] for detailed coverage of SBPI.

The control place calculated by SBPI is connected to a subset of transitions in the original Petri net. For the concurrent software control applications that interest us, this subset is typically very small. All other transitions in the original Petri net are unaffected by the added control logic, which operates in a highly concurrent fashion. This contrasts sharply with approaches to concurrency control that apply centralized checks every time certain operations are performed (e.g., Banker's Algorithm [7] and those based on automaton models).

Linear Separability between safe and unsafe markings is a necessary and sufficient condition for avoiding exactly the latter when using SBPI. The reachable marking space of a Petri net N, denoted as $\mathcal{R}(N)$, is a subset of the integer vector space $\mathbb{N}^n$, where n is the number of places in the Petri net. The linear constraint (l, b) is a hyperplane that partitions the space. Avoiding unsafe markings using SBPI is equivalent to finding a set of hyperplanes that separate these markings from safe markings; it is also equivalent to the condition that unsafe markings must be outside of the convex hull constructed from safe markings [24]. Formally:

Definition 2: Consider two vector sets S and U from an n-dimensional vector space. S is *linearly separable* from U if there exists a set of k linear constraints $(l_1, b_1), ...(l_k, b_k)$ such that

$$\forall m \in S, \forall i \in \{1, ..., k\} : l_i^T m \leq b_i \tag{2}$$

$$\forall m \in U, \exists j \in \{1, ..., k\} : l_j^T m > b_j \tag{3}$$

We can add a control place for each linear constraint (l_i, b_i), and the "conjunction" of these places achieves the desired separation. More specifically, inequalities (2) indicate that every marking in S should satisfy all linear constraints, and therefore are to be allowed by the control logic. On the contrary, inequalities (3) state that each marking in U must violate at least one constraint, and therefore are to be forbidden by the control logic. The following theorem recalled from [24] guarantees the existence of a linear classifier for any subset of reachable markings we want to avoid when dealing with nets with *binary markings*.

Theorem 1: [24] Any two disjoint sets of n-dimensional binary markings, i.e., $S \subseteq \{0, 1\}^n, U \subseteq \{0, 1\}^n$, and $S \cap U = \emptyset$, are linearly separable.

C. Control using Iterative Siphon Analysis

We have shown in [32] that circular-mutex-wait deadlock avoidance of a concurrent program is equivalent to *liveness* of its Gadara net model, where the notion of liveness is the standard one: every transition is eventually firable from every reachable marking. The first control methodology developed in the Gadara project for liveness enforcement in Gadara nets that is leveraged in the present work is called "Iterative Control Of Gadara nets" and abbreviated as ICOG. Due to space limitations, we refer the reader to [18], [19] for a description of ICOG and a formal study of its properties. Note that ICOG deals with uncontrollable transitions (cf. [18]), an essential requirement in the present application domain as was mentioned at the beginning of this section. Briefly, ICOG exploits the structural property of *siphons* in Petri net structures, where a set of places forms a siphon if its set of input transitions is a subset of its set of output transitions. Violations of liveness in a Gadara net can be completely captured by the reachability of certain types of siphons, called "resource-induced deadly marked siphons." These can be detected by solving an MILP problem, and an appropriate linear inequality can be written to prevent their reachability. Then, SBPI is used to enforce the desired linear inequality by means of a control place. ICOG is by necessity an iterative procedure as the addition of control places may affect the liveness of the controlled Gadara net, i.e., it may result in new resource-induced deadly marked siphons. We have shown in [18] that ICOG is finitely-convergent and that it enforces liveness of the controlled Gadara net in a maximally-permissive manner. These results extend the large body of literature on liveness-enforcement of special classes of Petri nets (see the survey paper [17] and the references therein) by suitably exploiting the structural properties of Gadara nets.

D. Control using Marking Separation with Linear Classifiers

Instead of iteratively searching for absence of liveness by siphon analysis as in ICOG, we can find *all unsafe markings* in the reachability graph of the original Gadara net, and then calculate a minimum set of linear inequalities to separate them from all the safe markings. After this is done, we use SBPI to implement the desired linear inequalities back on the Petri net by control places. This is the approach that is formally developed in [24], along with experimental results that demonstrate its scalability. The resulting methodology is called *Marking Separation with Linear Classifiers* and abbreviated as MSCL. The process of classifying reachable markings as "safe" or "unsafe" in a manner that deals with uncontrollable events and guarantees nonblocking (i.e., liveness) essentially requires solving the standard Supervisory Control Problem with Blocking (SCPB-NB) in the terminology of [4], which is done by computing the supremal nonblocking controllable sublanguage [26]. The calculation of the minimum linear classifier is based on an MIP formulation. Overall, MSCL has double exponential computation complexity in the worst case, the unfortunate price of obtaining the minimum linear classifier and working directly with the reachabililty graph of the net. However, several optimality-preserving pre-processing steps were developed in [24], and these can reduce the number of constraints in the MIP formulation by several orders of magnitude. For example, we only need to separate *boundary* unsafe markings, i.e., markings one edge away from safe markings, from safe markings. Furthermore, it is proved that the minimum-size classifier can be achieved using only non-negative coefficients. Therefore, it suffices to separate *maximal* safe markings from *minimal* unsafe markings. Here a marking m is no less than m' iff there exits an entry i such that $m[i] \geq m'[i]$. Finally, there is another MIP formulation with substantially less constraints that finds a suboptimal classifier. The formulation is based on the idea of greedy algorithms. It iteratively finds linear inequalities that separate

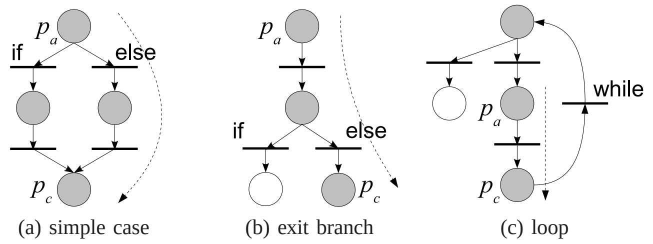


Fig. 4: Places to be protected to enforce atomicity

as many unsafe markings as possible. The reader is referred to [24] for full details. In the next section, we leverage and extend MSCL to the problem of atomicity bug elimination.

IV. Atomicity Enforcement

This section describes our solution to fixing single-variable atomicity violation bugs.

A. Program Modeling

Both ICOG and MSCL add control places to enforce atomicity, which can introduce deadlocks among newly added control places, or with existing locks in the program. Therefore our Petri net model includes existing locks and code sections protected by them (which give rise to semiflows in the net), using the technique described in Section III-A. In addition, our Petri net model captures atomicity violation bugs, which may or may not be protected by locks. For example, we model Statements a, b, c in Figure 2 by places p_a, p_b, p_c, respectively, shown in Figure 3. Our Petri net model satisfies all conditions in Definition 1 except 7), because semiflows of resource places in general may not subsume all code sections involved in atomicity violations. The rest of this section develops new control specifications and modifies existing control synthesis algorithms to address atomicity violation bugs.

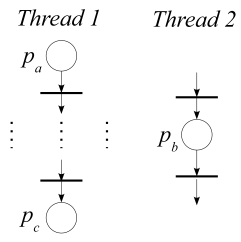


Fig. 3: Modeling atomicity violation

B. Control Specification

To prevent atomicity violations and races relevant to the trace in Figure 2, it is necessary and sufficient to allow at most one thread to execute either the trace from a to c, or statement b, i.e.,

$$\sum_{p \text{ on } Trace_{ac}} m(p) + m(p_b) \leq 1 \qquad (4)$$

Single-threaded execution of such code segments obviously avoids races and atomicity violations. We show the necessity of the condition by contradiction. Assume there are two threads in $Trace_{ac}$ and p_b. If one thread is in $Trace_{ac}$ and another in p_b, it is the same atomicity violation $a \to b \to c$. If both threads are in p_b, i.e., simultaneous access to `var`, it is a race. If both threads are in $Trace_{ac}$, it is a different atomicity violation $a \to a \to c$, where the second statement a corresponds to the second thread entering $Trace_{ac}$.

However, specification (4) protects only the given execution trace, and ignores the control flow structure of the program. It can miss potential atomicity violations not exposed by the trace. Figure 4a shows an example. The dashed arrow from p_a to p_c represents the execution trace from statement a to c, which traverses the `else` branch. Protecting only this branch does not limit the number of tokens in the `if` branch. An atomicity violation occurs if Thread 1 is in the `if` branch and Thread 2 in p_b. To solve this problem, we propose to use the following control specification that is stronger than (4):

$$\sum_{p_a \rightsquigarrow p \text{ and } p \rightsquigarrow p_c} m(p) + m(p_b) \leq 1 \tag{5}$$

where $p \rightsquigarrow q$ iff place p can reach place q in the Petri net, within the transaction that encapsulates a and c. There is no ambiguity with $p \rightsquigarrow q$ because the process subnet is a state machine. Finding all places p that satisfy $p_a \rightsquigarrow p$ and $p \rightsquigarrow p_c$ is a graph search problem that can be calculated in linear time.

Specification (5) allows at most one token for all places along all paths from p_a to p_c. Now the `if` branch in Figure 4a is also protected. In Figure 4b, the place in the `if` branch can not reach p_c and therefore it is not protected. In this case, specification (4) and (5) are equivalent. In Figure 4c, the top place is included in specification (5) because it is in a loop with p_a and p_c. In fact, if p_a and p_c are in a loop within a transaction, all places in the loop are included in specification (5). This extra protection is necessary because we do not know how many iterations the loop will be executed.

As mentioned in Section II-B, statement b in Figure 2 can be the same as a or c if there are two threads executing the same code section. It can also be on the paths from a to c. In this case, we should omit the term $m(p_b)$ in both (4) and (5) since it is already counted. We leave the term in the specifications to clarify the idea.

In practice, there can be multiple atomicity violation traces detected for the same shared variable. These traces may or may not overlap. We can combine all control specifications for all atomicity violations together, and require that the summation of tokens in all the relevant places should be no more than one. Each place is counted at most once even if it is in the overlapping section of multiple atomicity violations. Combining multiple atomicity violations of the same variable together not only simplifies the control specification, but also avoids potential atomicity violations not exposed by the traces. For example, if there are two atomicity violations, $a_1 \to b_1 \to c_1$ and $a_2 \to b_2 \to c_2$, limiting the token in all places along a_1 to c_1, a_2 to c_2, and b_1, b_2 also avoids the potential atomicity violation $a_1 \to b_2 \to c_1$.

C. Using Iterative Siphon Analysis

Using either specification (4) or (5), we can synthesize a control place using SBPI. Since places in the process subnet have no initial token, this control place has exactly one initial token, which is acquired at transitions that enter the places included in the specification, and released at transitions leaving these places. Each control place has the same initial number of tokens as a lock place, and its connectivity to process subnets also exhibits the same pattern. After these control places are added, if we trim all places not protected by any lock or control place, the resulting Petri net is exactly a Gadara net according to Definition 1, with condition 7) re-included. Therefore, deadlocks introduced by the added control places, as well as existing deadlocks that do not involve control places, can be avoided using ICOG [18].

The idea of adding locks to fix atomicity violation has occurred in the literature [14], but deadlocks introduced may require manual intervention. In our previous work, we applied an iterative siphon-based control synthesis to enforce atomicity without introducing any deadlock [21]. However, unlike ICOG, the algorithm employed in [21] does not keep track of the unsafe markings already avoided by synthesized control places, and therefore it may not converge.

D. Using Marking Separation with Linear Classifier

As mentioned in the previous subsection, after the synthesis of control places that enforce the specification for atomicity, the Petri net becomes a Gadara net. Therefore the deadlocks introduced by the added control places can be avoided by MSCL without modification. However, the linear classifier obtained may not be minimal with respect to the *original* program, because multiple atomicity violations, or atomicity violations and deadlocks together, can sometimes be avoided by a single linear inequality. To obtain the minimum size classifier, we should *first* find *all* atomicity violation markings and deadlock markings, and *then* apply MSCL to separate all of them in one pass. However, atomicity violations can occur in code sections not protected by locks. The corresponding places in our Gadara net (without condition 7) in Definition 1) can have an infinite number of tokens, and therefore an unbounded reachability graph. Even if we assume a bounded number of threads, i.e., a finite initial number of tokens in idle places, these unprotected places cause explosion in the state space. Furthermore, non-binary markings do not guarantee linear separability in general. Fortunately, our atomicity control specification requires these places to have no more than one token. *Therefore all safe markings are still binary.* The following theorem guarantees the existence of a linear classifier to separate any set of binary markings from a disjoint set of non-binary markings.

Theorem 2: In an n-dimensional vector space, $\forall S \subseteq \{0,1\}^n$, $\forall U \subseteq \mathbb{N}^n$, if $\mathcal{S} \cap \mathcal{U} = \emptyset$, then S is linearly separable from U.

Proof: We prove the theorem by construction of the linear constraint set. First we add constraints $p_1 \leq 1$,...,$p_n \leq 1$ to the set to separate all non-binary markings. In the binary vector space, the proof of Theorem 1 (Proposition 2 in [24]) shows that any binary vector can be isolated from all other binary vectors using a linear constraint. We add these constraints for every binary vector in U. The two sets of linear constraints separates S from U. ■

In Section III-D, we stated that the minimum size linear classifier can be obtained by separating only boundary unsafe markings from safe markings. We apply this result directly to the state space exploration process, and do not search beyond boundary unsafe markings. Markings beyond boundary may be reachable from other safe markings and will be explored eventually, or they can only be reached from unsafe markings and are not explored at all. These unexplored markings do not affect the correctness or optimality of our control synthesis since they will not be reachable in the controlled system.

Algorithm 1 Linear Classifier Calculation

```
Input: Gadara net $\mathcal{N}_G = (P, T, A, m_0)$, and $SPEC$
Output: Linear classifier $L$ as a set of linear inequalities
 1: $queue = \{m_0\}, visited = \emptyset, S = \emptyset, U = \emptyset$
 2: while $queue \neq \emptyset$ do
 3:    $m = queue$.remove()
 4:    for all $m' \in m$.nextStates() and $m' \notin visited$ do
 5:       $visited = visited \cup \{m'\}$
 6:       if $m'$ satisfies $SPEC$ then
 7:          $S = S \cup \{m'\}$
 8:          $queue$.add($m'$)
 9:       else
10:          $U = U \cup \{m'\}$
11:       end if
12:    end for
13: end while
14: $S'$ = supremal nonblocking controllable subset of $S$
15: return  $L =$MSCL$(S', U \cup (S \setminus S'))$
```

Overall, this result in the following algorithm that integrates all the steps of our proposed approach:

Algorithm 1 is the overall control synthesis procedure. It stops exploration at unsafe markings, as shown in the else branch at line 10. After state space exploration, we apply the automaton-based supervisory control theory initiated in [26] (see also [4]) to find the supremal nonblocking controllable subset in the safe state space S, at line 14. Finally, the last step applies MSCL as developed in [24] to find the minimum size classifier.

V. Case Studies

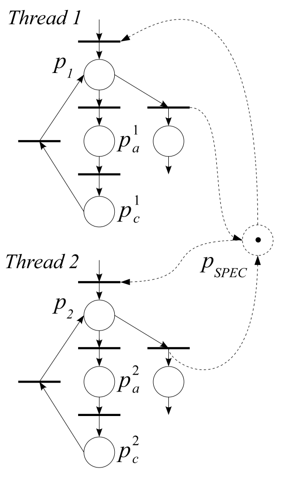
We study atomicity violations from two real Java programs, and present the control synthesis results using both ICOG and Algorithm 1 (with MSCL) in this section. In our previous work, we developed an atomicity violation detection tool based on dynamic analysis, and summarized the analysis result on a set of Java programs [12]. Here we pick two representative examples to illustrate the control synthesis procedure. (The code snippets shown have been simplified for the sake of presentation.)

Our first example in Figure 5 is from RayTracer, a standard benchmark program from the Java Grande Forum. The two threads are executing two code sections that are almost identical. Each section is a `synchronized` block, which implies single-threaded execution (Java variant of mutex lock). However, since the synchronization variables are different (`t0` and `t1`), concurrent execution of both sections are allowed. There are two atomicity violations detected for the same shared variable *checksum*, corresponding to $p_a^1 \to p_c^2 \to p_c^1$ and $p_a^2 \to p_c^1 \to p_c^2$ in the Petri net, respectively. Using the procedure presented in Section IV-B, the required control specifications are $m(p_1) + m(p_a^1) + m(p_c^2) + m(p_c^1) \leq 1$, and $m(p_2) + m(p_a^2) + m(p_c^1) + m(p_c^2) \leq 1$, respectively. These specifications include places p_1 and p_2 since they are in the same loop with the violation accesses. Because both specifications relate to the same shared variable, we combine them as discussed in Section IV-B, $m(p_1)+m(p_a^1)+m(p_c^2)+m(p_2) + m(p_a^2) + m(p_c^2) \leq 1$. The control place added by SBPI for this specification is shown as p_{SPEC} in Figure 5b.

```
Thread 1:
synchronized(t0) {
  for(int i=0;i<N;i++) {
    ...
    tmp0= checksum;
     checksum=tmp0+t0.csum;
    ...
  }
}

Thread 2:
synchronized(t1) {
  for(int i=0;i<N;i++) {
    ...
    tmp1= checksum;
     checksum=tmp1+t1.csum;
    ...
  }
}
```

(a) Code snippet



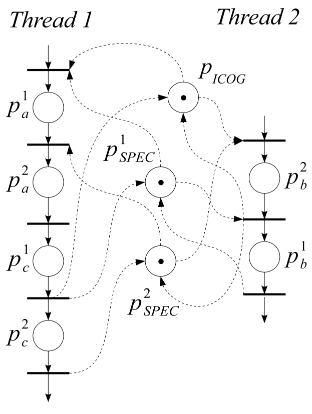
(b) Control synthesis result

Fig. 5: Control synthesis for a violation in RayTracer

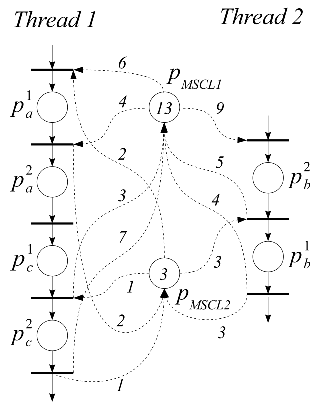
```
Thread 1:                          Thread 2:
synchronized invoke() {
  _in.read(...);                   _ index = ind+length;
  ...                              _ available = ava-length;
  _in.read(...);
}

_in.read(buffer, offset, length) {
  count = (length<=_ available)?length:_ available;
  _ index += length;
}
```

(a) Code snippet



(b) ICOG control synthesis



(c) MSCL control synthesis

Fig. 6: Control synthesis for two violations in OpenJMS

The added control place does not introduce any deadlock. This is verified by running ICOG and observing that no control place is added by it. For this program, applying Algorithm 1 and MSCL directly to the uncontrolled net with the combined specification results in the same control place added.

Our second example, shown in Figure 6, includes two atomicity violations from OpenJMS, which is one of the major providers of Java Message Service API. Function `invoke()` in Figure 6a is a `synchronized` method, which implies single-threaded execution of the function. Assuming the method is atomic, i.e., its entire scope is a transaction, there are two atomicity violations with two different variables, $\_available$ and $\_index$, corresponding to $p_a^1 \rightarrow p_b^1 \rightarrow p_c^1$ and $p_a^2 \rightarrow p_b^2 \rightarrow p_c^2$ in the much simplified Petri net, respectively.

Using the procedure of Section IV-B, control places p_{SPEC}^1 and p_{SPEC}^2 are added to enforce atomicity for $\_available$ and $\_index$, respectively. However, these two control places induce a deadlock, as can be shown by running ICOG. ICOG results in the addition of control place p_{ICOG} to avoid this deadlock. With all three control places, shown in Figure 6b the controlled program enforces atomicity and is deadlock free.

Using Algorithm 1, on the other hand, MSCL returns two linear inequalities, $6m(p_a^1) + 10m(p_a^2) + 10m(p_c^1) + 7m(p_c^2) + 9m(p_b^2) + 4m(p_b^1) \leq 13$ and $2m(p_a^1) + m(p_c^2) + 3m(p_b^1) \leq 3$. Using SBPI, these two inequalities are implemented by control places p_{MSCL1} and p_{MSCL2}, respectively, in Figure 6c. There are multi-weight arcs because of the non-unit coefficients in the linear inequalities. Both control synthesis solutions are maximally permissive with respect to the control specification of atomicity enforcement, and the requirement of liveness.

VI. Conclusion

We have shown how the framework developed in the Gadara project for handling circular-mutex-wait deadlocks in concurrent programs can be adapted and extended to handle another important class of concurrency bugs: single-variable atomicity violations. The class of Gadara nets introduced in [32] is employed to model the program, with suitable modifications that allow to capture atomicity violations in the form of linear inequality constraints. The resulting control problem is to synthesize a controller that enforces these linear inequalities while ensuring liveness of the controlled Gadara net. For this purpose, we have shown how either one of the methodologies in [18], [19] (ICOG) or [24] (MSCL) can be adapted/extended to synthesize a controller that is realized as a set of control places that are added to the Gadara net. These two algorithms embody different tradeoffs between optimality and computational efficiency, but both enforce maximally-permissive control. We have used case studies of real Java programs to illustrate our proposed solution procedure. Our results extend the range of concurrency bugs in software systems that can be handled by techniques from control engineering to include single-variable atomicity violations. Future work will address more general forms of atomicity violations.

References

[1] Gadara project. `http://gadara.eecs.umich.edu/`.

[2] S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, 2010.

[3] H. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. of Programming Language Design and Implementation*, 2008.

[4] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.

[5] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[6] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. ACM Conference on Languages, Compilers and Tools for Embedded Systems*, 2010.

[7] E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, chapter The Mathematics Behind the Banker's Algorithm, pages 308–312. Springer-Verlag, 1982.

[8] C. Dragert, J. Dingel, and K. Rudie. Generation of concurrency control code using discrete-event systems theory. In *Proc. ACM International Symposium on Foundations of Software Engineering*, 2008.

[9] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[10] A. Gamatie, H. Yu, G. Delaval, and E. Rutten. A case study on controller synthesis for data-intensive embedded system. In *Proc. International Conference on Embedded Software and Systems*, 2009.

[11] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[12] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, 2011.

[13] M. V. Iordache and P. J. Antsaklis. Concurrent program synthesis based on supervisory control. In *Proc. 2010 American Control Conference*, pages 3378–3383, 2010.

[14] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proc. of Programming Language Design and Implementation*, 2011.

[15] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[16] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

[17] Z. Li, M. Zhou, and N. Wu. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. Systems, Man, Cybernetics C*, 38(2), 2008.

[18] H. Liao, S. Lafortune, S. A. Reveliotis, Y. Wang, and S. A. Mahlke. Synthesis of maximally-permissive liveness-enforcing control policies for Gadara Petri nets. In *IEEE Conference on Decision and Control*, 2010. Expanded and comprehensive journal version to appear in *IEEE Trans. on Automatic Control*.

[19] H. Liao, J. Stanley, Y. Wang, S. Lafortune, S. A. Reveliotis, and S. A. Mahlke. Deadlock-avoidance control of multithreaded software: An efficient siphon-based algorithm for Gadara Petri nets. In *IEEE Conf on Decision & Control*, 2011.

[20] C. Liu, A. Kondratyev, Y. Watanabe, J. Desel, and A. Sangiovanni-Vincentelli. Schedulability analysis of Petri nets based on structural properties. In *Proc. International Conference on Application of Concurrency to System Design*, 2006.

[21] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering*, 2012.

[22] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[23] J. O. Moody and P. J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer, 1998.

[24] A. Nazeem, S. Reveliotis, Y. Wang, and S. Lafortune. Designing compact and maximally permissive deadlock avoidance policies for complex resource allocation systems through classification theory: The linear case. *IEEE Transactions on Automatic Control*, 56(8):1818 – 1833, aug. 2011.

[25] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., 1986.

[26] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), 1987.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 15(4):391–411, Nov. 1997.

[28] O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, 2011.

[29] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proc. of Symposium on Principles of Programming Languages*, 2006.

[30] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, 2008.

[31] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Proc. of Symposium on Principles of Programming Languages*, 2009.

[32] Y. Wang, H. Liao, S. Reveliotis, T. Kelly, S. Mahlke, and S. Lafortune. Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In *IEEE Conference on Decision and Control*, 2009. Expanded and comprehensive journal version to appear in *Discrete Event Dynamic Systems: Theory & Applications*.