LTL Model Checking of Parallel Programs with Under-Approximated TSO Memory Model

Jiří Barnat, Luboš Brim, and Vojtěch Havel
Faculty of Informatics,
Masaryk University
Brno, Czech Republic
{barnat,brim,xhavel1}@fi.muni.cz

***Abstract*—Model checking of parallel programs under relaxed memory models has been so far limited to the verification of safety properties. Tools have been developed to automatically synthesise correct placement of synchronisation primitives to reinstate the sequential consistency. However, in practice it is not the sequential consistency that is demanded, but the correctness of the program with respect to its specification. In this paper, we introduce a new explicit-state Linear Temporal Logic model checking procedure that allows for full LTL verification of programs under approximated Total Store Order memory model. We also present a workflow of automated procedure to place the synchronisation primitives into the system under inspection to make it satisfy the given specification under the approximated memory model. Our experimental evaluation has been conducted within DiVinE – our parallel and distributed-memory LTL model checker.**

I. Introduction

Model checking [1], [2] as an automated formal verification method, allows for verification of safety and liveness properties of concurrent programs. The model checking procedure requires the description of the concurrent program to be provided in some appropriate modelling language with a clear formal semantics. Unfortunately, most modelling languages were designed to merely mimic the principles of the underlying computation machine, and therefore they lack syntactic and semantic constructs to express low-level details of computation, the memory model of the hardware architecture, for example.

As regards the memory model, existing modelling languages such as the ProMeLa language used in the SPIN model checker [3] or DVE modelling language as used by DiVinE [4], verify concurrent programs under the traditional *Sequential Consistency* (SC) memory model [5]. Under this memory model any write or read to or from a shared variable is instantaneous and immediately visible to all concurrently executed processes. However, this behaviour is not realistic at all. In contemporary hardware architectures, the individual computation cores implement a relaxed memory model, in which the moment of visibility of an update to a shared memory variable may be postponed or even reordered with the visibility times of other updates to different memory locations. Moreover, the precise behaviour in respect to the delayed memory updates differs between the processors of different vendors. To cope with this rather complex situation theoretical models have been introduced to cover as much instances of different relaxed memory behaviour as possible. The most currently used models are the *Total Store Order* (TSO) [6], *Partial Store Order* (PSO) [6], or *x86-TSO* which is a Total Store Order enriched with interlocking instructions [7].

To protect from inconsistencies due to the reordered or delayed memory writes in the relaxed memory model architectures, specific low-level hardware mechanisms were introduced. The most used mechanism is a memory barrier instruction that makes sure that all the changes done prior to the barrier instruction are visible to all the processors before any other instruction listed after the barrier is executed. Other mechanisms include locked and atomic instructions that are used to make small but instant and globally visible updates to memory. Examples of these include hardware supported instructions for compare-and-swap, or fetch-and-add. With these low-level means for memory synchronisation the sequential consistency may be reinstated. Nevertheless, the use of these instructions comes with quite a significant penalty in terms of performance. Therefore, to achieve the correctness and performance at the same time, an appropriate algorithmic tooling is needed to synthesise minimal but correct placement of the necessary synchronisation primitives [7], [8].

When analysing a program under a relaxed memory model, multiple write buffers are used to simulate delayed writes to the shared memory. In theory, no bound may be laid on the size of these buffers. Therefore, if a finite state program is instrumented with the buffering mechanism to mimic, e.g., the TSO relaxed memory model, the number of states in the corresponding state space graph becomes infinite. It has been proved that for such instrumented programs the problem of reachability of a particular system configuration is decidable, while the problem of repeated reachability of a given system configuration is not [9]. As a result, the state space exploration techniques incorporating the full TSO memory mechanics are limited to reachability analysis and have to deal with the unbounded buffers.

In this paper, we suggest a method that allows for model checking of parallel programs running on a relaxed memory model architecture with respect to the properties formalised as Linear Temporal Logic (LTL) formulae. Since the problem of checking validity of an LTL formula on a program with relaxed memory model is generally undecidable, we introduce a mechanism to under-approximate the TSO memory model. Our approach allows for detection of most errors related to relaxed memory model behaviour, and at the same time, it does not require the usage of unbounded buffers. The bound on the size of buffers makes the LTL model checking problem decidable as the state space graph under investigation remains finite. This fact allows the new memory model scheme to be fully incorporated into an existing explicit-state model checkers. In particular, we show how the new memory model can be inserted in a high-level description of the system written in the DVE modelling language — the native language of the DiVinE model checker. This allows us to employ the DiVinE model checker to verify parallel programs under a relaxed memory model architecture against the full LTL specification. Furthermore, we describe a scheme to extend an existing model checker, DiVinE in our case, with the procedure to synthesise a correct placement of synchronisation primitives (atomic/interlocked writes in our case).

We emphasize that our procedure reinstates the correctness of the program under the approximated memory model behaviour with respect to the given LTL specification. Note that the minimal placement of synchronisation primitives may vary depending on the particular LTL formulae that are used to describe the behaviour of the program. Therefore, no existing methods or tools used to reinstate the sequential consistency that does not take the specification of the program into consideration may solve the problem we address.

The main contributions of our approach are as follows. First, to establish the correctness with respect to the LTL specification does not necessary require to use all the synchronisation primitives that would be needed in order to establish the sequential consistency. This results in much better computation performance of the program on a target platform. Second, our approach works with a high-level modelling language which allows the program designer to better understand where and why memory synchronisation primitives must be inserted in order to preserve the correctness of the program. Some other approaches are applicable at much lower level of the program specification (e.g. assembly language) where the position of synchronisation primitives might be obfuscated by compiler optimisations, hence with a limited feedback to the designer. And finally, since our approach to the verification does not require any extensions to the DIVINE modelling language, we are immediately equipped with the Partial Order Reduction technique [10] and distributed-memory processing as provided by the DIVINE model checker to fight the state space explosion problem.

The paper is organised as follows. In Section II we briefly summarise related work on verification of parallel programs under relaxed memory models. In Section III we introduce our new memory model and show how it can be inserted into existing models written in the DVE modelling language. The procedure we use to infer correct placement of synchronisation primitives is described in Section IV. In Section V we list concrete examples demonstrating our new approach and we report on a small case study we did on mutual exclusion protocols. Finally, we conclude the paper and plot some future directions in Section VI.

II. Related Work

The idea of using model checkers to verify programs under weak or relaxed memory models has been discussed first in connection with the explicit-state model checker Murφ [11]. The tool was used to generate all possible outcomes of small assembly language multiprocessor programs in a given memory model [12]. This was achieved by encoding of the memory model and program under analysis in the Murφ description language, which is an idea we intend to follow in this paper.

A particular technique that incorporates TSO store buffers into the model and uses finite automata to represent the possibly infinite set of possible contents of these buffers has been introduced in [13]. Since the state space explosion problem is even worse with TSO buffers incorporated into the model, authors of [13] extended their approach with a partial-order reduction technique later on [7].

A different approach has been taken in [14] where the algorithm to be analysed was transformed into a form where the statements of the algorithm could be reordered according to a particular weak memory ordering. The transformed algorithm was then analysed using model-checking tool, SPIN in that case.

A lot of research has been conducted to actually detect deviation of an execution of the program on a relaxed memory model architecture from an execution with the sequential consistency (SC). An SC deviation run-time monitor using operational semantics [15] of TSO and PSO was introduced in [16], where authors considered a concrete sequentially consistent execution of the program and simulated it on the operational model of TSO and PSO by buffering stores as long as they generated the same trace as the SC execution. Another approach to detect discrepancies between a sequential consistency execution and real executions relied on axiomatic definition of memory models and (SAT-based) bounded model checking [17].

The problem of relaxed memory model computation has been addressed also in program analysis community. Given a finite-state program, a safety specification and a description of the memory model, the framework introduced in [18] computes a set of ordering constraints that guarantee the

```
int x=0, y=0;

process A
{
    state q0, q1, q2; init q0;
    trans
        q0->q1 { effect x = 1; },
        q1->q2 { effect y = 1; };
}

process B
{
    state q0, q1; init q0;
    trans
        q0->q1 { guard x == 1;
                 effect y = y*2; };
}

system async;
```

Figure 1. Parallel program as written in the DIVINE modelling language.

correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution violating the specification. To address the undecidability of the problem, an abstraction from precise memory models has been considered by the BLENDER tool [19]. The tool employs abstract interpretation to deliver an effective verification procedure for programs running under relaxed memory models.

Another program analysis tool, called OFFENCE, was introduced to ensure program stability [8]. The problem of relaxed memory model and correct placement of synchronisation primitives is also relevant for compiler community [20].

During the last decade numerous methods to deal with the problem of correctness of programs under relaxed memory models have been introduced. However, to our best knowledge none of them have been used in order to tune the correctness of the program with respect to liveness properties, LTL formulae in particular.

III. Mimicking Relaxed Memory Model in DVE Models

DVE is a native modelling language of parallel LTL model checker DiVinE. The language presumes that a parallel program to be verified is described as an asynchronous composition of processes that may communicate via possibly buffered communication channels or shared-memory variables. Individual processes are implemented as finite state machines enriched with local variables. For every DVE process, the set of process states, initial state, and the set of transitions are explicitly enumerated. Transitions of processes are decorated with guards – Boolean expressions over local and global variables, effects – assignments to local or global variables, and synchronisation primitives – CSP-like message passing operators. Arithmetic expressions are built from both local and global variables using standard C-like operators for integer arithmetics. An example of a DVE modelled program is given in Figure 1. Supposing that a DVE model has been verified under sequential consistency, our goal is to extend the model so that it mimics the behaviour under relaxed memory model and to check, whether the relaxed memory behaviour violates the specification.

Theoretical models of TSO and PSO allow for a write to a memory location to be arbitrarily delayed with respect to the writes coming from other processors. This is modelled with the help of a buffer data structure that is placed between the processor and the memory bus. When a processor is about to update a value on a concrete memory location, it stores the value (and the location) in the processor-local buffer. The value resides in the buffer until it is flushed out and stored to the main memory. Should a value be read from a particular memory location, the local buffer is first checked for the latest value associated with the location. If an update to the location is found in the buffer, the latest updated value is used for the read operation, in the other case the value stored in the globally visited memory location is used instead. When simulating TSO memory model, all writes coming from a single process proceed to the same shared buffer. Since the buffer is First-In-First-Out structure, writes to two different locations preserve their order. Note that two consecutive writes to the same memory location results in two records to be pushed to the buffer. As a result the number of elements stored in the buffer may in theory be unbounded. In PSO model the effects of two consecutive writes to two different memory locations may actually be seen by another process in the reverse order. This can be effectively achieved if a separate buffer is introduced for every memory location [15], which results in multiple unbounded buffers.

In our approach we avoid the unbounded buffering situation by allowing only a single value per process to be buffered for a particular memory location. In particular, if the second write to the same memory location is issued by a process while the previous update still resides in the buffer, the second write operation waits until the first write is flushed to the main memory. (Note that in this setting the TSO FIFO queuing mechanism forces all write operations issued by the same process prior the first write to be flushed too.) This is inspired by a simple hardware implementation of a relaxed memory buffering mechanism. In practice, Content-Addressable Memory (CAM) modules could be used to implement the delayed writes to the main memory. Every CAM module implements an associative array data structure (a mapping function) that for a given

key (memory location in this case) stores a single value. Under this assumption, the total size of all buffers used by a single process is bound by the size of the main memory. Even though this is not exactly how the TSO memory model works, it is a reasonable approximation of it. The bound on the amount of buffered delayed writes efficiently recovers the decidability of the LTL model checking problem, since the state space of a finite program remains finite even after the relaxed memory mechanism is included in the program.

To extend the description of the program in the DIVINE modelling language with the approximated TSO operational semantics as explained above, we proceed as follows. We enrich every process in the DIVINE model of the program to maintain a temporary variable for every memory location accessed by the process, a validity indicator indicating whether the temporary variable is currently in use or not and a FIFO buffer maintaining the order of writes in which they are executed. Under this setting, a write to a memory location equals to an update to the temporary variable, to the indicator and to the insertion of the memory location identifier into the FIFO buffer. See Figure 2. A read operation either reads the contents of the temporary variable, or the value from the main memory depending on the value of the validity indicator. The program under verification is asynchronously executed in parallel with the so called *memory-model process* that non-deterministically writes the contents of temporary variables to the corresponding memory locations following the order as stored in the FIFO buffer. Every such a write destroys the value of the temporary variable, hence, sets the corresponding validity indicator to false. Initially the validity indicators are set to false.

When extending a DVE model $\mathcal{M}$ with the above described mechanism for mimicking the relaxed memory model behaviour, we proceed as follows. Let $\mathcal{G}$ be the set of all global variables in $\mathcal{M}$, $\mathcal{P}$ be the set of all processes of $\mathcal{M}$ and $\mathcal{T}$ be the set of all transitions of all processes in $\mathcal{P}$. We define a new set of global variables $sb(g,p)$ for all $g \in \mathcal{G}, p \in \mathcal{P}$ that will serve as the temporary store buffers to keep the values of delayed updates to the main memory. For all $g \in \mathcal{G}, p \in \mathcal{P}$ we also define $i(g,p)$ Boolean variables to be the validity indicators. Finally, for each process p we define $|\mathcal{G}+1|$-element array $queue_p$ and two variables $head_p$ and $tail_p$. This triple together is used as a FIFO buffer (implemented as circular buffer) for unique identifiers of global variables. The purpose of $queue_p$ is to keep the order of writes that are delayed in the store buffers. The insertion of the relaxed memory model behaviour proceeds at the level of individual transitions of the DVE model $\mathcal{M}$. Note that every transition t of a DVE model defines two (possibly intersecting) sets of variables: the set $R(t)$ of global variables that are read by the transition t and the set $W(t)$ of global variables that are written by the transition t. During the transformation, each transition of $\mathcal{M}$ is replaced by 2^n transitions where n is the number of read variables, i.e. size of the set $R(t)$. The newly defined transitions differ in the places from where the values of the read variables are taken from. Remember that a variable may be read either from the temporary store buffer or from the main memory.

Formally, the transformation substitutes every transition t of a process p with the set of transitions

$$\{t^A \mid A \subseteq R(t)\},$$

where t^A denotes the original transition t with the following modifications:

1) Guard of t is extended with:
$$\bigwedge_{g \in A} i(g,p) \wedge \bigwedge_{g \notin A} \neg i(g,p)$$
.
2) All occurrences of a variable $g \in A$ in the guard expression and on the right-hand sides of all assignments are substituted with $sb(g,p)$.
3) Every occurrence of any variable g on the left-hand side of an assignment is substituted with $sb(g,p)$, new assignment $i(g,p) = true$ is added to the transition, an unique identifier of the variable g is inserted into $queue_p$ and the whole transition is conditioned by $\neg i(g,p)$. This step is skipped if the write is marked as atomic. In that case, the expression is kept intact, only the condition $\bigwedge_{h \in \mathcal{G}, q \in \mathcal{P}} \neg i(h,q)$ is added to the transition.

The particular role of the enumerated modifications are as follows. Modification *1.* guarantees that t^A is enabled if and only if t is enabled and the set A contains exactly those global variables whose values have been recently updated by the process p so they are still stored in the temporary store buffers of p; *2.* makes all reads from the global variables in A to happen from the corresponding store buffer; and finally *3.* changes locations of all writes to be the temporary store buffers rather than the main memory. An atomic write changes the main memory directly. To ensure that an atomic write is not reordered with some previous writes we require all store buffers of all processes to be empty before writing atomicaly to the main memory. (Note that later on we introduce a mechanism to make selected memory writes instant and atomic.) Note that $queue_p$ can never be full because it contains at most one identifier of each variable at a moment.

Another step in the transformation is the addition of the memory-model process (`process MM`). The memory-model process non-deterministically chooses an occupied temporary store buffer for a variable g and a process p and updates the main memory instance of the variable invalidating at the same time the content of the temporary variable. This update can be performed only when the first element in the $queue_p$ is the identifier of the variable g. After the memory-model process writes the content of the

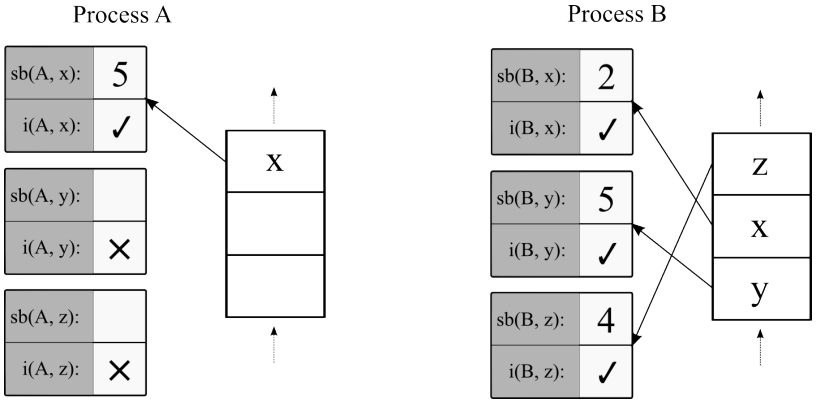

Figure 2. Approximated TSO memory model scheme: $sb(X, Y)$ denotes the auxiliary variable for the process X and memory location Y, $i(X, Y)$ denotes the validity indicator for that variable. Each process is also equipped with a FIFO structure to capture the order of memory updates that were issued by the process, but not yet flushed to the main memory.

temporary store buffer to the variable g, it removes the top element of $queue_p$.

An example of a DVE model and the corresponding transformation is given in Figure 1 and Figure 4, respectively. Note that the transformed model as listed in Figure 4 contains additional assignments to the temporary store buffer variables in the memory-model process. Setting unused store buffer variables to zero when they are invalidated generally helps the model checking procedure to avoid exploration of different but otherwise equivalent states.

To complete the transformation it remains to address the problem of permanent delay of a write to a global variable [6], [21]. In our approach, we let the memory-model process to run asynchronously, which, in the case of cyclic programs, may cause the permanent delay problem. What we are in need of is some fair behaviour of the memory-model process to ensures that every write to a temporary store buffer variable is eventually followed by a memory-model process action that takes the value of the temporary variable and stores it in the main memory. We suggest to address this problem by enriching the specification to be verified by a model checker. In particular, should the model checker verify validity of a formula φ, we let it check for formula $\mathcal{V} \Rightarrow \varphi$, where $\mathcal{V}$ ensures that none of the writes to store buffer is delayed forever. $\mathcal{V}$ is defined as follows:

$$\bigwedge_{g \in \mathcal{G}, p \in \mathcal{P}} \mathbf{GF} \neg i(g, p).$$

The memory model we describe is an under-approximation of TSO memory model. In Figure 3 we give a piece of a parallel program with two processes P and Q, both accessing a global variable x and local variables r and s, to demonstrate a situation that is not covered in the introduced TSO approximation. Suppose that both writes to variable x are performed concurrently. While on TSO memory model compliant architecture the state where $r = 1 \wedge s = 2$ is reachable, it is unreachable in our under-approximated TSO because the double writes to variable y forces all pending writes to complete before copying x to local variable r or s. We could improve our approximation of the TSO memory model by extending the buffers to keep two delayed writes to a single memory location for each process. Then, it would require to perform three consecutive writes to variable y to introduce a run that is not covered by the approximated TSO memory model. Note that this can be generalised for any number of buffered writes.

P	Q
x ← 1	x ← 2
y ← 0	y ← 0
y ← 0	y ← 0
r ← x	s ← x

Figure 3. A piece of a parallel program with two processes P and Q. The state where $r = 2 \wedge s = 1$ is reachable on TSO memory model, but is unreachable under the under-approximated TSO memory model.

IV. Automated Synthesis of Atomic Writes

As mentioned above, DVE modelling language is a high-level modelling language that lacks syntactic and semantic constructs to express hardware-specific commands such as memory fence instruction or atomic memory writes. However, these are the necessary concepts that a user of the model checker must be able to express should we expect of her to verify parallel programs under relaxed memory model behaviour. Since the store buffer semantics is not part of the modelling language, any extension of the languages in the direction of low-level hardware instruction makes no sense. We solved the problem by introducing a parameter to the transformation of DVE model into a DVE model with relaxed memory model behaviour. The parameter is a list of memory writes in the original DVE model that bypass the temporary store buffer variable and directly modify the

```
int x; int y;
int sb_x_A = 0, sb_x_B = 0, sb_y_A = 0, sb_y_B = 0;
int i_x_A = 0, i_x_B = 0, i_y_A = 0, i_y_B = 0;
int queueA[3], queueB[3], tailA, tailB, headA, headB;

process A {
state q0, q1, q2; init q0;
trans
  q0 -> q1 { guard  not i_x_A;
             effect sb_x_A = 1, i_x_A = 1,
                    queueA[tailA] = 0, tailA = (tailA+1)%3; },
  q1 -> q2 { guard  not i_y_A;
             effect sb_y_A = 1, i_y_A = 1,
                    queueA[tailA] = 1, tailA = (tailA+1)%3; };
}

process B {
state q0, q1; init q0;
trans
  q0 -> q1 { guard  not i_x_B && x && not i_y_B;
             effect sb_y_B = 2*y, i_y_B = 1,
                    queueB[tailB] = 1, tailB = (tailB+1)%3; },
  q0 -> q1 { guard  i_x_B && sb_x_B && not i_y_B;
             effect sb_y_B = 2*y, i_y_B = 1,
                    queueB[tailB] = 1, tailB = (tailB+1)%3; };
}

process MM {
state q0; init q0;
trans
  q0 -> q0 { guard  i_x_A == 1 && queueA[headA]==0 && headA != tailA;
             effect x = sb_x_A, i_x_A = 0, sb_x_A = 0,
                    queueA[headA]=0, headA = (headA+1)%3; },
  q0 -> q0 { guard  i_x_B == 1 && queueB[headB]==0 && headB != tailB;
             effect x = sb_x_B, i_x_B = 0, sb_x_B = 0,
                    queueB[headB]=0, headB = (headB+1)%3; },
  q0 -> q0 { guard  i_y_A == 1 && queueA[headA]==1 && headA != tailA;
             effect y = sb_y_A, i_y_A = 0, sb_y_A = 0,
                    queueA[headA]=0, headA = (headA+1)%3; },
  q0 -> q0 { guard  i_y_B == 1 && queueB[headB]==1 && headB != tailB;
             effect y = sb_y_B, i_y_B = 0, sb_y_B = 0,
                    queueB[headB]=0, headB = (headB+1)%3; };
}
system async;
```

Figure 4. The DIVINE modelling language description of the program from Figure 1 instrumented with the approximated TSO memory model mechanism. We omit two transitions of process `B` with unsatisfiable guard (`i_y_B and not i_y_B`) for brevity.

contents of the main memory. See modification rule number *3* from the previous Section.

The overall workflow for automated synthesis of atomic writes is depicted in Figure 5. We start with a `source.dve` file that is supposed to contain a DVE model that is valid under Sequential Consistency memory model. The original model is transformed into a model with store buffers taking into account a list of write operations that should remain atomic (`atomic.txt` file). DiVinE model checker is called to verify the transformed DVE model (`sourceSB.dve`) against the given LTL specification. Either the DVE model is correct under the relaxed memory model behaviour, in which case the whole procedure terminates, or it exhibits invalid execution that is witnessed with a counterexample run as provided by the model checker. If so, we analyse the counterexample automatically to extend the list of memory writes of the original DVE model that must be made atomic. Note that it may take multiple iterations of the model checking loop before we synthesise a list of atomic writes that guarantee satisfaction of the verified LTL formula.

The memory writes to be performed atomically are listed in a separate file. This allows to maintain different atomic-write configurations associated with a single DVE model. It is therefore possible to synthesise different atomic-write configurations for different LTL formulae. The atomic writes are identified by pairs containing the name of the variable to be written atomically and the identifier of the transition that performs the update. Note that we use a syntactic shortcut $(*, t)$ to mark as atomic all variable updates made by transition t. Also note that with this type of identification of atomic writes we may differentiate between two writes to the same variable made from different transitions (lines of code).

At the moment we have no direct way to specify where to put a memory barrier, but this is rather a technical restriction as our way of mimicking the relaxed memory model behaviour can easily simulate memory barrier by requiring all validity indicators to be set to false. Also there is no direct way to explicitly express that a read operation should happen directly from the main memory. However, we can force this type of behaviour of a read instruction indirectly. A value is read from the main memory whenever the corresponding temporary local buffer is empty, which can be achieved if all the immediate preceding writes to the read variable are systematically made atomic.

Since we assume that the original DVE model was valid with respect to the verified LTL formula under the sequential consistency model, any counterexample generated for the modified DVE model must be an exposure of the relaxed memory model behaviour. We employ automatic procedure to detect the so called *hazard intervals* and *inconsistent access* in the counterexample run to derive an automatic update to the list of atomic writes.

A *counterexample* run $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \dots$ of the

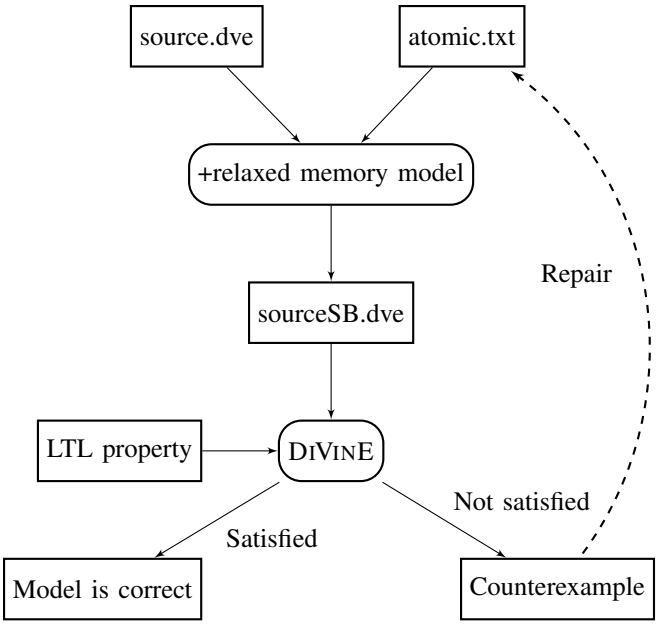

Figure 5. General workflow.

modified model M with respect to the LTL property φ is a lasso-shaped sequence of states and transitions. A *hazard interval* of the counterexample π denoted as $[k, k+l]_X$ is a finite sub-sequence $s_k \xrightarrow{t_k} s_{k+1} \xrightarrow{t_{k+1}} \dots \xrightarrow{t_{k+l-1}} s_{k+l}$ of π such that transition t_k of a process p writes non-atomically a value to the global variable X, and t_{k+l-1} is the first action of the memory-model process among transitions $t_{k+1}, ..., t_{k+l-1}$ that writes the value of X from the store buffer variable $sb(p, X)$ to the main memory. An *inconsistent access* to the hazard interval $[k, k+l]_X$ of counterexample π is a transition t_m, $k < m < k+l$, that reads from or writes to the global variable X and is not part of the same process as transitions t_k.

Hazard intervals precisely describe parts of the counterexample run where a value write is delayed in the temporary store buffer variable, hence invisible for the other processes. However, a hazard interval cannot be the reason for a relaxed-memory-related bug if this hazard remains "hidden", i.e. no other process manipulates (reads from or writes to) the variable within the interval. Therefore, we only search for hazard intervals for which there exists at least one inconsistent access. We consider those intervals as candidates for synthesis of an atomic-write instruction. We choose one such an interval $[k, k+l]_X$ and derive an atomic-write instruction (X, t_k) to be inserted into *atomic.txt*. After that we restart the procedure.

The choice of the inconsistent access to be mended can be crucial with respect to the efficiency (number of iterations and number of atomic-write instructions inferred) of the whole synthesis procedure. For now we use a simple heuristics that prescribes to fix the hazard interval with the

largest number of inconsistent accesses. More elaborated strategy could be used, see e.g. [22].

V. Experimental Validation

In Figure 6 a DVE model of a simple parallel program is presented. We show an example of the iterative algorithm execution for this model and an LTL formula. We begin with empty atomic.txt file. Suppose we want this DVE model to be correct under the relaxed memory model against the LTL formula $\mathcal{F}(x = 2 \ \wedge \ y > 0)$. Note that marking only one (arbitrary) write as atomic is not sufficient for fixing the bug in the code – both writes to the variable x must be performed atomically. In other words, at least two iterations of the algorithm are necessary.

In the first iteration, assume that DiVinE outputs a counterexample where firstly both transitions of the process `A` are executed (i.e. `A` moves to the state `a2` and then to `a3`), then both transitions of the process `B`, and finally memory-model process updates all pending values in the store buffers in an arbitrary order consistent with the TSO ordering on writes. Clearly, this run violates the given formula. There are two inconsistent accesses in this counterexample, namely both actions of the process `B` wrote to the global variables x or y in the situation when a value for each variable had been delayed in corresponding store buffer of the process `A`. As a next step the algorithm chooses one of these two inconsistent accesses for repair. Because none of the relevant hazard intervals was hit by higher number of inconsistent accesses, it is selected at random. Suppose the write to the variable x is chosen, and consequently the pair (x, t), where t denotes the first transition of the process `A`, is added to atomic.txt, that is the last step of the first iteration.

In the second iteration, the model is still not correct with respect to the given formula and a counterexample is obtained. Remember, now the write to the variable x in the process `A` is made directly to the global variable (due to the changed atomic.txt), therefore, in the counterexample the process `A` must write to x after process `B` has done the same, but prior to the final update from $sb(x, B)$ to x. From this we can conclude that the pair (x, t), where t denotes the first transition of process `B`, is one of the possible candidates for repair. If it is chosen, then the next iteration is the final one, because marking both writes to the variable x as atomic is sufficient for satisfying the given formula.

Overall there are three different possible outcomes of this procedure, differing only by the number of added pairs containing the variable y. This non-deterministic behaviour of the algorithm is caused by random choice of inconsistent access and by the fact that a counterexample may not be unique.

We have tested our procedure on three different mutual exclusion algorithms from the BEEM database [23]. In all these algorithms, two processes communicate via shared global variables intensively, therefore, addition of relaxed memory behaviour changes them significantly. See Table I for the numbers of atomised writes needed for the synthesis of a correct model for the given specification.

```
byte x=0;
byte y=0;

process A {
  state a1, a2, a3;
  init a1;
trans
  a1->a2 { effect x=x+1; },
  a2->a3 { effect y=y+1; };
}

process B {
  state b1, b2, b3;
  init b1;
trans
  b1->b2 { effect x=x+1; },
  b2->b3 { effect y=y+1; };
}

system async;
```

Figure 6. Simple DVE model of two processes that manipulate with two shared variables.

Table I
Models of mutual exclusion protocols and the number of writes marked as atomic by the synthesis procedure.

Model	Property	Writes	Atomic
Anderson	$\mathcal{G}(wait \Rightarrow \mathcal{F}granted)$	30	24
Lamport	$\mathcal{G}\neg collision$	27	16
Peterson	$\mathcal{G}\neg collision$	8	4

VI. Conclusions and Future Work

We have introduced a procedure that allows to infer correct placement of memory synchronisation primitives for parallel programs running under relaxed memory model with respect to an LTL specification. Our procedure introduces a new TSO-like memory model that is simulated with a number of one-sized store buffers. While this memory model is not as expressible as the standard TSO or PSO memory models, it is strong enough to capture most relaxed-memory-behaviour errors and can be implemented with some effort within existing high-level modelling languages, which in turn allows usage of standard techniques to fight the state space explosion. The scheme is not optimal in terms of the number of inferred atomic writes nor in the number of model checking runs, but on the other hand it enables sensitive tuning of the placement of atomic writes with respect to the expected behaviour of the program. This sensitive tuning allows for better performance compared to any solution that would reinstate the full sequential consistency.

While the memory model used in our approach is weaker in terms of expressibility than the standard TSO model and the PSO model, the precise classification of the difference yet remains to be done. We would also like to improve methods used to analyse the counterexamples in order to synthesise minimal number of atomic writes in less model checking iterations. Our long-term goal is, however, to tightly integrate the option of relaxed memory model verification into DIVINE model checker, which will require some extensions to the modelling language. We also intend to explore the possibility of application of the scheme to the problem of verification of unmodified C and C++ programs via low-level virtual machine byte-code [24].

REFERENCES

[1] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT press (1999)

[2] Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)

[3] Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, London (2003)

[4] Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010), IEEE (2010) 4–7

[5] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9) (1979) 690–691

[6] SPARC International, Inc., C.: The SPARC architecture manual (version 9). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)

[7] Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: Proceedings of the 18th international SPIN conference on Model checking software, Berlin, Heidelberg, Springer-Verlag (2011) 144–160

[8] Alglave, J., Maranget, L.: Stability in weak memory models. In: Proceedings of the 23rd international conference on Computer aided verification. CAV'11, Berlin, Heidelberg, Springer-Verlag (2011) 50–66

[9] Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '10, New York, NY, USA, ACM (2010) 7–18

[10] Barnat, J., Brim, L., Ročkai, P.: Parallel Partial Order Reduction with Topological Sort Proviso. In: Software Engineering and Formal Methods (SEFM 2010), IEEE Computer Society Press (2010) 222–231

[11] Dill, D.: The Murphi Verification System. In: Computer Aided Verification. Volume 1102 of LLNC., Springer (1996) 390–393

[12] Park, S., Dill, D.: An executable specification and verifier for relaxed memory order. IEEE Trans. on Computers **48**(2) (1999) 227–235

[13] Linden, A., Wolper, P.: An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In: Model Checking Software. Volume 6349 of LNCS., Springer (2010) 212–226

[14] Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). SIGARCH Comput. Archit. News **36** (2009) 65–71

[15] Mador-Haim, S., Alur, R., Martin, M.M.K.: Specifying relaxed memory models for state exploration tools. In: $(EC)^2$: Workshop on Exploting Concurrency Eficiently and Correctly. (2009)

[16] Burnim, J., Sen, K., Stergiou, C.: Sound and Complete Monitoring of Sequential Consistency in Relaxed Memory Models. Technical Report Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley (2010)

[17] Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 107–120

[18] Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: Formal Methods in Computer-Aided Design (FMCAD 2010), IEEE (2010) 111–119

[19] Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Programming language design and implementation (PLDI'11), ACM (2011) 187–198

[20] Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: International Conference on Supercomputing (ICS'03), ACM (2003) 285–294

[21] Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7) (2010) 89–97

[22] Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-Example Guided Fence Insertion under TSO. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012). Volume 7214 of LNCS., Springer (2012) 204–219

[23] Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Model Checking Software (SPIN 2007). Volume 4595 of LNCS., Springer (2007) 263–267

[24] Barnat, J., Brim, L., Ročkai, P.: Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In: NASA Formal Methods Symposium. Volume 7226 of LNCS., Springer (2012) 252–267