MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Relaxed Memory Models in DiVinE

BACHELOR'S THESIS

Vojtěch Havel

Brno, 2012

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vojtěch Havel

Advisor: doc. RNDr. Jiří Barnat, Ph.D.

Acknowledgement

I would like to thank my adviser doc. RNDr. Jiří Barnat, Ph.D. and prof. RNDr. Luboš Brim, CSc. for giving me the opportunity to participate in an interesting research and for numerous discussions on the topic.

Abstract

In this thesis we suggest an extension of the DiVinE model checker which allows verification of models written in the DVE modelling language under a weaker memory model than the Sequential Consistency (SC) memory model. On the top of this extension we devise a synthesis procedure that can compute a set of positions of writes which should be atomised in order to repair a parallel program correct on SC, but not on the weaker memory model.

Keywords

model checking, memory model, TSO, PSO, DVE, DiVinE

Contents

1 Introduction

Ensuring correctness of parallel programs is a challenging task. Besides all kinds of errors well-known from the sequential programming, the parallel environment is a source of new classes of errors (e.g. race conditions). These are often hard to find manually or even with the help of testing, because of the non-determinism, which is caused by the environment, especially by the scheduler. Therefore, formal verification techniques may be very helpful as a part of the development process of a parallel software. One such technique is *model checking* [9]. It is a fully automatic procedure, which decides whether a model satisfies given property specified in some, usually temporal, logic, such as LTL or CTL. Unlike testing, model checking of parallel programs, in some sense, examines all interleavings (or at least all interesting interleavings) of concurrent threads and so it always finds an interleaving leading to an error, if present.

A model for model checking is specified in some modelling language. Variety of modelling languages capable of expressing multiple threads running concurrently are available. However, contemporary processors implement various low-level optimisations, such as caching or store buffering, that can influence the behaviour of a parallel program. Consequently, when verifying a model of a parallel program, it may behave not as intended, because the semantics of a modelling language does not capture this phenomenon called *relaxed* or *weaker memory model*, which is, unfortunately, the case of most modelling languages. Moreover, standard modelling languages lack constructs which can influence relaxed memory model, such as memory barriers or atomic memory operations.

In this thesis, we suggest an extension of the workflow of the DiVinE [7] model checker that allows LTL verification of DVE models under a weaker memory model. An algorithmic procedure to synthesise positions of synchronisation primitives (atomic writes) which guarantees correctness against the desired LTL specification is presented as a part of this extension.

The problem of automatic synchronisation primitives insertion into parallel programs has been intensively studied in recent years. However, standard approach to this problem is to statically analyse the given program and compute a set of positions of synchronisation primitives which is sufficient, in some sense, to restore SC. For example `Offence` tool enriches the given assembly code with lock-based of lock-free synchronisation instructions that guarantees program *stability* to SC [1]. Our approach has the following advantage over tools based on static analysis. It repairs a given parallel program with respect to given LTL specification and thus it can do it more sensitively; it can even keep the program untouched, if it has non-SC executions but is correct against the specification.

This thesis is structured in the following way. In Chapter 2 we shortly describe model checking in general and in DIVINE, and we briefly mention memory models – formal descriptions of the main memory. Main contribution of this thesis is contained in Chapter 3, where an method for verifying DVE models under a weaker memory model is suggested. We evaluate our method in Chapter 4 on three mutual exclusion protocols. And finally, we summarise our work and outline directions of possible future work in Chapter 5.

This work is based on our previous work [4], this thesis is its extended version.

2 Background

2.1 Relaxed memory models

Standard reasoning about parallel program is done on the traditional *Sequential Consistency* (SC) memory model [11]. In this memory model, every write to a shared variable is immediately visible to every other processor. In other words, after a processor writes a value a to a shared variable X, every subsequent read from the variable X obtains a until any other change of X is made. Each execution of a parallel program under SC is equivalent to some sequential execution which is constructed as interleaving of parallel threads.

However, Sequential Consistency memory model does not capture the real behaviour of the most contemporary processors, which implement various "weaker memory" optimisations. Since these optimisations are mainly designed to hide memory latencies, their effect can be characterised by describing the relation between the order of memory operations as executed by processor and the guaranteed order in which are these operations seen by other processors. For example, if arbitrary pair of memory instructions is executed by a program in some order on SC (we will later refer this order of memory operations as *program order*), then the instructions affect the main memory in the same order and all changes are visible for all processors immediately.

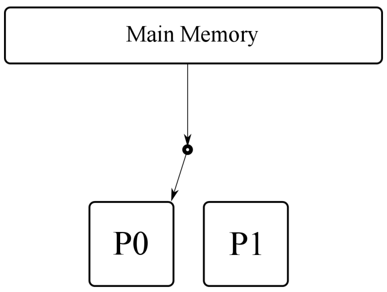

Figure 2.1: Abstract machine implementing the Sequential Consistency memory model.

Unfortunately, processor vendors do not provide precise formal specification of memory model implemented by their hardware and so it is hard to design reliable and usable formal tools working in the area of relaxed memory models. This issue was addressed in [2], where a general framework is suggested for definitions of semantics of memory models in the proof assistant tool called `Coq` [8]. Subsequently, given definition of a memory model defined in such a way, a `diy` tool tries to generate *litmus* tests (programs in pseudo-assembly language) that should, if iterated many times, reveal potential discrepancies in the memory model definition.

Memory models can be partially ordered by the "*allows at least as much executions as*" relation $\preceq$. A pair of memory models (A, B) is in the $\preceq$ relation if for an arbitrary parallel program P and its execution E on memory model B it is possible that P exhibits E on the memory model A.

In this chapter, we give short descriptions of two important memory models, namely the *Partial Store Order* (PSO) memory model and the *Total Store Order* (TSO) memory model and its modified version *x86-TSO*. See [10] for exhaustive list of memory models, their properties and connections to real hardware.

2.1.1 Total Store Order

TSO memory model was firstly defined by SPARC in [15] in terms of ordering of memory operations. SPARC manual defines which pairs of memory operations may be *reordered*. Loads are ordered with respect to earlier loads, the same rule applies for stores, but earlier stores can be reordered with loads from different memory location. The name Total Store Order copies the fact that all processors agree on the same ordering of stores issued by one processor, because stores cannot be reordered. Because it is possible that no instruction reordering occurs, TSO is weaker than SC (TSO $\preceq$ SC). The reverse does not hold, as shows Figure 2.3. This memory model is in [13] extended with the semantics of atomic instructions and memory barriers resulting in the memory model called *x86-TSO*.

TSO/x86-TSO has an elegant operational semantics. Note that in TSO, the only one optimisation is that stores can be delayed until some another store (or atomic instruction or memory barrier) is executed. Therefore, one can imagine hardware implementing TSO as

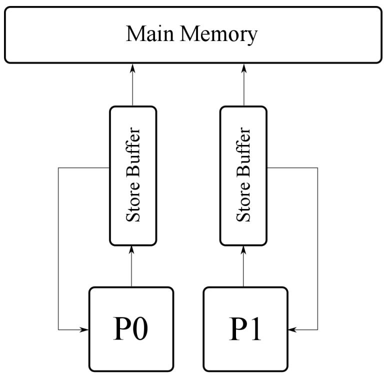

Figure 2.2: Abstract machine with store buffers.

a computational device, which constitutes of processors, the shared main memory and one FIFO buffer called *store buffer* for each processor. The store buffer contains pairs (v, a), where v is a value and a is an address in the main memory. Operational semantics of this abstract machine can be defined in few rules as follows.[13]

- Processor p can read a value v from main memory at address a if none pending value to address a is present in p's store buffer, and the main memory contains v at the address a.
- Processor p can read a value v from its store buffer if the value v is the newest pending value in the store buffer waiting to be written to address a.
- Processor p can write a value v to its store buffer at any time.
- Processor p can silently deque the oldest item (v, a) in its store buffer at any time and write v to the main memory at the position a.
- Processor p can execute the `MFENCE` instruction (the full memory barrier) if its store buffer is empty.

Moreover, in [13] the semantics of the `LOCK` instruction is given. A processor can issue the `LOCK` instruction on a memory location a. This operation results in acquiring a lock on memory location a until the `UNLOCK` instruction is executed by the same processor. While the lock is acquired, other processors cannot operate with the memory location a in the main memory, i.e. different processor cannot read from the memory location a or deque a value from its store buffer to the memory location a. Note that other processors can write a value to the store buffer.

In Figure 2.3 is given a example of a program written in pseudo-assembler that behaves differently on TSO and on SC processor. On TSO, an execution where registers `EAX` of processor 0 and `EBX` of processor 1 hold value 0 at the end, may be observed. On SC, this situation is impossible to happen – at least one processor must hold the value 1 in the register $EA(B)X$.

Note that the definition does not give any a priori bound on the size of the store buffer, so a finite-state program enriched with store buffers can become infinite-state. Moreover, some interesting verification problems become undecidable. [3] shows that reachability for a program with store buffers is decidable (although with non-elementary complexity), but repeated reachability is not. A state is repeatedly reachable if it is reachable from an initial state and it is reachable from itself. Therefore, even LTL formulae of the form $\mathcal{FG}p$ cannot be algorithmically verified.

TSO and x86-TSO are important memory models as they are implemented by the x86 architecture [14].

2.1.2 Partial Store Order

PSO is also defined by SPARC in [15]. PSO is weaker memory model than TSO; besides reordering store-load pairs, PSO architecture allows store-store pairs to different locations to be reordered. It means, that processor which performed multiple stores sees them in the order in which they were executed, but some other processor may see them in different order. This phenomenon is illustrated in the program given in Figure 2.4. Operational definition of PSO is similar to the definition of TSO, but instead of one FIFO store buffer for each process, each process has one FIFO store buffer for each shared memory location. The fact that these store buffers remain FIFO guarantees that stores to one memory location are totally ordered, on the other hand separate store buffers allow values to different memory locations to hit the main memory in different order than in the program

Proc 0	Proc 1
MOV [x] ← 1 MOV EAX ← [y]	MOV [y] ← 1 MOV EBX ← [x]
Forbidden Final State: `Proc 0:EAX=0 ∧ Proc 1:EBX=0`	

Figure 2.3: Litmus test indicating difference between TSO and SC.

Proc 0	Proc 1
MOV [x] ← 1 MOV [y] ← 1	MOV EAX ← [y] MOV EBX ← [x]
Forbidden Final State: `Proc 1:EAX=1 ∧ Proc 1:EBX=0`	

Figure 2.4: Litmus test indicating difference between PSO and TSO.

order.

As in the TSO case, state reachability is non-elementary for PSO and repeated reachability is even undecidable [3]. PSO is in contemporary hardware used rarely; the reason why we include it here is that our memory model to mimic in DVE is similar to the PSO memory model.

As PSO may preserve the order of all store-store pairs, it is weaker than TSO (PSO $\preceq$ TSO). The reverse does not hold as demonstrates Figure 2.4.

2.2 DIVINE model checker

2.2.1 LTL model checking in DIVINE

Definition Given a set of atomic propositions $\mathcal{AP}$, a *Kripke structure* is a 4-tuple (S, T, s_0, L), where

- S is a nonempty set of states
- $T \subseteq S \times S$ is a left-total transition relation (i.e. $\forall x \exists y T(x, y)$)
- $s_0 \in S$ is an initial state
- $L : S \to 2^{\mathcal{AP}}$ is a labelling function

Definition A *run* in a Kripke structure M is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that s_0 is the initial state and for each $i \in \mathbb{N}$ there is a transition between s_i and s_{i+1} (i.e. $(s_i, s_{i+1}) \in T$).

Kripke structure can be used as a low-level description of a system, where each state of a Kripke structure represents one state of a system. Note that by the definition of Kripke structure the set of states may be infinite, DIVINE deals only with finite systems (with finite number of states). Next, we will give a definition of the Linear Temporal Logic (LTL), which is used in the DIVINE model checker for properties specification.

Definition *Syntax* of LTL is described in BNF (*Backus–Naur Form*) by the following equation:

$$\varphi ::= tt \mid ff \mid p \mid \varphi \land \varphi \mid \neg\varphi \mid \mathcal{X}\varphi \mid \varphi\mathcal{U}\varphi,$$

where p belongs to AP.

Definition *Semantics* of LTL is defined over runs of a Kripke structure $M = (S, T, s_0, L)$, given a set of atomic propositions $\mathcal{AP}$. π^i for $\pi = s_0, s_1, \ldots$ denotes the run $s_i, s_{i+1}, \ldots$ and π_0 denotes the first state of the run π. Inductive definition of LTL semantics is as follows.

- $\pi \models tt$ for arbitrary π
- $\pi \not\models ff$ for arbitrary π
- $\pi \models p$ iff $p \in L(\pi_0)$ for $p \in \mathcal{AP}$
- $\pi \models \varphi \land \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$
- $\pi \models \mathcal{X}\varphi$ iff $\pi^1 \models \varphi$
- $\pi \models \varphi\mathcal{U}\psi$ iff $\exists k \geq 0 . \pi^k \models \psi$ and $\forall 0 \leq i < k . \pi^i \models \varphi$
- $\pi \models \neg\varphi$ iff not $\pi \models \varphi$

A few more derived operators are used in LTL, namely the temporal operators $\mathcal{F}\varphi \equiv tt\mathcal{U}\varphi$ and $\mathcal{G}\varphi \equiv \neg\mathcal{F}\neg\varphi$ and Boolean connectives $\lor, \Rightarrow$, defined in the standard way. A Kripke structure M satisfies the property φ if and only if all runs in M satisfy φ.

The DIVINE model checker uses standard automata-based approach to LTL model checking. Negation of the given formula $\neg\varphi$ is translated to a Büchi automaton $A_{\neg\varphi}$ which accepts exactly those runs that satisfy the formula $\neg\varphi$. The question whether the given model M satisfies the property φ is reduced to the question of non-emptiness of the product automaton $A_{\neg\varphi} \times M$. Besides yes or no answer, DIVINE displays in the case of negative outcome a *counterexample*, a lasso-shaped trace which demonstrates an erroneous behaviour of the model, which can be further analysed.

Being explicit model checker, verification in DIVINE suffers from the state space explosion problem. For this reason, it utilises the partial order reduction technique [5] and it is able to use distributed memory.

2.2.2 DVE modelling language

Models for verification in the DIVINE model checker can be specified in several languages, including the Promela or the Murφ modelling language. The native modelling language of DIVINE is the DVE language. A model (called system) in the DVE language is basically asynchronous composition of finite automata extended with local or global (shared) variables. Every variable has finite range defined by its type, which can be `byte`, `int`, or fixed-size array of these types. Each transition can be enriched with *guard* – an arithmetic expression over variables and states of a system, *effects* – assignments to local or global variables and *synchronisation* – a value-passing mechanism between two processes, which is performed in a single step of a system. Two processes passing a value cannot assign to the same variable in effect. DVE models can be parametrised using the *m4* macro language. Examples of parametrised DVE models can be found in the BEEM (*BEnchmark for Explicit Model checkers*) model database [12].

Example of a simple system specified in DVE language is given in Figure 2.5. It consists of two processes `A` and `B`, which may use global variables x and y. Both processes do not define any local variables.

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

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

process B {
  state b1,b2;
  init b1;
  trans
    b1->b2 {guard x; effect y=y*2;};
}

system async;
```

Figure 2.5: Parallel program as written in the DVE modelling language.

3 Relaxed memory model in DVE

3.1 Delayed writes modelling

In this section we describe a way of modelling weaker memory behaviour in DVE. The memory model we consider is similar to PSO, but in our approach we avoid the unbounded buffering situation by allowing only a single value for a particular memory location to be buffered. In particular, if the second write to the same memory location is issued while the previous update still resides in the buffer, the second update destroys the previous one. Under this assumption, the total size of all buffers used by a single process may be bound by the size of the main memory, which efficiently recovers the decidability of the repeated reachability problem.

To extend a DVE model with the relaxed memory behaviour we consider the PSO operational semantics that is restricted with the assumption that a single value is stored for a single memory location. This results in a situation where every DVE process maintains as many different buffers as there are memory locations the process writes to. Moreover, the size of each such a buffer is constant and equals to one. In other words, every process maintains a temporary variable to store the updated value for every memory location accessed by the process and a validity indicator indicating whether the temporary variable is currently in use or not. Under this setting, a write to a memory location equals to an update to the temporary variable and the indicator. 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. Every such a write destroys the value of the temporary variable, hence, sets the corresponding validity indicator to false. Note that due to the asynchronous execution of the program and the memory-model process, two consecutive writes to a single memory location performed by one process may happen according to two different scenarios depending on the concrete interleaving of the writing process and the memory-model process. In the first scenario, the second write is issued before the first one is flushed out of the buffer by the memory process, in which case the first update is lost. In the second scenario, the first update is made globally visible by the memory-model process before the second update is issued resulting in the situation that the first update is temporarily visible to other processes.

To extend a DVE model $\mathcal{M}$ with the above described mechanism for mimicking the relaxed memory 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 Boolean variables $i(g, p)$ that will serve as the validity indicators that are set to true if and only if the store buffer variable $sb(g, p)$ contains a delayed value. The insertion of the relaxed memory 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 substitute 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 $sb(g,p)$ and a new assignment $i(g,p) = true$ is added to the transition, unless this write to variable g is atomised.

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. (Note that later on we introduce a mechanism to make selected memory writes instant and atomic.)

Another step in the transformation is that we add the memory-model process (`process MM`). The memory-model process `MM` nondeterministically 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. An example of a DVE model and the corresponding transformation is given in Figure 2.5 and Figure 3.1, respectively. Note that the transformed model as listed in Figure 3.1 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 [15, 13]. 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 ensure 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, in order to the model checker to verify the 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:

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

3.2 Atomic writes and memory barriers

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 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 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 contents of the main memory and a list of positions of memory barriers, which causes the store buffers to write all stored values to shared variables.

The semantics of atomic writes is reflected in the delayed memory writes modelling in two parts. The first is included in the modification rule number *3* – an atomic write to a global variable g is written directly to g instead of to the store buffer variable $sb(g,p)$. The second part ensures ordering of writes to the same global variable. Any previous write (in program order) to g cannot be performed after the atomic write and therefore we set the validity indicator $i(g,p)$ to zero when executing the atomic write, which models destroying any delayed write to g. Again, when destroying a value in store buffer, we set $sb(g,p)$ to zero in order not to create unwanted duplicated states.

We include a possibility to specify in the `atomic.txt` a place (a transition) in a DVE model where a full memory barrier should be executed. However, memory barriers are not used by the synthesis algorithm, but it can be added manually to the `atomic.txt` file.

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

process A {
state a1,a2,a3; init a1;
trans
  a1 -> a2 {effect sb_y_A = 1, i_y_A = 1;},
  a2 -> a3 {effect sb_x_A = 1, i_x_A = 1;};
}
process B {
state b1,b2; init b1;
trans
  b1 -> b2 {guard x && (not i_y_B) && (not i_x_B);
            effect sb_y_B = y*2, i_y_B = 1;},
  b1 -> b2 {guard x && i_y_B && (not i_x_B);
            effect sb_y_B = sb_y_B*2, i_y_B = 1;},
  b1 -> b2 {guard sb_x_B && i_x_B && (not i_y_B);
            effect sb_y_B = y*2, i_y_B = 1;},
  b1 -> b2 {guard sb_x_B && i_y_B && i_x_B;
            effect sb_y_B = sb_y_B*2, i_y_B = 1;};
}
process MM {
state q0; init q0;
trans
  q0 -> q0 {guard i_x_A == 1; effect x = sb_x_A, i_x_A = 0, sb_x_A = 0;},
  q0 -> q0 {guard i_x_B == 1; effect x = sb_x_B, i_x_B = 0, sb_x_B = 0;},
  q0 -> q0 {guard i_y_A == 1; effect y = sb_y_A, i_y_A = 0, sb_y_A = 0;},
  q0 -> q0 {guard i_y_B == 1; effect y = sb_y_B, i_y_B = 0, sb_y_B = 0;};
}
system async;
```

Figure 3.1: Parallel program from Figure 2.5 with store buffers.

Memory barrier is mimicked in DVE by requiring all validity indicators $i(g, p)$ of the corresponding process p to be set to false. A memory barrier is bound to a transition. When a transition t of process p is set to perform a memory barrier, the guard of t is extended with the condition

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

Note that this causes memory barrier to execute *prior* the effect of the transition t.

The overall workflow for automated synthesis of atomic writes is depicted in Figure 3.2. 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 and places where is a memory barrier (`atomic.txt` file). The 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 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).

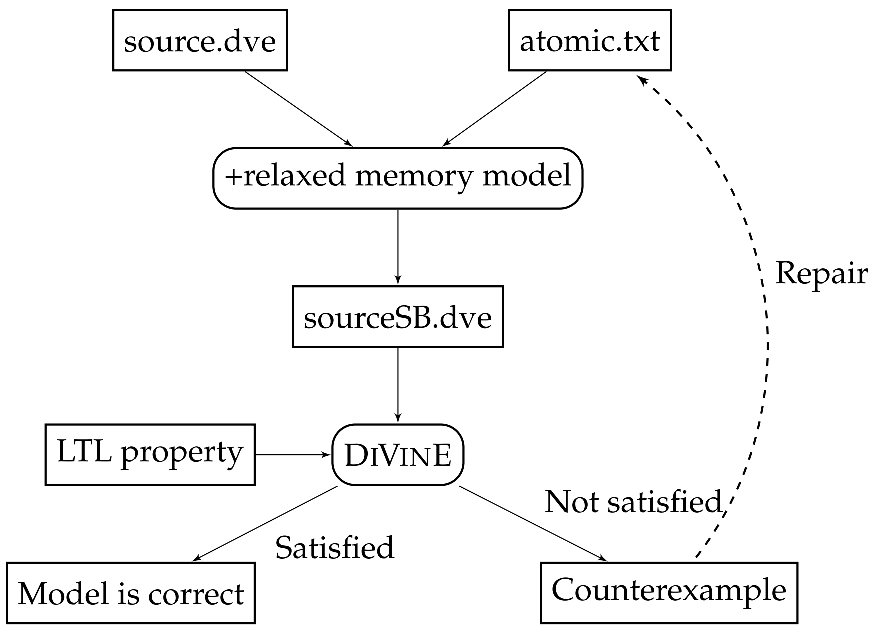

Figure 3.2: General workflow.

3.3 Atomic writes synthesis

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 $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots$ of the 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 $s_k \xrightarrow{t_k} s_{k+1} \xrightarrow{t_{k+1}} \ldots \xrightarrow{t_{k+l-1}} s_{k+l}$ 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 $t_{k+1}, \ldots, t_{k+l-1}$ 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

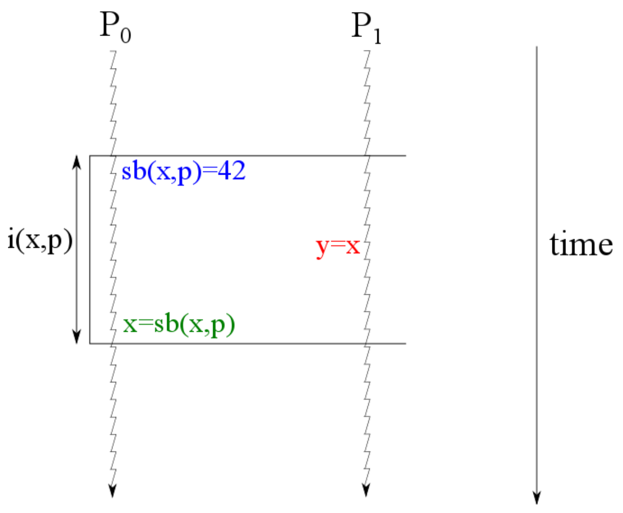

Figure 3.3: Example of an execution with a hazard interval and an inconsistent access. Process P_0 writes non-atomically to the variable x (blue) and after few steps the memory-model process updates the shared variable x with delayed value 42 (green). Between this two events an inconsistent access from process P_1 occurred (red). The write from P_0 to the shared variable x (blue) a candidate for atomising.

the 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 the transition 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) of whole synthesis procedure. For now we use a simple heuristics that prescribes to fix the hazard interval with the largest number of inconsistent accesses.

3.4 Implementation

To allow us to experimentally evaluate properties of suggested methods, we have created a prototypical implementation of both the delayed memory writes modelling procedure and the synthesis procedure as a script in the Python programming language which externally calls the DIVINE tool.

4 Experiments

4.1 Performance impact of atomic writes

To show that the requirement of sensitive synchronisation instructions placement is well motivated we have tested the performance impact of using atomic writes instead of non-atomic writes. We have measured the execution times of simple parallel program written in the C programming language with none, one or both writes to a variable atomically. The program consists of two parallel threads, both execute the function given in Figure 4.1. We used the gcc built-in `__sync_fetch_and_add()` function. Execution times were 3.85 seconds for unmodified program, 12.00 seconds in case of only the first increment of x is atomic and 20.21 seconds for both increments atomic. Therefore, for parallel programs that work with shared variables intensively can the number of atomic writes be crucial.

```
int x,y;

void * work()
{
    for (int i = 0; i < I; i++) {
        x = x+1;
    }
    for (int i = 0; i < I; i++) {
        y = y+1;
    }
}
```

Figure 4.1: Example of a parallel program in C.

4.2 Evaluation on mutual exclusion protocols

We have selected three DVE mutual exclusion protocols from the BEEM database [12] – Peterson's, Anderson's and Lamport's protocol. In all three protocols two or more processes communicate via shared memory quite intensively. We chose LTL properties such that the models satisfy them under SC. In all cases, the property was not satisfied under the relaxed memory model with no memory access atomised. We have counted pairs (x, t) added to the `atomic.txt` file by a run of the synthesis procedure, the results are presented in Table 4.1. Only in the case of the `Peterson.dve` model the number of synthesised atomic writes is optimal (a model with three atomic and no memory barrier cannot satisfy the given specification). Of course, the numbers may be different when repeating experiment.

Some of these models contains shared variables of an array type, and so firstly we had to create manually an equivalent models of these protocols without arrays – instead of array A with k elements, we created new variables $A_0, A_1, \dots, A_{k-1}$ for each element of A. If an element of A is accessed by transition t with dynamically computed index given by expression e, we created set of new transitions $t_0, t_1, \dots, t_{k-1}$ for each possible value of e – a transition t_i is guarded by the condition $e = i$ and instead of accessing the variable $A[e]$ we used the variable A_i.

Table 4.1: The number of writes to global variables atomised by the synthesis procedure.

Model	Property	Writes	Atomic
Anderson	$\mathcal{G}(wait \Rightarrow \mathcal{F} granted)$	30	24
Lamport	$\mathcal{G}\neg collision$	27	13
Peterson	$\mathcal{G}\neg collision$	8	4

4.3 Numbers of states

We have also measured the numbers of states of models enriched with store buffers (see Table 4.2). The results show that the blow-up induced by the store buffering may be significant. If we consider a model composed by p processes, each global (shared) variable introduces p new validity indicators and p new store buffer variables. Therefore, the state space size may increase exponentially with respect to the number of processes and exponentially with respect to the number of global variables. Such exponential growth of the number of states may be problem for even small toy models. For example verifying the `Lamport` model against a short formula satisfied by the model (when building full state space is necessary) causes that DIVINE consumes over 10 gigabytes of memory.

Table 4.2: The number of states of models without store buffers (SC) and with store buffers (SB).

Model	shared variables	states (SC)	states (SB)
Anderson	4	80	380224
Lamport	5	29242	74948667
Peterson	4	196	7267

On the other hand, the state space size decreases with each new item added to the `atomic.txt` file. While executing the synthesis procedure, building full state space of models with no atomic writes may not be necessary in the case when a model does not satisfy a specification under the relaxed memory model, because DIVINE may find a counterexample before it builds full state space. We were able to execute whole synthesis procedure on the `Lamport` model on a machine with only two gigabytes of memory in less than a minute.

5 Conclusion and future work

We have devised an extension of the model checking procedure of the DIVINE tool that allows programmer to verify a model written in the DVE modelling language under a relaxed memory model. The memory model used is similar to the Partial Store Order memory model. To complete the modelling part, we have showed how the memory barrier and atomic write can be expressed in DVE in this setting. On the top of that, we have built a procedure which synthesises a set of writes which, if atomised, is sufficient to repair the given parallel program with respect to the given LTL specification. The synthesis procedure is complete, it always finds a sufficient set of positions of atomic writes. However, it is not optimal in the number of atomised writes and requires multiple model checking invocation. We have implemented these methods as a Python script and we have done few experiments, which show that few naive implementations of classical mutual exclusion algorithms would not work correctly when run on weaker memory model. Next, we have observed an immense growth of the state space size when enriching a model with relaxed memory behaviour.

Aims of possible future research can be divided in two directions. First direction, the synthesis method itself currently uses only simple rules for choosing the write to repair (make atomic). The static analysis may be employed to prune the set of candidate writes for atomising; for now we do not use the structure of the examined program at all. It may improve the synthesis procedure in terms of number of inferred atomic writes and hence it can improve the performance of resulting parallel program.

Second direction, we would like to apply the whole scheme to different input language than DVE. Current development version and the future 3.0 version of the DIVINE tool allow verification of models specified in the LLVM bitcode [6]. As LLVM bitcode can be emitted by compilers clang and gcc (with the dragonegg plugin), it allows to do LTL model checking of unmodified C or C++ programs. When DIVINE creates a state space of the given LLVM bitcode program, the weak memory model of a processor is not reflected, hence the model could be imprecise when comparing to its execution on real hardware. We would like to study the possibility of using the same scheme we have suggested in this thesis on LLVM models instead of DVE models.

Bibliography

[1] J. Alglave and L. Maranget. Stability in weak memory models. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 50–66. Springer-Verlag, 2011.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *CAV*, pages 258–272, 2010.

[3] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 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, pages 7–18. ACM, 2010.

[4] J. Barnat, L. Brim, and V. Havel. LTL Model Checking of Parallel Programs with Relaxed Memory Model.

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

[6] J. Barnat, L. Brim, and P. Ročkai. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods Symposium*, volume 7226 of *LNCS*, pages 252–267. Springer, 2012.

[7] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology*, 9, pages 4–7, 2010.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[10] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, 1995.

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

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

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

[14] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[15] SPARC International. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., 1994.

A Content of the attached archive

- **model_example/** directory contains models of Peterson's, Lamport's and Anderson's mutual exclusion algorithm. Both the original version (`<algorithm>.dve`) and the modified version (`<algorithm>.sb.dve`) are available.

- **tools/** directory contains the following Python scripts

  - **add_SB.py** script enriches the given DVE model with the relaxed memory behaviour and writes the output to standard output. The file describing positions of atomic writes and memory barriers may be supplied. The command for execution is (the second parameter is optional):

    ```
    ./add_SB.py m.dve [atomic.txt]
    ```

  - **repair.py** takes an unmodified DVE model, that enriches with the relaxed memory behaviour and finally runs the synthesis procedure against the LTL property. The resulting set of positions of atomised writes will be written to the file `atomic.txt`.

    The command for execution is (the third parameter is optional):

    ```
    ./repair.py m.dve property.ltl [atomic.txt]
    ```