Using temporal logics for specifying weak memory consistency models

Maximilian Senftleben* and Klaus Schneider

Department of Computer Science,
University of Kaiserslautern, Germany
Email: senftleben@cs.uni-kl.de
Email: schneider@cs.uni-kl.de
*Corresponding author

Abstract: The formal verification of multithreaded programs is not just more difficult due to the concurrent behaviours, but also due to the used underlying weak memory consistency models. Weak memory models arise from techniques like store buffering that were introduced to increase the performance. However, all of these techniques weaken the memory consistency, and may result in unintuitive behaviours where processors may disagree on the order in which write operations occurred. Requirements for verification are therefore unambiguous and complete specifications of such memory consistency models. In the past, specifications based on different formalisms have been presented which often lacked of comparability and the direct usability for model checking. In this paper, we therefore introduce the use of temporal logic to describe the behaviour of memory systems. In particular, we use linear temporal logic (LTL) to define the weak memory models. Thereby, we can easily check the properties of a multithreaded program against several different consistency models and determine the weakest consistency guarantees required to fulfil the given specification.

Keywords: weak memory model; memory model; weak memory consistency; memory consistency; weak consistency; memory specification; specification; temporal logic; model checking; verification; LTL.


Biographical notes: Maximilian Senftleben studied Computer Science from 2008 to 2013 at the University of Kaiserslautern and received his Master's in Computer Science in 2013. He is currently working for his PhD degree as a Research Assistant at the Chair for Embedded Systems at the University of Kaiserslautern. His research interests focus on multiprocessor architectures, weak memory systems and their verification.

Klaus Schneider studied Computer Science from 1987 to 1992 at the University of Karlsruhe in Germany. Between 1992 and 2002, he worked as a Research Assistant at the Institute of Computer Design and Fault Tolerance at the University of Karlsruhe. He received his PhD degree for his research in hardware verification from the Department of Computer Science at the University of Karlsruhe in 1997, and wrote a habilitation thesis in 2001 about the verification of reactive systems. In April 2002, he became a Full Professor at the University of Kaiserslautern and since then, he has headed the Embedded Systems Chair there. His main research interests are innovative design methods.
for the model-based design of embedded systems including their formal verification. To this end, he studies various system level design languages and their models of computation, in particular, synchronous languages and ways to implement these as distributed parallel embedded systems.

This paper is a revised and expanded version of a paper entitled ‘Specifying weak memory consistency with temporal logic’ presented at Verification and Evaluation of Computer and Communication Systems (VECoS), Tunis, Tunisia, 6–7 October 2016.

1 Introduction

In most modern multiprocessor and multi-core architectures, shared memory communication is one of the most significant performance bottlenecks. Memory architectures do not scale well with the number of processors: the more processors compete for memory access, the longer each of them has to wait for getting access to the memory. Therefore, all modern shared memory architectures utilise a multitude of memory optimisations to increase their performance.

Some of these memory optimisations maintain the expected behaviour and therefore do not influence the semantics of programs, e.g., using local caches with cache coherence protocol (Papamarcos and Patel, 1998). In detail, snooping-based cache coherence protocols like the invalidation-based MESI protocol (Papamarcos and Patel, 1998) maintain sequentially consistent (SC) behaviours (Lamport, 1979). SC behaviours are those that could have occurred if the program would have been executed on a single processor by interleaving the executions of the active threads in an arbitrary way.

However, other optimisations like store buffers might lead to unexpected execution results (Weaver and Germond, 1994) that cannot occur in a SC execution. For example, if store operations are buffered in a queue locally visible by a thread, but not by other threads, the threads will disagree on their view on the shared memory as long as store operations are in their local queues. A memory model is called weak if it allows different processors to disagree on the memory operation ordering.

As single-threaded programs still account for a large part of the workload of general purpose processors, weak memory is not noticed most of the time. Multithreaded programs which rely on inter-thread communication, however may lead to different results when executed on a single core or multiple cores.

Figure 1 Two threads P and Q demonstrating weak memory consistency

```java
bool b = false; int v = 0;

thread P() {
    v = value;
    b = true;
}

thread Q() {
    while(!b) {
        r = v;
    }
}
```

For example, consider the two processes P and Q shown in Figure 1 and assume variable $b$ has initially the value false. Process P writes a value to variable $v$ and then signals its completion by setting another variable $b$ to true. Process Q observes variable $b$, and as
soon as it finds a value true in b, it proceeds with reading the value of variable v. Obviously, this program is written with the assumption that write operations of one process (P) are observed in the same order by all other processes (Q). If we relax this restriction in a weak memory model, then it may happen that Q first observes the assignment to b and then proceeds with reading from v even though the assignment to v is not yet visible to Q, resulting in an unexpected behaviour.

Each possible optimisation of a memory system may result in different possible behaviours, and therefore define their own so-called memory consistency model. For example, current multi-core processors maintain for each core a load/store buffer where they buffer their load/store instructions to be issued to the memory system. While these instructions may not yet have been accepted by the memory system, they can already be seen by the processor core that owns the buffer. If even different buffers were used by one core, e.g., for different memory banks, this leads to a weak memory consistency model where assignments executed later by a processor may arrive first in memory.

Figure 2  Hierarchy of weak memory models

Many different optimisations of memory architectures have been implemented so far that lead to different weak memory consistency models. Figure 2 gives an overview of some of these models (see also Mosberger (1993), Adve and Gharachorloo (1996), Steinke and Nutt (2004) and Furbach et al. (2015) for recent surveys). A memory model is thereby called weaker than another one if it allows more possible executions as denoted by the arrows in the figure. For example, all executions that are SC are total store ordering (TSO) consistent as well, but there may exist TSO-consistent executions that are not SC.

Since the presence of weak memory models often leads to unintuitive behaviours, it is strongly recommended to formally verify multithreaded programs taking the underlying weak memory model of the hardware platform into account. To this end, one obviously needs precise formal descriptions of the possible behaviours of each model that can be used for formal verification of the multithreaded programs. However, there is still no commonly accepted formalism to define weak memory models which makes it also very difficult to formally reason about them (see next section). The original descriptions of the weak memory models were unfortunately written only in natural language and are therefore often ambiguous. Recent efforts use formal methods to specify weak memory...
models, as, e.g., higher order logic (Owens et al., 2009) and partial orders (Steinke and Nutt, 2004), which are however both not well-suited for automated model checking.

In this paper, we propose the use of temporal logic as a new way to formally define weak memory models. We emphasise in this paper that the so far mainly used weak memory models can all be conveniently defined by temporal logic. This directly allows us to formally verify temporal logic specifications for a given system with different weak memory models in a flexible way. In particular, we can determine the minimal requirements for a memory system for correctly implementing a multithreaded system.

The outline of the paper is as follows: Section 2 surveys the state of the art in specifying weak memory consistency models. Section 3 gives a short introduction to linear temporal logic (LTL) and explains assumptions and basic properties needed to describe the specifications of the different models in general. Section 4 is the core of the paper that presents the use of LTL for specifying different weak memory models. In Section 5, we present first experimental results, and finally list first conclusions in the final section of the paper.

2 Related work

McKenney (2010) as well as Adve and Gharachorloo (1996) provide recommendable introductions to memory consistency in general. Mosberger (1993) gives a good overview over many of the models known at that time and compares these with each other. Steinke and Nutt (2004) introduced a unified framework that derives weak memory models as the composition of four independent basic properties and defined this way a lattice of memory models. In our previous work, we analysed the complexity of testing whether given execution traces comply with a certain memory model in Furbach et al. (2014, 2015). In most cases, this problem is NP-complete even if some parameters like the number of variables are kept constant. Other unified frameworks were introduced by Adve and Hill (1993), Higham et al. (1998), and Alglave (2012).

The first descriptions of weak memory models were only given in an informal manner and sometimes lead to misinterpretations, e.g., the interpretations of Mosberger (1993) and Ahamad et al. (1993) for pipelined random access memory (PRAM)-consistency which was informally introduced by Lipton and Sandberg (1988) were different: In contrast to the definition by Ahamad et al. (1993), the definition by Mosberger (1993) assumes that a processor first updates its local memory before it broadcasts a write operation to other processors. When manufacturers recognised the impact of weakly consistent systems on programming, they provided distinguishing test cases [so-called litmus tests (Grisenthwaite, 2009, Alglave et al., 2011)]. Litmus tests are either examples of possible weak behaviours or examples of behaviours that may not occur at all (Grisenthwaite, 2009, Alglave et al., 2011).

It became quickly clear that the inherent incompleteness given by sample programs as well as the ambiguity given by informal definitions are inadequate for any kind of formal reasoning about multithreaded programs. Nardelli et al. (2011) discusses the problems that arise from ambiguously defined memory models in modern architectures and high-level languages, and Pugh (2000) revealed that the java memory model was flawed for many years.
It is therefore very important to come up with formally precise, but still comprehensive definitions of memory models. There are already many established ways of specifying the behaviour of different memory systems. In the remainder of this section, we classify the so-far given approaches in different categories.

2.1 Operational definitions

Providing a full system description even at an abstract level would provide an operational semantics of a memory system, but due to the proprietary nature of processors such descriptions are usually not publicly available. Nonetheless, in some cases like publicly available processor cores, the system implementation might be available, and in other cases, an operational semantics can be re-engineered from an available formal definition (Boudol and Petri, 2009, Senftleben, 2013). However, the operational models introduce a big burden for formal verification since in addition to the multithreaded program, one also has to model the underlying hardware platform with its memory transactions for the verification.

2.2 Axiomatic definitions

Another way to specify a weak memory model is to list axioms that have to be satisfied by the allowed behaviours. This has been done first for the SPARC processors (Weaver and Germond, 1994), but would have to be re-engineered for the other models. Axiomatic semantics makes use of quantifiers and often of higher order logic (Owens et al., 2009), and is therefore not well-suited for verification by model checking.

2.3 View-based definitions

View-based definitions describe the ordering of operations seen from the individual processors or the memory system. This type of definition is superior in that it allows one to abstract from the actual implementation and focuses on the processors’ view which simplifies its use for the programmer. Some view-based definitions use the notion of an execution which consists of write and read operations that might occur in that way, more specifically, the read operations are already annotated with the value they will read. A benefit of view-based definitions is that they allow to introduce unified ways of defining different memory models as shown by Steinke and Nutt (2004). This concept can be helpful for analysing specific cases, but may be hard to follow, as it is rather related with litmus tests than with actual programs because in a program, we usually do not know the read value in advance.

3 The general setting

While temporal logics have been proven to be well-suited to describe concurrent reactive behaviours, they have not been employed to describe weak consistent memory behaviour until today. In the following, we introduce a formalism to describe several of these models with linear time logic (LTL). This section provides a short introduction to LTL and explains the general idea of the specification. Furthermore, assumptions and minimal
properties are explained that are required for the specifications in Section 4 where the idea is applied to several memory models.

3.1 Linear temporal logic

LTL is a variant of temporal logic whose models are single execution paths of a system. As LTL is based on a discrete notion of time, each point of time can be denoted as an integer value. The semantics is defined for a labelled transition system (a Kripke structure) $K = (S, I, R, L)$ that consists of a set of states $S$, initial states $I \subseteq S$, a transition relation $R \subseteq S \times S$, and the label function $L$ that maps each state to the variables that hold there. Each path consists of a sequence of states determined by the transition relation $R$. An LTL formula is satisfied for a given structure $K$ if it is satisfied for all infinite paths starting in any of the initial states $I$.

Temporal logic can be defined by different temporal operators. The temporal operators used in this paper are the following ones:

- $G\varphi$ (globally): $\varphi$ holds in the current state and all future states
- $F\varphi$ (finally): $\varphi$ eventually holds at least once (now or in future)
- $X\varphi$ (next): $\varphi$ holds in the next state of the path
- $[\psi \cup \varphi]$ (until): $\varphi$ holds until the first time when $\psi$ holds, and $\psi$ will eventually hold
- $[\psi \cup \varphi]$ (weak until): $\varphi$ holds until the first time when $\psi$ holds, and $\psi$ may never hold (in which case $\varphi$ holds ad infinitum)
- $PF\varphi$ (past finally): $\varphi$ has held at least once (now or in the past).

For more information about temporal logics, see e.g., Emerson (1990), Manna and Pnueli (1992), Baier and Katoen (2008), Clarke et al. (1999) and Schneider (2003).

3.2 Read/write events

In general, a multiprocessor system can be modelled by a set of processors that execute a set of programs connected to a shared memory via a well-defined interface. In the following, we will provide LTL specifications for various memory consistency models. The specifications describe the behaviour of the memory system in a specific environment. The environment consists of multiple processors which interact with the memory system via events. For each processor $p$, we distinguish write events $W_p$ and read events $R_p$. A global unique identifier (an integer value) is assigned to each write event $W_p$ denoted by $Id(W_p)$. We write $W_p^i$ as a shorthand for $W_p \land Id(W_p) = i$. A write instruction which is issued multiple times, e.g., in a loop, will result in different write events with different identifiers. The expression $Loc(W_p)$ denotes the memory location to write to, and $Val(W_p)$ the value to be written. Similarly, $Loc(R_p)$ denotes the memory location to read from, and $Val(R_p)$ the value returned by the memory system. Note that only $Val(R_p)$ is an output of the memory system and everything else is considered as an input. In addition to the mentioned events $W_p$ and $R_p$, the specifications use an additional event $qO$ which models that a processor $q$ has observed a write event. $Prc(qO)$ denotes the originating processor and $Id(qO)$ the identifier of the observed write. For conciseness, we
write $^O_qP$ for $^O_qP \land \text{Pr}c(^O_q) = p \land \text{Id}(^O_q) = i$. A processor is assumed to observe at most one write at a time. Similar to the previous definitions, $\text{Loc}(^O_q)$ denotes the memory location, and $\text{Val}(^O_q)$ the value of the observed write.

### 3.3 Minimal requirements for all models

To provide a well-defined context for the specifications, the environment has to fulfil certain assumptions. First, quite obviously the same event should not occur more than once (1), next the identifiers of the writes should be globally unique (2), and for each processor strictly increasing (3). Furthermore, it is assumed that a processor only issues either a write or a read event at a time (4).

\[
G\left( W^p_i \rightarrow XG\left( \neg W^p_i \right) \right) \quad (1)
\]
\[
G\left( W^q_i \rightarrow \neg \lor_{q \neq p} \{ W^q_j \} \right) \quad (2)
\]
\[
G\left( W^p_i \rightarrow XG\left( \land \neg W^p_j \right) \right) \quad (3)
\]
\[
G\left( \neg (W^p_i \land R^p) \right) \quad (4)
\]

A system has to satisfy at least some basic properties to be considered a reasonable memory system. First, observation events should be causally related to write events, i.e., an observation event $^O_q$ may only occur if there was a corresponding write event $W^q_i$ before (5). Next, a processor should observe each write event only once (6). Naturally, we would like $\text{Loc}(O)$ and $\text{Val}(O)$ to return the same values as their corresponding write event (7).

\[
G\left[ (\neg ^O_q) \cup W^p_i \right] \quad (5)
\]
\[
G\left( ^O_q \rightarrow XG\neg ^O_q \right) \quad (6)
\]
\[
F\left( W^p_i \land \text{Loc}(W^p) = I \land \text{Val}(W^p) = v \right) \rightarrow G\left( ^O_q \rightarrow \text{Loc}(^O_q) = I \land \text{Val}(^O_q) = v \right) \quad (7)
\]

Read operations should either return the default value (in this paper denoted as $\bot$) as long as there was no observed write to that location (8) or the value of the latest observed write event (9).

\[
\left[ (R^y \land \text{Loc}(R^y) = 1) \rightarrow \text{Val}(R^y) = \bot\right] \left( ^O_q \land \text{Loc}(^O_q) = I \right) \quad (8)
\]
\[
G\left( ^O_q \land \text{Val}(^O_q) = v \land \text{Loc}(^O_q) = I \right) \rightarrow \left[ (R^y \rightarrow \text{Val}(R^y) = v) \cup \left( ^O_q \land \text{Loc}(^O_q) = I \land \text{Id}(^O_q) \neq i \right) \right] \quad (9)
\]
4 Specifying weak consistency by temporal logic

In the following, we will introduce the additional properties that are needed to specify some of the known memory models.

4.1 Local consistency

Local consistency was defined by Heddaya and Sinha (1992) as the weakest memory model. It requires each process to observe its own writes in the order they were issued, but allows other processes’ writes to be observed in any order. This correlates to the read-my-writes consistency property known in the database community. This property can be expressed in LTL as follows:

\[ G(W_i^p \rightarrow^p O^p) \]

In other words, a write event requires the issuing processor to immediately observe its own write. Figure 3 shows an example execution for local consistency. It shows that local consistency allows reordering of writes of another processor, i.e., only the issuing processor is required to observe its own writes in order.

Figure 3 TestLocal: local consistent execution which is not slow consistent

4.2 Slow consistency

Slow consistency (Hutto and Ahamad, 1990) was introduced to increase memory performance by reducing consistency maintenance. Slow consistency extends local consistency by requiring processors to observe the writes of another processor to the same location in the order they were issued. The corresponding LTL representation of that property is as follows:

\[ G(O^p \rightarrow XG \land \neg[sO^p \land Loc(sO^p) = Loc(sO^p)]) \]

This means that if a write is observed then no earlier write of that process to the same location may be observed in the future any more. As shown in Figure 4, slow consistency allows reordering of writes if they do not target the same location. In this case, the write to \( Y \) is visible before the second write to \( X \) was observed.

Figure 4 TestsLow: slow consistent execution which is not PRAM consistent
4.3 PRAM consistency

PRAM consistency was first introduced in 1988 by Lipton and Sandberg (1988). PRAM requires each process to respect the order of the writes of other processes, but not their read operations. This means that two writes of the same process will always be observed in the same order by all other processes. Therefore, the PRAM specification extends slow consistency by an additional property:

\[
\left( F(\neg O^p) \right) \rightarrow \bigwedge_{j \neq i} \left[ \left( F(W^j) \rightarrow \left( \neg F(0^p) \cup O^p \right) \right) \right]
\]

That is, if a processor observes a write operation, then it has to observe all earlier writes of that processor beforehand. While the execution in Figure 5 is PRAM consistent, it is not cache consistent (see next paragraph) as cache consistency (CC) requires writes to the same location to be observed in the same order by all processors.

Figure 5  TestPRAM: PRAM consistent execution which is not CC consistent

\begin{align*}
&\text{Write}(X = 1) \quad \text{Write}(X = 2) \\
&\text{Read}(X = 2) \quad \text{Read}(X = 1)
\end{align*}

4.4 Cache consistency

In 1989, Goodman (1991) introduced weak consistency and claimed that it would be the weakest form of consistency. Later on, after shown not to be the weakest model (see Local or Slow consistency), it became known as CC. CC is stronger than slow consistency and extends it by the following property:

\[
\left( F(\neg O^p) \wedge \left[ F(\neg O^p) \wedge \text{Loc}(\neg O^p) = \text{Loc}(\neg O^p) \right] \right) \\
\rightarrow \bigwedge_{j \neq i} \left[ F(W^j) \rightarrow \left( \neg F(\neg O^p) \cup \neg O^p \right) \right]
\]

This implies that if a processor observes two writes to the same location, then if another processor observes them as well they have to be in the same order. In the case of Figure 6, the second processor does not observe all writes of the first processor in the order they were issued and therefore cannot be PRAM consistent, but as the observations are not in conflict with the location-specific ordering of the first processor, the execution is still cache consistent.

Figure 6  TestCC: CC consistent execution which is not PRAM consistent

\begin{align*}
&\text{Write}(X = 1) \quad \text{Write}(X = 2) \\
&\text{Read}(Y = 1) \quad \text{Read}(X = 1) \\
&\text{Write}(Y = 1)
\end{align*}
Using temporal logics for specifying weak memory consistency models

4.5 Sparc TSO consistency

TSO as defined in the SPARC architecture manual (Weaver and Germond, 1994) allows reordering of writes after reads. While writes are buffered, reads are served immediately either from the buffer or by reading from main memory.

While we have not been able to express TSO solely with the events R, W, O, yet, the use of an additional event to express the store buffers behaviour allows us to describe TSO as well. The additional event \( M^p \) models that the write corresponding to \( W^p \) left the store buffer. This allows us to express that a value should be available to other processes. In general \( M^p \) cannot be expressed using \( O^p \) in a straightforward way as it may go unnoticed if the other processes have writes to that location in their buffer as well.

The following extends the specification of CC to express TSO consistency.

\[
G \left[ M^p \rightarrow (\text{PFW}^p) \land \bigwedge_{p \neq q, j} \neg M^j \land \left( G \neg \bigvee_{j < i} M^j \right) \right]
\]

\[
G \left[ \bigwedge_{p \neq q} O^p \rightarrow M^p \right]
\]

\[
G \left[ (W^p \land \text{Loc}(W^p) = l) \rightarrow \left( \bigwedge_{p \neq q} \neg (O^p \land \text{Loc}(O^p) = l) \right) \cup M^p \right]
\]

In detail, writes may only hit the memory once and only in the order they were issued before, furthermore only one write may leave the buffers at a time (10). A write may only be observed by another processor at the time it hits the memory (11). Finally, a write will eventually leave the store buffer and until then the processor may not observe other processor’s writes to that location (12).

Figure 7 TestTSO: TSO consistent execution which is not SC

Further work might show if the specification of TSO actually requires the additional set of events and if so what makes it more difficult to describe than other models.

4.6 Sequential consistency

Sequential consistency was defined by Lamport (1979) and defines a behaviour that may occur if programs are executed on a single processor (core). It requires all processors to agree upon a single sequential total ordering of all write operations they observe. The first required property (totality) can be expressed as:

\[
G \left[ W^p \rightarrow (\text{F}^p O^p) \right]
\]
That means, whenever a write event occurs, then each processor has to observe that write operation some time in the future. The other property to ensure a unique sequential representation is as follows:

\[
\left[ F(\Diamond O^p \land F\Diamond O^q) \right] \rightarrow \bigwedge_{\text{all processors}} \left[ F(\Diamond O^p \land F\Diamond O^q) \right]
\]

This implies that if one processor observes two writes in a specific order, then all other processors have to observe these two writes in the same order.

5 Experimental results

The environment and specifications described in the previous section have been implemented in the input language of SMV (Cimatti et al., 1999). This allows us to utilise NuSMV (Cimatti et al., 1999) and NuXMV (Cavada et al., 2014) for LTL verification either using BDD-based or SAT-based bounded model checking (BMC). For the processor implementation, the environment follows an assembler representation of the programs or test cases to analyse.

5.1 Property verification

To verify a safety property, we abstract from the actual memory system implementation and only provide a memory specification to check the property against all possible behaviours.

Multithreaded programs often come along with parts of the code that requires mutually exclusive access to some variables to achieve the correct behaviour, i.e., they contain a critical section. For example, an instance of Peterson’s algorithm should never allow two processes to access the critical section at the same time. When implementing such programs as parallel modules in SMV and expressing the properties in LTL, we can verify them using LTL model checking. This can be written as follows:

\((\text{Process modules}) \models (\text{Memory Spec}) \rightarrow (\text{Property})\)

Using tools like NuSMV or NuXMV, properties can be verified using BDD approaches or by searching for counterexamples using a SAT-based BMC approach.

5.2 Examples and results

In the following, some algorithms are described and analysed for their minimal required memory models. We start with the well-known mutual exclusion algorithm due to Peterson, followed by a chained computation scenario, and then consider a simple consumer-producer algorithm. First, Peterson’s mutual exclusion algorithm (see Figure 8) will be shown to work as expected with sequential consistency, but fails with weaker consistency models.

Peterson’s mutex algorithm works as follows: Whenever a process wants to enter the critical section, it sets its own flag. Then, it sets the turn variable to the id of the other process. Afterwards, the algorithm checks whether the other process indicated a critical section request with its flag, too. If so, depending on the state of the turn variable, it will
either idle as long as the others process flag holds or it proceeds to the critical section. After a process finished its critical operations, it resets its flag to signal the other process that it is safe to progress. In the example provided, the critical section consists of an increment and an decrement operation of data. Assuming the mutual exclusive execution of the critical sections, the value of data should always be either 0 or 1.

**Figure 8** Peterson mutual exclusion protocol (see online version for colours)

```c
bool flagP, flagQ;
int turn, data;

thread P()
{
    flagP = true;
    turn = 0;
    while(flagQ & turn==0) {};
    // Begin of Critical Section
    data = data + 1;
    data = data - 1;
    // End of Critical Section
    flagP = false;
}

thread Q()
{
    flagQ = true;
    turn = 1;
    while(flagP & turn==1) {};
    // Begin of Critical Section
    data = data + 1;
    data = data - 1;
    // End of Critical Section
    flagQ = false;
}
```

To examine the memory behaviour of that algorithm, it has to be translated in a (pseudo) assembler representation which reveals the individual load-store instructions as shown in Figure 9. The representatin uses memory locations already present in the high-level implementation (see Figure 8): flag[0], flag[1], turn, and data. Consider the non-atomic instructions data = data + 1 and data = data – 1. Assuming the initial value of data is 0, then data should only alternate between 0 and 1 and after both processes are finished, we expect it to be 0. But if the reading and writing part of the instructions are interleaved, data may have more intermediate values –1, 0, 1, 2 and either –1, 0, or –1 in the end. Therefore, the instructions can be used to model a critical section, as it adds unexpected behaviour if the mutual exclusion is not ensured. In this case, we would like to verify that the location data is 0 or 1 all the time.

**Figure 9** Peterson’s algorithm for two processes in an ‘assembler’ representation

```assembly
1  write(flag[id],1)
2  write(turn,1−id)
3  reg = read(flag[1−id])
4  if (reg=0) then goto 7
5  reg = read(turn)
6  if (reg=(1−id)) then goto 3
7  reg = read(data)
8  write(data,reg+1)
9  reg = read(data)
10 write(data,reg−1)
11 write(flag[id],0)
12 goto 12
```

Note: Constant id is 0 for the first process and 1 for the other.
To achieve this verification, we model the instructions depicted in Figure 9 on two processor modules in the SMV input language. Then, we add an LTL specification which reads \((\text{ModelSpecification}) \rightarrow (G(data = 0 \lor data = 1)))\). This means that whenever a path of the state transition system satisfies the specification of the memory model, then it will never be the case that \(data\) is a value other than 0 or 1. Using NuSMV, we proved the safety property to be valid for sequential consistency, and to be invalid for the other models described in this paper: local, slow, CC, and PRAM consistency by providing counterexamples.

Another scenario analysed covers the concept of a chained computation: One process computes a value, another uses that value and a third one collects both the intermediate and the final values. This is depicted in Figure 10: Process \(P\) writes some intermediate value to \(dataP\), signals its availability by setting \(flag\) to 1. Afterwards, process \(Q\) can read that value, calculate some final value, write the result to \(dataQ\) and signal its completion by setting \(flag\) to 2. Last, process \(R\) may read both values and do some post-processing.

![Figure 10](image)

```
int flag, dataP, dataQ = 0; bool done = false;

thread P() {
    // Calculation of vP
    dataP = vP
    flag = 1
}

while(flag != 1) {
    v = dataP
    // Calculation of v
    dataQ = v
    flag = 2
}

thread Q() {
    while(flag != 1) {
        v1 = dataP
    }
}

thread R() {
    while(flag != 2) {
        v2 = dataQ
    }
    done = true;
}
```

Having sequential consistency in mind, process \(R\) is expected to receive the correct final value and the corresponding intermediate value. Relaxing the memory consistency may result in process \(R\) reading wrong values for the intermediate or final result or both, even though \(flag = 2\) was read. Analogously to the first algorithm, using a suitable low-level representation and defining a correctness property like \((\text{ModelSpecification}) \rightarrow (G(done \rightarrow (v_1 = vP \land v_2 = vQ)))\), we were able to prove that the algorithm works as expected for sequential consistency. While we were able to find counterexamples for PRAM, and CC consistency, we could prove that Causal consistency (Lamport, 1978, Steinke and Nutt, 2004) is sufficient to ensure the specified correctness property. This shows that a more efficient, but weakly consistent memory system can be used for this algorithm without compromising the expected behaviour.

![Figure 11](image)

```
bool ready, done = false; int data = 0;

thread P() {
    for(i=0; i<N-1) {
        data = i;
        ready = true;
        while(ready) {
            
        }
    }

thread C() {
    int local[N]; bool done;
    for(i=0; i<N-1) {
        local[i] = data;
        ready = false;
        
    }
    done = true;
}
```
The last algorithm we analyse is a simple producer-consumer algorithm as seen in Figure 11. Producer \( P \) writes one data value and then waits until it has been read by consumer \( C \). \( P \) signals the availability of data by setting variable \( \text{ready} \) to \( \text{true} \) and \( C \) signals that it read the value by setting \( \text{ready} \) to \( \text{false} \) again.

Clearly, the depicted producer-consumer algorithm works as expected for sequential consistency: whenever \( C \) observes \( \text{ready} \) to be \( \text{true} \), the corresponding write to \( \text{data} \) is visible to \( C \), too. Therefore, there is only a single possible outcome for the values of the local registers in a SC environment.

Just like the scenario mentioned before, we were able to prove that the correctness property \( (\text{ModelSpecification}) \rightarrow (\text{G(done} \rightarrow \text{\forall}_{i \geq 1} (\text{local}(i) = i))) \) does not hold for Local, Slow and CC consistency, but in fact holds for sequential consistency as expected but more interestingly for PRAM consistency as well. Following that, costly synchronisation may be saved for such producer-consumer constructs in distributed scenarios (as PRAM is a distributed concept).

Note that in these examples, it is quite easy to determine a suitable depth as both processes terminate after a predetermined number of steps. This approach can be used for repetitive programs as well, but suitable bounds have to be determined based on the number of steps required to cover all relevant behaviour.

Using the described technique, we were able to verify other small litmus-test-alike examples for different memory models, and to disprove them for weaker models. This way, we are able to determine the minimally required consistency models/guarantees to ensure that a given property holds. Verifying against multiple models is as easy as replacing the LTL specification. Neither the processor representation nor the property description have to be changed.

5.3 Other notations

Following our previous publication, we intended to utilise computational tree logic (CTL) to express the specifications hoping that this would allow us to use other tools and techniques which might be less expensive in space. After quite some effort, we were not able to express all properties in CTL and are reasonably confident that some of them are in fact not expressible in CTL at all. While there are more expressive logics used in specification like CTL* or \( \mu \)-calculus, they get harder to grasp and therefore the initial benefit of using LTL is lost.

6 Conclusions and future work

In this paper, we introduced a novel approach to specify weak memory systems using temporal logic. Using temporal logic, we were able to describe the behaviour of different memory consistency models, i.e., restricting the allowed read results in correspondence to the history of issued memory write operations. This itself is already a useful result, as it offers a new perspective and makes the topic more accessible for programmers already familiar with property specifications in LTL. Model checking can directly use our LTL specifications, so that we can use established tools to verify multithreaded programs. Moreover, the approach allows us to easily determine the weakest consistency requirements a program needs to satisfy a given property.
However, the approach suffers from the state explosion problem as weak consistency considers all possible write events and therefore has to quantify over time, processes, variables, and all possible values. Verification with NuSMV's BDD model checking of nontrivial examples like the Peterson mutual exclusion introduced in Figure 8 already requires several GB of memory. Using NuSMV’s BMC with reasonable bounds allows us to inspect more examples, but inevitably will run out of memory for more complex examples, too. While we believe that our representations can be optimised to reduce the amount of required memory, we do not expect that the state explosion problem can be solved in general. Due to the required quantification over time, we expect that the possible improvements will not change the general situation.

In the future, we would like to provide specifications for more models. Furthermore, we are trying to find more efficient specifications, especially to get rid of the global write identifier. In this context, it may be possible to reuse already computed information like the reachable states for model checking when only the memory model is changed. To strengthen the confidence in the presented LTL specifications, we are interested in proving the equivalence between the LTL specification and reference implementations, and furthermore to prove the relationship between different models using our representations (e.g., that sequential consistency implies PRAM consistency).

References


Using temporal logics for specifying weak memory consistency models


