Modelling Memory Consistency Models for Formal Verification

Dissertation

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation

von

Maximilian Senftleben

Datum der wissenschaftlichen Aussprache	07.06.2019
Dekan	Prof. Dr. Stefan Deßloch
Gutachter	Prof.Dr. Klaus Schneider
	Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch

D 386

Abstract

Most modern multiprocessors offer *weak memory* behavior to improve their performance in terms of throughput. They allow the order of memory operations to be observed differently by each processor. This is opposite to the concept of sequential consistency (SC) which enforces a unique sequential view on all operations for all processors. Because most software has been and still is developed with SC in mind, we face a gap between the expected behavior and the actual behavior on modern architectures. The issues described only affect multithreaded software and therefore most programmers might never face them. However, multi-threaded bare metal software like operating systems, embedded software, and real-time software have to consider memory consistency and ensure that the order of memory operations does not yield unexpected results. This software is more critical as general consumer software in terms of consequences, and therefore new methods are needed to ensure their correct behavior.

In general, a memory system is considered weak if it allows behavior that is not possible in a sequential system. For example, in the SPARC processor with total store ordering (TSO) consistency, all writes might be delayed by store buffers before they eventually are processed by the main memory. This allows the issuing process to work with its own written values before other processes observed them (i.e., reading its own value before it leaves the store buffer). Because this behavior is not possible with sequential consistency, TSO is considered to be weaker than SC. Programming in the context of weak memory architectures requires a proper comprehension of how the model deviates from expected sequential behavior. For verification of these programs formal representations are required that cover the weak behavior in order to utilize formal verification tools.

This thesis explores different verification approaches and respectively fitting representations of a multitude of memory models. In a joint effort, we started with the concept of testing memory operation traces in regard of their consistency with different memory consistency models. A memory operation trace is directly derived from a program trace and consists of a sequence of read and write operations for each process. Analyzing the testing problem, we are able to prove that the problem is NP-complete for most memory models. In that process, a satisfiability (SAT) encoding for given problem instances was

developed, that can be used in reachability and robustness analysis.

In order to cover all program executions instead of just a single program trace, additional representations are introduced and explored throughout this thesis. One of the representations introduced is a novel approach to specify a weak memory system using temporal logics. A set of linear temporal logic (LTL) formulas is developed that describes all properties required to restrict possible traces to those consistent to the given memory model. The resulting LTL specifications can directly be used in model checking, e.g., to check safety conditions. Unfortunately, the derived LTL specifications suffer from the state explosion problem: Even small examples, like the Peterson mutual exclusion algorithm, tend to generate huge formulas and require vast amounts of memory for verification. For this reason, it is concluded that using the proposed verification approach these specifications are not well suited for verification of real world software. Nonetheless, they provide comprehensive and formally correct descriptions that might be used elsewhere, e.g., programming or teaching.

Another approach to represent these models are operational semantics. In this thesis, operational semantics of weak memory models are provided in the form of reference machines that are both correct and complete regarding the memory model specification. Operational semantics allow to simulate systems with weak memory models step by step. This provides an elegant way to study the effects that lead to weak consistent behavior, while still providing a basis for formal verification. The operational models are then incorporated in verification tools for multithreaded software. These state space exploration tools proved suitable for verification of multithreaded software in a weak consistent memory environment. However, because not only the memory system but also the processor are expressed as operational semantics, some verification approach will not be feasible due to the large size of the state space.

Finally, to tackle the beforementioned issue, a state transition system for parallel programs is proposed. The transition system is defined by a set of structural operational semantics (SOS) rules and a suitable memory structure that can cover multiple memory models. This allows to influence the state space by use of smart representations and approximation approaches in future work.

Danksagung

Mit Abgabe dieser Arbeit kann ich auf eine lehrreiche und interessante Zeit an der Technischen Universität Kaiserslautern zurückblicken. Es ist erstaunlich wie viele Fragen in der Informatik noch offen sind und wie schnell sich der State-Of-The-Art tatsächlich noch ändert. Nach meinem Masterabschluss ergab sich aus meiner Begeisterung für Prozessorarchitektur die Neugier für mein späteres Forschungsgebiet der schwachen Speicherkonsistenz.

An dieser Stelle möchte ich mich bei all jenen bedanken, die mir bei der Erstellung dieser Arbeit geholfen und mich im Laufe meiner Promotion unterstützt haben. Im Besonderen gilt mein Dank Herrn Prof. Dr. Klaus Schneider, der mir nach interessanten Einblicken im Verlauf meines Studiums ermöglichte eine Promotion anzustreben. Ich möchte ihm danken, dass er mir stets mit Ratschlägen und einem offenem Ohr zur Seite stand.

Für die Zweitbegutachtung möchte ich mich bei Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch bedanken. Für die Leitung der Promotionskommission bedanke ich mich bei Prof. Dr. Pascal Schweitzer.

Weiter gilt mein Dank meinen Kollegen für das angenehme Umfeld und die interessanten Diskussionen, denen so manche Idee entsprang.

Nicht zuletzt möchte ich meiner Familie für die langjährige Unterstützung danken, insbesondere meinem Vater, der mir bereits in jungen Jahren die Wunder der Informationsverarbeitung aufzeigte und so mein Interesse an der Informatik weckte.

Außerdem danke ich meinen Freunden, die in all den Jahren stets für mich da waren und die meiner Begeisterung für meine Forschungsrichtung nie Leid wurden.

Januar 2019, Maximilian Senftleben

Contents

List of Figures

Acronyms

BDD	Binary decision diagram
BMC	Bounded model checking
CAUSAL	Causal consistency
CC	Cache consistency
CTL	Computational tree logic
FIFO	First-in-first-out
GAO	Global anti order
GDO	Global data order
GPO	Global process order
GWO	Global write order
LOCAL	Local consistency
LTL	Linear temporal logic (also: Linear-time temporal logic)
NP	Non-deterministic polynomial time (complexity class)
PC	Processor consistency
PC-D	Processor consistency (as defined by DASH)
PC-G	Processor consistency (as defined by Goodman)
PRAM	Pipelined RAM consistency
PRAM-M	Pipelined RAM (as defined by Mosberger)
PSO	Partial store ordering (by SPARC)
Quartz	Quartz: synchronous system programming language [Schn09]
RMO	Relaxed memory ordering (by SPARC)
SAT	Satisfiability (propositional satisfiability problem)
SC	Sequential consistency
SLOW	Slow consistency
SMV	Symbolic model verifier (tool, input language, and file extension)
SOS	Structural operational semantics
TSO	Total store ordering (by SPARC)

Chapter 1

Introduction

Historically, computer architectures were considered to consist of a single processor that is connected with a single memory via a bus (von Neumann architecture; 1945). The sequentialization of the read and write operations via the single bus ensured that each read operation returns the value most recently written to the corresponding memory location and that we can at all define *the most recently written value*. Even if a processor of this kind of computer architecture would be used to execute multiple processes by interleaving their executions, the memory operations would still take place one after the other and will therefore form a sequence where all memory operations are totally ordered.

Nowadays, essentially all computer architectures consist of multicore processors or even multiple processors, which share a common main memory. Early multiprocessor systems still connected multiple processors via a single bus with the shared memory. This way, processors had to compete for bus access that still enforced an ordering of the memory operations in a linear sequence. Hence, for multiprocessors, communication over shared memory is a performance bottleneck. Modern multiprocessor systems, however, are based on much more complex memory architectures that do not only make use of caches with cache coherence protocols, but also add further local memories to improve their performance. In particular, the use of local store buffers between the processor cores and the caches allows a significantly faster execution. Using store buffers, processors simply 'execute' write operations by putting a pair consisting of an address and the value to be stored at that address in a FIFO buffer. The processor can then continue with the execution of its next instruction and may consult its own store buffer in case a later read operation is executed. The store buffer will execute its write operations as soon as it is given access to the main memory. This avoids idle times due to waiting for the bus access for each write operation and allows a faster execution in general. However, since processors cannot access the store buffers of other processors, they will temporarily have different views on the shared memory. Note that after the store buffers were finally emptied, a coherent view on the shared memory is established. However, before that point of time, the

Procedure p	**Procedure** q
1 x = 1;	1 y = 1;
2 **while** *y=1* **do**	2 **while** *x=1* **do**
3 x = 0;	3 y = 0;
4 Sleep(time);	4 Sleep(time);
5 x = 1;	5 y = 1;
6 **end**	6 **end**
7 Critical Section;	7 Critical Section;
8 x = 0;	8 y = 0;

Figure 1.1.: *Simplified Dekker's mutual exclusion algorithm. [Dijk68b]*

different views that exist due to the contents of the local store buffers allow executions that are otherwise impossible. For this reason, one speaks about weakly consistent memory models that do not impose as strong constraints as the traditional sequential memory model that just interleaved the memory operations of different processes. A weak memory model can be seen as an interface to the programmer, that abstracts from architectural details while providing the essential information to create correct programs.

In essence, weak memory models define additional executions of memory operations that are not possible in a sequential consistency (SC) memory [Lamp79]. Therefore, algorithms that have been developed with SC in mind can have undesirable effects when run on a system with weak memory. In particular, mutual exclusion algorithms and other programs with data races behave incorrectly if the program order is only slightly relaxed.

To illustrate the problems caused by weak consistency, Figure 1.1 shows a simplified version of Dekker's mutual exclusion algorithm (without a token) [Dijk68b]. Both processes claim a resource by setting their variable to 1. If the resource is claimed by the partner, a process releases the resource and waits for some time before it claims the resource again. If the resource is not claimed by the partner, the process enters its critical section and releases the resource afterwards.

Intuitively, the protocol guarantees mutual exclusion. However, if the algorithm is executed on an architecture where the processes buffer their writes, like total store ordering (TSO), it may behave incorrectly (Figure 1.2). Both p and q issue their write operations and put them into their buffers. No write operation was passed to the main memory yet. Since each process can only access its own write buffer, both will still see the initial value of the main memory for each other's variable. This means both processes will read 0 and enter the critical section.

Store buffers are one – but not the only – reason that lead to the introduction of weak memory consistency models [AdGh96; HePa03; Lawr98; StNu04]. For example, in distributed computer systems, the single memory is replaced by multiple distributed memories which can be specific to single processors [LiSa88] or can be shared with some of the other processors (e.g. different

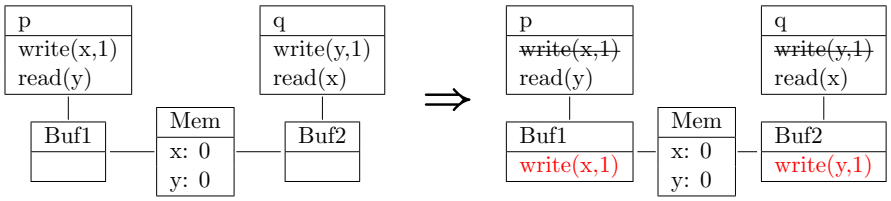

Figure 1.2.: *Weakly consistent behavior of Dekker's algorithm under TSO. Both processes issue their first write which is buffered by their own buffer. Afterwards, both issue a read operation and retrieve the initial value from memory as both buffered values have not yet been written to memory.*

connected sites with multiple processors). Depending on the implemented memory architecture, different weak memory models were developed through the past decades, and some of them may lead to behavior that is quite unexpected for the programmer. It is therefore very important that the designers of modern computer systems are able to describe the potential memory behaviors of their systems in a precise but yet comprehensive way in order for programmers to be able to determine when memory synchronization is required in their programs.

Memory consistency models have been defined in different ways: First descriptions of weak memory models were just given in natural language and were therefore often ambiguous. In fact, such ambiguous descriptions lead to non-equivalent versions of the processor consistency (PC) model [Good91a; ABJK93].

Another way to define a memory consistency model is the so-called view-based approach. In the view-based approach the views, that processes may have during the execution of a multithreaded program, are formally specified. From the viewpoint of a particular process, this usually means that it has to be determined which of the memory operations of other processes have to be interleaved with the memory operations of the own process to define its local view. For example, for pipelined RAM consistency (PRAM) [LiSa88], one would have to consider every write operation of other processes, but not their read operations, while for other memory models other subsets of operations may be considered. The view-based approach can also be defined from the viewpoint of the memory, providing rules for the ordering of all operations as observed by the main memory. View-based definitions are quite popular, and [StNu04] showed how most of the existing weak memory models can be defined in a view-based manner. In [StNu04], Steinke and Nutt even managed to organize many weak memory models in a hierarchy regarding their weakness, and they were able to describe most of the weak memory models systematically as combinations of four basic constraints.

However, the view-based approach remains quite abstract and formal, and while being precise for a formal analysis [FMSS15], it is not comprehensive enough to serve as a general description for programmers. A slightly different approach has been followed by the SPARC memory models TSO and partial

store ordering (PSO) that are described in an axiomatic way [WeGe94]. Also being view-based in principle, these weak memory models were specified by just a few axioms that can be directly used for formal reasoning about the potential executions of a multithreaded program. While also lacking of comprehensiveness, these descriptions are much more compact, and allow one to directly make use of formal verification. The same does not hold if the views are defined by a couple of total or partial orders.

More recent efforts made use of theorem provers to specify weak memory models, using e.g., higher order logic [OwSS09; MMSM12] or temporal logic [SeSc16; SeSc18a] as it will be explained in this thesis. The motivation for this choice is to ensure the well-definedness of the given non-trivial formalization, and to directly reason about properties of the specified memory models with verification tools. However, also these approaches tend to be too difficult to be used as a reference for programmers.

From programming languages, it is well-known that besides the axiomatic and denotational semantics, the operational semantics are often preferred for defining simulators or virtual machines [BrMo17]. Usually, programmers also prefer operational semantics, obviously since that kind of semantics directly determines how the programs are executed. Operational semantics [SeSc18] are well suited to define programming models and as shown in this thesis most memory models can be defined that way.

In the following, the scientific contributions of this thesis are outlined. Then, related work is discussed briefly. The introduction is closed by an outline of the thesis.

1.1. Contributions

In this thesis representations for weakly consistency memory systems are developed and their suitability for verification purposes is explored. Of course, each instance of a given program and a given model could be hand-crafted in different representations. While they might be well optimized, they also require a lot of (probably) repetitive work. In contrast, this thesis proposes representations that are able to cover a multitude of different models. As a consequence, these representations allow to easily compare the influence of different memory models on the same program, e.g. if a program is still working correctly with weakened memory constraints.

A first approach is based on the testing problem. The testing problem asks if a specific execution is possible in a memory model. A joint work [FMSS14; FMSS15] that initially focused on the complexity analysis also resulted in a satisfiability (SAT) encoding for the execution and the memory model. The SAT encoding is composed of different building blocks that can be assembled together based on the memory model in question. While this representation is well suited for use with SAT solvers, it only covers single executions. A program may however take many different paths and result in many different executions. This resulted in further research for representations that are able to model a full program.

Coming from the previous approach using propositional logic, other representations were investigated that are based on predicate logic and its extensions. In that process, a representation using temporal logic, or more precisely linear temporal logic (LTL), is presented. For a given memory model, the LTL specification defines the values that may be returned in response to a processes' read event based on the history of write events. Additional events reflect that a process has observed a write event. In order for a verification tool to reason about a safety property, the program, the specification, and additional state variables have to be encoded in a proper way. To ease that process, a tool was developed that takes a simple multithreaded program written in Quartz, a memory specification and a safety property and returns a SMV file. The resulting SMV file can be used with NuSMV [CCGR99] or NuXMV [CCDG14] for verification either using a BDD-based or SAT-based bounded model checking (BMC) approach. Due to the many state variables the approach struggles with the state space explosion problem.

In an attempt to tackle the mentioned state space problem, an approach is introduced that uses operational semantics to model memory behavior. Using modern system-level languages allows for precise and executable models. This approach has proven to be the best representation for use in teaching so far, as it is the first one that is related to an implementation. While each memory model has to be defined individually, the approach uses as few different core components as possible and reuses them in different models (e.g. FIFO buffers). To cover all possible memory behavior the models have to provide non-deterministic choices and unbounded buffers. Therefore, the models are neither optimized in size nor in execution time, but are complete with respect to the memory model. Further research is aimed at more realistic and optimized implementations which in return are not complete anymore.

The implementation and simulation of the operational semantics are limited by the used language and its tools. In conclusion, a state space simulation is proposed for a minimalistic C-like language. The state space transition system is given as a set of structural operational semantics (SOS) rules. More importantly, the memory state is modelled as a set of queues, one for each memory location. The queues are supplemented by pointers for each process that reflect their observation state. This structure allows to express several different memory models that are similar to cache consistency (CC). The proposed transition system could be used as starting point for future verification tools.

1.2. Related Work

Good introductions to weak memory consistency are given by [McKe10] and [AdGh96]. An early overview of existing memory models was provided by [Mosb93].

The beginning of this research was largely influenced by Steinke and Nutt [StNu04]. They introduced a view-based framework that covered many common memory consistency models and allowed to formally argue about relations

among them. The formalism and some definitions in Chapter 3 are based on their work. Based on this framework, [Mire14; MMSG16] separate the memory models in a distinct classes of convergent and relaxed models in their search on eventual consistency. In contrast to a relaxed model, a convergent model will eventually ensure that all processes see the same memory values if meanwhile no more write operations are issued. Both [AdHi93] and [Algl12] also provide unified memory model formalizations. More work on view-based definitions has been presented in [ABJK93] and [BaBe97a]. The Steinke and Nutt formalism is used instead of the more recent framework of Alglave [Algl10] as it covers more models from the start.

The joint work [FMSS15] of Furbach, Senftleben, Meyer, and Schneider that will be briefly explained in Chapter 3 focused on the complexity analysis of the testing problem. A continuation of the joint work that aims to identify porting bugs when switching memory models can be found in [LFHM17]. The testing problem was first studied in [GiKo97] for SC and linearizability. These results were later extended by [CaLS05a] for TSO, PSO, PC and more.

The inherent incompleteness of sample programs (Litmus tests) as well as the ambiguity given by informal definitions are inadequate for any kind of formal reasoning about multithreaded programs, justifying better formal definitions for memory models. [NSSS11] discusses the problems that arise from ambiguously defined memory models in modern architectures and high-level languages, and [Pugh00] revealed that the Java memory model was flawed for many years.

Similar to Chapter 5, [HiKV98] provides definitions and comparisons of several consistency models and defined machines for the models. It provides a good glimpse at how those machines work and formally justifies the correctness of these models. But, their proposed formalism is not well suited for use in verification, whereas the operational semantics as defined in this thesis are more adequate. Lipton and Sandberg [LiSa88] introduced their PRAM model as operational semantics by defining its structure and communication rules. More recently, the ARMv8 architecture has been described in an operational manner by [FGPS16]. The approach in this thesis is more general, and claims to have the potential to be used to describe most known weak memory models in an operational, and thus comprehensive way.

In [Cali16] a state space exploration tool for TSO is introduced. By using under-approximation and over-approximation to reduce the state space Calin allows for the verification of larger programs. Once more, the approach introduced in Chapter 6 of this thesis is more general as it aims to cover multiple models at once.

In [Lust15] tables are proposed for memory ordering specification that claim to capture all details of a memory model. These *MOST* tables encode the allowed and prohibited reordering between store and load operations and enhance this information with more details, for example the locality, or single- or multi-copy atomicity. Furthermore, it proposes a verification framework to verify a microarchitectural implementation against its specification. However, the specification needs to be given as axioms in an introduced language. The

specification is therefore only as good as the axioms reflect the microarchitecture.

Recently, Alglave et al. [AlCM16] presented the *cat* language that is used to define memory models by constraining the set of program executions to *candidate* executions. While a different formalism is used, the approach introduced in Chapter 4 is quite similar in that it looks at all traces and then only looks and those that match the given specification.

1.3. Outline

In Chapter 2 the fundamentals for this thesis are presented. Starting with the concept of memory models, it is shown how they can be related to each other and examples of their behavior are explained.

Chapter 3 allows for a quick detour to the complexity analysis of the testing problem and explains the construction of a SAT problem corresponding to the given problem instance.

Still using propositional logic but covering a program rather than a single execution the use of temporal logic to describe a weak consistent model is proposed in Chapter 4.

For a more practical and comprehensible representation Chapter 5 introduces operational semantics to represent memory systems with a system description language.

The previous approaches relied on existing tools for verification. To better understand the verification process and optimize it in the future the next Chapter 6 defines a state transition system for a minimal C-like language and a memory representation that can cover different memory models.

Finally, in Chapter 7 the thesis is summarized and ideas for future work are sketched and discussed.

Chapter 2

Background

Contents

2.1. Weak Memory

Using shared memory for multiprocessor communication is a bottleneck for performance of modern systems. Nowadays, processors utilize a variety of techniques to reduce the idle time of processors in case of memory operations. One example of these techniques are write buffers that allow processors to issue a write and immediately continue instead of waiting for the memory to be ready. Another example are distributed shared memories that allow data to be closer to the processor while propagating updates to other processors.

In the past, a multiprocessor system was expected to behave as if processors can only execute in turns. This behavior is based on even older systems where multiple threads were executed on a single processor and therefore only alternating execution was possible resulting in a total order of operations. A memory system that adheres to these principles is called SC consistent.

The introduction of the before-mentioned optimization techniques results in memory systems that are not sequentially consistent anymore. Memory systems that allow more executions than a sequential consistent memory are called weak memories [AdGh96; HePa03; Lawr98; StNu04].

When working with weak memory, it is important to distinguish and classify the considered weak memory models as done in [Algl12; Mosb93; StNu04]. A memory model M_w is called *weaker than* another memory model M_s, denoted by $M_s \preceq M_w$ and indicated by a path in Figure 2.1, if every execution allowed under M_s is also valid under M_w. The hierarchy shows that SC is the strongest and local consistency (LOCAL) is the weakest model. Steinke and Nutt [StNu04] have shown that most weak memory models can be obtained as a combination of four basic models called global anti order (GAO), global write order (GWO), global data order (GDO), and global process order (GPO). To be precise, this applies to SC, PRAM [LiSa88], causal consistency (CAUSAL) [HuAh90], cache consistency (CC) [Good91a], two variants of processor consistency (PC-G, PC-D) [Good91a; ABJK93], slow consistency (SLOW) [HuAh90], and LOCAL [HeSi92]. As a consequence of this characterization via basic models, the memory models form the hierarchy depicted in Figure 2.1.

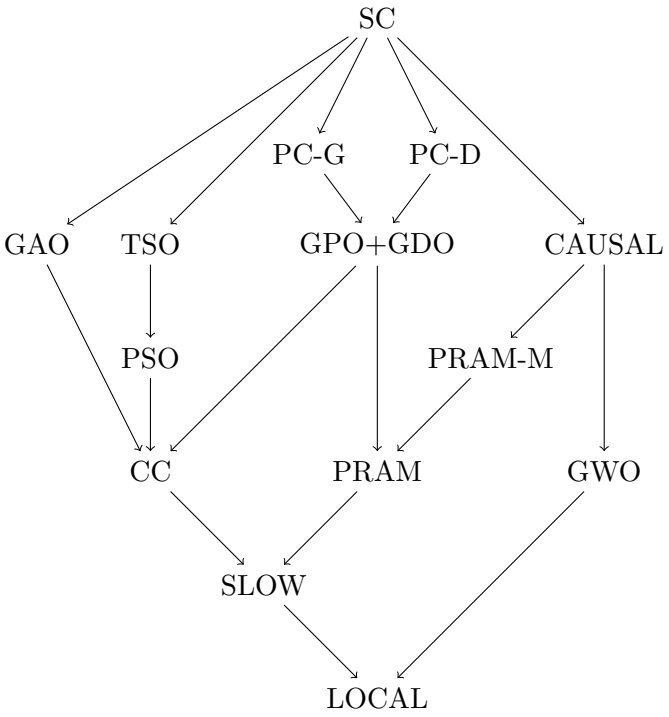

Figure 2.1.: *Hierarchy of weak memory models based on [StNu04]. An arrow indicates that the originating model is stronger than the other.*

There are already several ways to provide definitions of memory consistency models and this thesis will propose some new ones. The first models were described informally, possibly resulting in misinterpretations. The implemen-

tation itself would be the most unquestionable definition, but in most cases these are of proprietary nature and not available to the public. Nonetheless, in some cases operational semantics are available for publicly available processors or can be reverse engineered from available formal definitions [BoPe09]. Unfortunately, most manufacturers only offer partial details about their memory system either formally or by outlining them using litmus tests. Litmus tests are short pieces of software that narrow down behavior that should be expected and restrict behavior that is not possible at all. Some consistency models like the SPARC processors [WeGe94] are given as axiomatic definitions. Axiomatic semantics make use of quantifiers and higher order logic [OwSS09; LCCM06] to reason about the order of memory events in a global manner. In contrast to axiomatic semantics, view-based definitions describe the ordering of operations seen from the individual processors or the memory system. The next section will introduce a formalism to describe memory models in a uniform way. Following that, a couple of view-based definitions for memory models used throughout this thesis are introduced.

2.2. View-Based Formalism

In this section, we will introduce a modified terminology originally used by Steinke and Nutt in [StNu04] to provide formal definitions of weak memory consistency models in a unified way. Note that this formalism describes the multithreaded system in an abstract and formal way. Similar to the concept of litmus tests, it determines the set of possible executions in terms of possible traces of memory operations, but considers the memory system as a black box.

This thesis considers memory systems to only use *read* and *write* operations, i.e. no additional synchronization operations are available. This decision allows to describe more memory models and is justified by the assumption that most software is not (yet) optimized for weak consistent memory and therefore lacks synchronization operations anyway. As the formalism describes the possible executions, both read and write operations already have a determined address and value.

Definition 2.1 ⟨ Memory Operation ⟩
A memory operation is an element in $\mathcal{O} := \mathcal{C} \times \mathcal{V} \times \mathcal{D} \times \mathcal{P} \times \mathbb{N}$.

A operation describes either a read or a write command from $\mathcal{C} := \{r, w\}$ *that reads or writes a data value from* $\mathcal{D}$ *to a variable from* $\mathcal{V}$. *Each operation is assigned its process identifier from* $\mathcal{P}$ *and an issue index. The issue index is a natural number in* $\mathbb{N}$ *that indicates the order of a process's operations.*

Given an operation $o = (c, v, d, p, i)$, the following functions are defined: $\text{cmd}(o) = c$, $\text{var}(o) = v$, $\text{data}(o) = d$, $\text{proc}(o) = p$, $\text{index}(o) = i$ to access the command, the variable, the data value, the process identifier, and the issue index.

Given a set of operations $\mathcal{T} \subset \mathcal{O}$, a subset with shared properties is denoted using a wildcard symbol $*$. For example, the set of operations writing to variable v, $\{o \in \mathcal{T} \mid \text{cmd}(o) = w \wedge \text{var}(o) = v\}$, is denoted by $(*, w, v, *, *)_{\mathcal{T}}$.

A *trace* is a sequence of read and write operations for each process. This sequence characterizes the observed memory events when running multithreaded software. Formally, a *trace* is a set of operations that contains at most one operation per combination of process and issue index, and contains an initial write of initialization process $wmInitProcess$ for each used variable.

Definition 2.2 ⟨ Trace ⟩
A trace *is a subset of operations* $\mathcal{T} \subset \mathcal{O}$ *with:*

- $\forall_{pin\mathcal{P}, i \in \mathbb{N}} \quad \left|\{o \in \mathcal{T} \mid proc(o) = p \wedge \text{index}(o) = i\}\right| \leq 1$
- $\forall_{v \in \mathcal{V}} \quad \left|(*, v, *, *, *)_{\mathcal{T}}\right| \geq 1 \;\Rightarrow\; \left|(w, v, 0, \varepsilon, *)_{\mathcal{T}}\right| = 1$
- $\forall_{o \in (*,*,*,\varepsilon,*)_{\mathcal{T}}} \;\Rightarrow\; cmd(o) = w \wedge \text{data}(o) = 0$

The failing of Dekker's algorithm as shown in Figure 1.1 and Figure 1.2 can be expressed as a trace $\mathcal{T}_{Dekker}$. The trace assumes a violation of the mutual exclusion property. Both processes p and q write the value 1 to their variable but then read the initial value 0 from the other's process variable. Reading 0 both p and q assume to have exclusive access and enter the critical section.

$$\{(w,x,1,p,0),(r,y,0,p,1),(w,y,1,q,0),(r,x,0,q,1),(w,x,0,\varepsilon,0),(r,y,0,\varepsilon,1)\}$$

For better readability traces are presented separated by processes and each process as sequences of operations. For a fixed variable ordering, the initial writes can be implied by the used variables. Therefore, they can be dropped from the representation for simplification. This results in the following notation for the previous example.

$$(w,x,1).(r,y,0) \parallel (w,y,1).(r,x,0)$$

All operations of the initialization process ε are considered to occur before any other operation. For a process p, the local order extends the order in which p's operations are issued by ordering all its operations after the initialization operations.

Definition 2.3 ⟨ Local Order ⟩
The local order *of a process* $p \in \mathcal{P} \setminus \{\varepsilon\}$ *is a relation* $\prec_p : \mathcal{O} \times \mathcal{O}$ *so that*

$$\forall_{o_1, o_2 \in \mathcal{O}} : (o_1 \prec_p o_2) \Longleftrightarrow \big[(\text{proc}(o_1) = \varepsilon \wedge \text{proc}(o_2) = p) \vee (\text{proc}(o_1) = \text{proc}(o_2) = p \wedge \text{index}(o_1) < \text{index}(o_2))\big]$$

Definition 2.4 ⟨ Program Order ⟩
The program order $<^p$ *is the conjunction of all local orders:*

$$<_P = \bigcup_{p \in \mathcal{P}} <_p$$

A trace is considered to reflect the observed memory behavior. To determine the source of these observations we introduce the notion of an *execution* that defines a writes-to relation $\mapsto$ for a given set of operations. The writes-to relation assigns exactly one write operation w to each read operation r which has the same variable and value. Therefore, an execution does not only summarize the observed behavior but also indicates the origin of read values.

Definition 2.5 ⟨ Execution ⟩
An execution *of a trace* $\mathcal{T}$ *is given as the* writes-to *relation* $\mapsto$: $(w, *, *, *, *)_{\mathcal{T}} \times (r, *, *, *, *)_{\mathcal{T}}$ *so that:*

- $\forall_{r \in (r,*,*,*,*)_{\mathcal{T}}} \exists_{w \in (w,*,*,*,*)_{\mathcal{T}}} [w \mapsto r]$
- $[w \mapsto r] \Rightarrow [var(w) = var(r) \wedge \text{data}(w) = \text{data}(r)]$
- $[w_1 \mapsto r] \wedge [w_2 \mapsto r] \rightarrow (w_1 = w_2)$

For small litmus test like traces there often exist few or only a single possible execution.

The before-mentioned trace $\mathcal{T}_{Dekker}$ has only one viable execution as depicted by the dashed arrows in Figure 2.2.

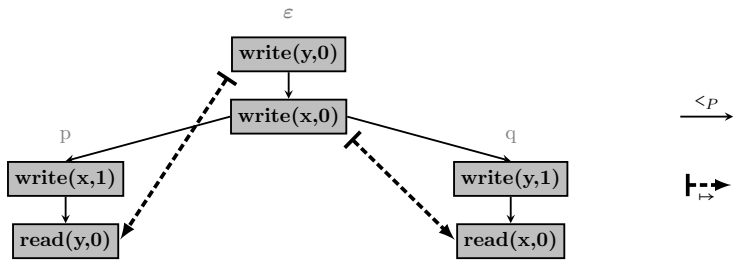

Figure 2.2.: *Example: Execution of the Dekker mutual exclusion example trace* $\mathcal{T}_{Dekker}$*. The initial writes are ordered before all other operations. As there is only a single write for the values read only a single execution exists for the trace.*

In this execution both reads receive their values from the corresponding write of the initialization process ε.

Memory consistency models define if an execution is considered to be *valid* with respect to the given model. To reason about the validity of executions the framework used in this thesis utilizes the concept of *serial views* as shown in [StNu04]. A serial view of an execution resembles the total order in which memory operations are visible to a process. The serial view has to order each read operation r after the write operation w that it reads from according to the given writes-to order $w \mapsto r$ and it has to ensure that no other write operation to the same variable may be ordered in-between r and w. A process may, however not observe all the other operations. Therefore, the serial view is defined on a different subset of the trace for each model. However, all write operations of read operations that are contained in the subset also have to be in the subset. Such a subset is called source-closed and is formally defined as follows: $\forall_{r \in \mathcal{O}'}(w \mapsto r) \rightarrow w \in \mathcal{O}'$. Furthermore, a serial view may have to satisfy additional constraints depending on the memory model.

Recall that total orders are asymmetric and transitive.

Definition 2.6 ⟨ Serial View ⟩
Given an execution $\mapsto$ on a trace $\mathcal{T}$, a strict partial order $< \subset \mathcal{O} \times \mathcal{O}$ and a source-closed subset $\mathcal{O}' \subseteq \mathcal{T}$. A strict total order $<_{sv} \subset \mathcal{O}' \times \mathcal{O}'$ is a serial view *of $\mathcal{O}'$ for execution $\mapsto$ that respects $<$ if is satisfies the following properties:*

- $< \subseteq <_{sv}$
- $\forall_{w,r \in \mathcal{O}'}(w \mapsto r) \rightarrow \big[(w <_{sv} r) \wedge \neg \exists_{w' \in \mathcal{O}'}(var(w) = var(w') \wedge w <_{sv} w' <_{sv} r)\big]$

Abbreviated as $<_{sv}$ is $SerialView(\mapsto, \mathcal{O}', <)$

2.3. Model Definitions

To demonstrate the usage of the introduced formalism to define memory models, definitions of several memory models are introduced, starting with sequential consistency (SC) [Lamp79]. Additionally, exemplary executions are presented to familiarize with the differences and similarities of the models.

2.3.1. Sequential consistency (SC)

While technically not a weak model, we include sequential consistency as defined by Lamport [Lamp79] as a base reference. SC has been the preferred memory model for programmers since it just considers the interleaving of the single thread executions. SC ensures that all processes observe all operations in the same order and in the order in which they were issued. Using Definition 2.6 an execution is valid under SC if there exists a serial view of all operations that respects the program order. The definition of sequential consistency [Lamp79] as expressed by [StNu04] is given in Definition 2.7.

Definition 2.7 ⟨ Sequential consistency (SC) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* sequentially consistent *if*

$$\exists_{<_{sv}} : <_{sv} = SerialView\,(<_P |(*,*,*,*,*)_{\mathcal{T}})$$

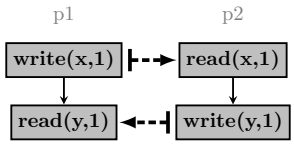

Figure 2.3.: *Example: SC consistent execution.*

Example The execution shown in Figure 2.2 that shows a violation of the mutual exclusion property is not sequentially consistent. The program order puts the initial writes before all other operations and totally orders the operations of one process. Each strict total order for $\mathcal{T}_{Dekker}$ that respects the program order will put one of the writes with value 1 directly after the initial operations. Hence, this write is then between one of the read operations and the corresponding initial write that it has to receive its value from. This violates the second serial view property and demonstrates that no such serial view exists and therefore that the execution is not sequentially consistent. The example in Figure 2.3 however is sequentially consistent as the following serial view exists: $write(x,1).read(x,1).write(y,1).read(y,1)$.

2.3.2. Processor consistency (PC)

In 1989, Goodman defined an intermediate level of consistency that is weaker than SC but stronger than most other consistency models [Good91a].

As there exists another consistency model that slightly differs from Goodman's definition but is also called processor consistency the notation PC-G is used for Goodman's definition. The other model is based on the the Stanford DASH multiprocessor system [LLGW92] and implements a variation of processor consistency which will be referred to as processor consistency (DASH) (PC-D) in this thesis. PC-D was shown to be incomparable to PC-G in [ABJK93; GLLG90].

To prevent misunderstandings, only PC-G will be used within this thesis.

Goodman's definition of processor consistency can be expressed using the introduced formalism as follows [StNu04]:

Definition 2.8 ⟨ Processor consistency (Goodman) (PC-G) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* PC-G consistent *if*

$$\forall_{v\in\mathcal{V}}\exists_{<_v} : <_v = SerialView\left(<_P|\ (*, v, *, *, *)\right)$$
$$\wedge\forall_{p\in\mathcal{P}}\exists_{<_{sv}} : <_{sv} = SerialView\left((\cup_{v\in\mathcal{V}} <_v)\cup <_P|(*, *, *, p, *) \cup (w, *, *, *, *)\right)$$

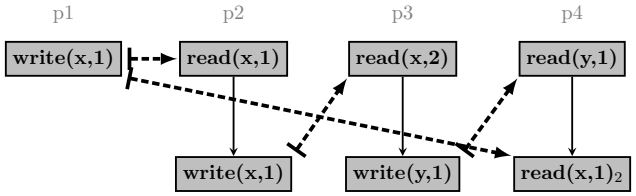

Figure 2.4.: *Example: PC-G consistent execution that is neither GWO nor PSO consistent.*

Example The example in Figure 2.4 is PC-G as the following serial views exist:

$$[x] : write(x,1).read(x,1).read(x,1).write(x,2).read(x,2)$$
$$[y] : write(y,1).read(y,1)$$
$$[p1] : write(x,1).write(x,2).write(y,1)$$
$$[p2] : write(x,1).read(x,1).write(x,2).write(y,1)$$
$$[p3] : write(x,1).write(x,2).read(x,2).write(y,1)$$
$$[p4] : write(x,1).write(y,1).read(y,1).read(x,1)_2.write(x,2)$$

2.3.3. Causal consistency (CAUSAL)

Causal memory is based on potential causality defined by Lamport [Lamp78] which defines a partial order on all memory operations. This partial order orders operations that are causally related. Writes are considered causally related if a write is issued after another write was read by the same process (Write-Read-Write order), or if they are ordered by the program order.

The write-read-write order orders two writes if a read exists which reads from the first write and is issued before the second write by the same processor.

Definition 2.9 ⟨ Write-Read-Write Order ⟩
Two writes are ordered by write-read-write order*,* $w_1 <_{wrw} w_2$*, iff there exists a read* r*, such that* $w_1 \mapsto r <_P w_2$*.*

Causal memory enforces that if a process issues a write operation w after it has read from some write w', then all processes reading w must have observed w' before as well.

Definition 2.10 ⟨ Causal consistency (CAUSAL) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* causal consistent *if*

$$\forall_{p \in \mathcal{P}} \exists_{<_{sv}} : <_{sv} = SerialView\left(<_P \bigcup <_{wrw} | (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}\right)$$

The concept of CAUSAL honors the fact that the preceding write might either influence the newer write or be overwritten by the newer write.

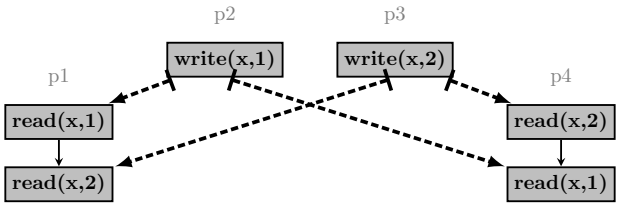

Figure 2.5.: *Example: CAUSAL consistent execution.*

Example The example in Figure 2.5 is CAUSAL consistent as the following serial views exist:

$$[p1] : write(x,1).read(x,1).write(x,2).read(x,2)$$
$$[p2] : write(x,1).write(x,2)$$
$$[p3] : write(x,2).write(x,1)$$
$$[p4] : write(x,2).read(x,2).write(x,1).read(x,1)$$

2.3.4. Pipelined RAM consistency (PRAM)

One of the first well known weak memory models described was PRAM which was presented 1988 by Lipton and Sandberg [LiSa88]. They showed that their shared memory system PRAM scales better than sequentially consistent systems as it is immune to high network latency. Additionally, synchronization costs remain low while performance increases significantly. Due to its informal textual definition, there exists an interpretation of Ahamad et al. [ABJK93], and another slightly different one by Mosberger [Mosb93a] as shown in [Senf13]. PRAM consistency based on Ahamad et al. [ABJK93] can be expressed as shown in Definition 2.11.

Definition 2.11 ⟨ Pipelined RAM consistency (PRAM) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* PRAM consistent *if*

$$\forall_{p \in \mathcal{P}} \exists_{<_{sv}} : <_{sv} = SerialView\,(<_P |(*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}})$$

In a PRAM consistent execution, every process observes all the writes of all other processes in the order they were issued. However, different processes may see the writes of the other processes in a different order. A system implementing PRAM consistency therefore only has to ensure that the communication from one process to another does not reorder or lose writes, while the transmission delay between processors is arbitrary.

Example The execution of $\mathcal{T}_{Dekker}$ in Figure 2.2 is PRAM consistent. For each of the two processes a serial view exists that respects the program order and covers all own operations and all other write operations.

$$[p] : write(y,0).write(x,0).write(x,1).read(y,0).write(y,1)$$
$$[q] : write(y,0).write(x,0).write(y,1).read(x,0).write(x,1)$$

Figure 2.6 shows an execution that is not PRAM consistent. The initial writes have been omitted as they are not required to show that the execution is not valid for PRAM. A serial view for p is of no importance as there are no read operations and therefore any arbitrary interleaving would satisfy the requirements. For process q however there can be no serial view as follows. The first read operation of process q reads from process p's last write. Moreover, the operations have to be kept in program order, therefore a potential serial view $<_{sv}$ would have to satisfy: $(w,x,1,p,0) <_{sv} (w,x,2,p,1) <_{sv} (w,y,1,p,2) <_{sv} (r,y,1,q,0) <_{sv} (r,x,1,q,1)$. In that order however, the second write overwrites the value of the first one before it has been read by the last read operation. And therefore the order violates the serial view property.

2.3.5. Cache consistency (CC)

In 1989, Goodman [Good91a] provided a definition for cache consistency, which he called weak consistency since he assumed that it is the weakest form of memory consistency. Furthermore, he expected that no synchronization guarantees could be given in cache consistency. Meanwhile, both assumptions have been proven wrong by the existence of weaker models and algorithms that can ensure mutual exclusion in weaker models like slow consistency. Definition 2.12 gives a definition of cache consistency [Good91a].

Definition 2.12 ⟨ Cache consistency (CC) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* cache consistent *if*

$$\forall_{v \in \mathcal{V}} \exists_{<_{sv}} : <_{sv} = SerialView\left(<_P | \ (*, v, *, *, *)_{\mathcal{T}}\right)$$

The definition states that each process observes the same ordering on memory operations regarding the same memory location, but processes may see operations regarding different memory locations in different orders.

Example The execution of $\mathcal{T}_{Dekker}$ in Figure 2.2 is cache consistent. For each of the two variables a serial view exists that respects the program order.

$$[x] : write(x,0).read(x,0).write(x,1)$$
$$[y] : write(y,0).read(y,0).write(y,1)$$

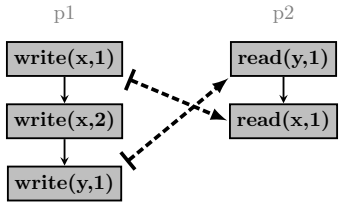

Figure 2.6.: *Example: CC consistent execution that is not PRAM consistent.*

While not PRAM consistent the execution in Figure 2.6 is CC consistent. For each of the two variables a serial view exists that respects the program order.

$$[x] : write(x,0).read(x,1).write(x,2)$$
$$[y] : write(y,1).read(y,1)$$

2.3.6. Slow consistency (SLOW)

Hutto and Ahamad [HuAh90] developed the SLOW model to solve the exclusion and dictionary problems with minimal consistency maintenance. A SLOW consistent memory requires that all write operations to the same variable are observed in the same order by all processes. SLOW consistency can be formally defined as shown in Definition 2.13.

Definition 2.13 ⟨ Slow consistency (SLOW) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* slow consistent *if*

$$\forall_{p \in \mathcal{P}} \forall_{v \in \mathcal{V}} \exists_{<_{sv}} : <_{sv} = SerialView\,(<_P|\,(*, v, *, p, *)_{\mathcal{T}} \cup (w, v, *, *, *)_{\mathcal{T}})$$

Example As SLOW has fewer restrictions than both PRAM and CC, both executions in Figure 2.2 and Figure 2.6 are slow consistent. In fact, it can be shown that every PRAM or CC consistent execution is SLOW consistent as well [StNu04; Senf13].

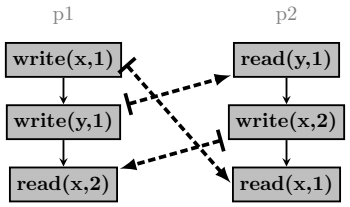

Figure 2.7.: *Example: SLOW consistent execution that is neither CC or PRAM consistent.*

The execution illustrated in Figure 2.7 however is SLOW consistent as well but neither PRAM nor CC consistent.

$$[p, x] : write(x, 1).write(x, 2).read(x, 2)$$
$$[p, y] : write(y, 1)$$
$$[q, x] : write(x, 2).write(x, 1).read(x, 1)$$
$$[q, x] : write(y, 1).read(y, 1)$$

Therefore, SLOW has to be weaker than PRAM and CC, i.e., while every PRAM or CC consistent execution is SLOW consistent there exist executions that are SLOW consistent but neither PRAM nor CC consistent.

2.3.7. Local consistency (LOCAL)

LOCAL was first defined by Heddaya and Sinha [HeSi92] as the weakest constraint that could be required of a shared memory system. In a LOCAL consistent system, each process observes its own operations in local order while all other operations may be observed in an arbitrary order. Different processes' orders are not related at all in this memory model. LOCAL [HeSi92; BaBe97a] can be expressed in the introduced formalism [StNu04] as shown in Definition 2.14.

Definition 2.14 ⟨ Local consistency (LOCAL) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* local consistent *if*

$$\forall_{p \in \mathcal{P}} \exists_{<_{sv}} : <_{sv} = SerialView\left(\prec_p | \ (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}\right)$$

Example LOCAL can be proven to be weaker than SLOW [StNu04; Senf13]. Therefore, all aforementioned executions are valid executions for LOCAL as well. The execution in Figure 2.8 however is only LOCAL consistent:

$$[p] : write(x,1).write(x,2)$$
$$[q] : write(x,2).read(x,2).write(x,1).read(x,1)$$

For the sake of completeness the execution $\mathcal{T}_{NotLocal}$ shown in Figure 2.9 is not LOCAL consistent. Following $(w,x,1,p,0) \prec_p (w,x,2,p,1) \prec_p (r,x,1,p,2)$, the execution violates the only property required for LOCAL: A process has to observe its own operations in the order they are issued. While this behavior seems improbable for processor architectures there exist memory models for information system that do not provide this guarantee, namely *eventual consistency*.

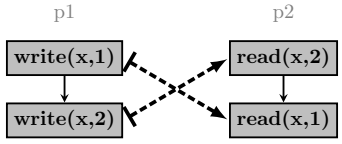

Figure 2.8.: *Example: LOCAL consistent execution that is not SLOW consistent.*

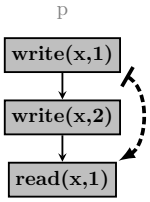

Figure 2.9.: *Example: Execution of* $\mathcal{T}_{NotLocal}$ *that is not local consistent.*

Other models

Plenty of other models exist that are not covered in this introduction. Ultimately, every implementation or optimization of a processor architecture might define another model not described before. For some of them it is difficult to provide a view-based definition. During the research for this thesis, the observation emerged, that whenever a write may not be observed at all by some processes because it is shadowed, i.e., not visible, due to another operation, then a view-based definition gets cumbersome because the shadowed operations have to be explicitly excluded from the serial view. Some of the most prominent examples for such models are SPARC's memory models partial store ordering and total store ordering. In the process of this thesis several attempts have been made to specify them using the introduced formalism. While no proof of the correctness of the resulting definitions can be provided so far, they hold well against attempts to disprove them so far. Therefore, for some models of interest, namely PSO and TSO, their original definition will be provided using the notation of this thesis as well as the proposed view-based definition of TSO for further research.

2.3.8. Total store ordering (TSO)

Total store ordering is the default memory model of SPARC architectures. The SPARC architecture manual [Spar92; WeGe94] states that every implementation has to offer TSO. TSO allows the reordering of stores after loads.

In the following, a simplified axiomatic definition for TSO is given based on the SPARC Architecture manual [Spar92; WeGe94; SiFC92; LCCM06]. 'Simplified' means that the original definition uses 6 axioms, but two of them (Atomicity, Termination) are irrelevant for this consideration. Atomicity only concerns *Swap* operations which are not covered by the introduced formalism which only considers write and read operations. Further, Termination gives a guarantee that a store will eventually be written to the memory and removed from the store buffer.

Definition 2.15 ⟨ Total store ordering (TSO) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* TSO consistent *if a memory order* $\leq$ *exists which respects:*

- *Order:* $\forall_{w_1,w_2 \in (w,*,*,*,*)_{\mathcal{T}}} : w_1 \leq w_2 \oplus w_2 \leq w_1$
- *Value:* $\forall_{r \in (r,v,*,*,*)_{\mathcal{T}}}$:

$$\text{data}(r) = \text{data}\Big(max_{\leq}\left\{w \in (w,v,*,*,*)_{\mathcal{T}} \,|\, w <_P r \vee w \leq r\right\}\Big)$$

- *LoadOp:* $\forall_{r \in (r,*,*,*,*)_{\mathcal{T}}, o \in \mathcal{T}} : (r <_P o) \rightarrow (r \leq o)$
- *StoreStore:* $\forall_{w_1,w_2 \in (w,*,*,*,*)_{\mathcal{T}}} : (w_1 <_P w_2) \rightarrow (w_1 \leq w_2)$

TSO consistency is best explained by describing a possible architecture: Each process has a store buffer which buffers writes before writing them to memory in order. If a process reads a location for which a write exists in its store buffer, then it reads the latest value from the store buffer, otherwise it reads the value from memory.

An execution is TSO if a process observes its own operations in program order and there exists a total order on all write operations which respects program order and is observed by all processes regarding others' writes. Also a process may not observe all writes of other processes as they may be hidden by own writes in its store buffer.

A view-based definition of TSO consistency based on the axiomatic definition is proposed as follows:

Definition 2.16 ⟨ tso order ⟩
Two writes are ordered by tso order, $w_1 <_{tso} w_2$*, iff one of the following properties holds:*

- $w_1 <_P w_2$
- $w_1 <_{wrw} w_2$
- $\text{var}(w_1) = \text{var}(w_2) \wedge \exists_{r1,r2} : w_1 \mapsto r_1 \wedge w_2 \mapsto r_2 \wedge r_1 <_P r_2$
- $\exists_{o'} : o_1 <_{tso} o' <_{tso} o_2$

Definition 2.17 ⟨ TSO (view-based) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* TSO consistent *iff:*

$$\exists_< : < = SerialView\left(<_{tso}|\,(w,*,*,*,*)_{\mathcal{T}}\right)$$
$$\forall_{p\in\mathcal{P}}\exists_{<_{sv}}\ <_{sv} = SerialView\left(< \,|\,(w,*,*,*,*)_{\mathcal{T}} \setminus W^p_{shadowed}\right)$$

with $W^p_{shadowed} \subset (w,*,*,*,*)_{\mathcal{T}} \setminus (w,*,*,p,*)$ *so that*

$$w \in W^p_{shadowed} \rightarrow \bigwedge \neg\exists_{r,\text{proc}(r)=p} : w \mapsto r$$
$$\exists w', \text{var}(w') = \text{var}(w), \text{proc}(w') = p : w < w'$$
$$\wedge \neg\exists_{w'',r} : w < w'' \rightarrow r <_P w'$$

Informally, a write is shadowed by a write of process p if there exists a write of p to the same location that will be written back at a later time (tso order $<$) and it is not provable that p ever observes that write.

The difficulty to describe TSO in a view-based manner emerges from TSO allowing local writes to temporarily hide others writes. In the axiomatic definition this is noticeable in that two terms combine to define the most recent write (memory order and local order) while in a view-based definition the serial view defines most recent write on the serial order. Therefore, a view-based model has to restrict the set of considered operations as they would conflict with the serial view property.

Example Figure 2.10 shows an example of an execution which is TSO consistent. This can be easily seen, if $write(x, 0)$ is buffered by p. On the other hand, the example is not PRAM consistent. Enhancing [Senf13], it can be summarized that TSO is incomparable to GWO, GAO, and PRAM (GPO) but strictly stronger than CC (GDO).

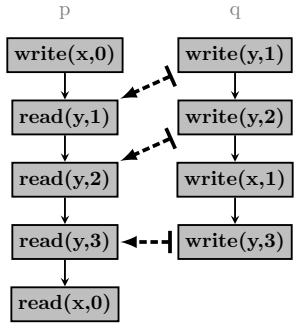

Figure 2.10.: *Example: TSO consistent execution that is not PRAM consistent.*

Figure 2.11 shows another example of an execution which is PSO consistent. While this time it is PRAM consistent as well, it is not GAO consistent.

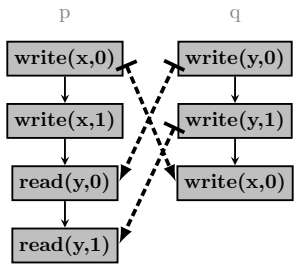

Figure 2.11.: *Example: TSO consistent execution that is not GAO consistent.*

2.3.9. Partial store ordering (PSO)

Partial store ordering is one memory model used in SPARC architectures. It provides a better performance than the default total store ordering mode, but is defined as optional in the architecture manual, so not all SPARC architectures may provide PSO. It allows to reorder writes after writes of different locations and writes after reads.

In the following, a simplified axiomatic definition for PSO is given based on the SPARC Architecture manual [Spar92; WeGe94; SiFC92]. 'Simplified' means that the original definition uses 7 axioms, but three of them (Atomicity, Termination, StoreStore) are irrelevant for this consideration. Atomicity only concerns *Swap* operations that are not covered by the introduced formalism, which only considers write and read operations. Termination gives a guarantee that a store will eventually be written to the memory and removed from the store buffer. And StoreStore considers $STBAR$ instructions which are also not covered by the used formalism.

Definition 2.18 ⟨ Partial store ordering (PSO) ⟩
An execution $\mapsto$ *of trace* $\mathcal{T}$ *is called* PSO consistent *if a memory order* $\leq$ *exists which respects:*

- *Order:* $\forall_{w_1,w_2 \in (w,*,*,*,*)_{\mathcal{T}}} : w_1 \leq w_2 \vee w_2 \leq w_1$
- *Value:* $\forall_{r \in (r,v,*,*,*)_{\mathcal{T}}}$:

$$\text{data}(r) = \text{data}\Big(max_{\leq}\left\{w \in (w,v,*,*,*)_{\mathcal{T}} \,|\, w <_P r \vee w \leq r\right\}\Big)$$

- *LoadOp:* $\forall_{r \in (r,*,*,*,*)_{\mathcal{T}}, o \in \mathcal{T}} : (r <_P o) \rightarrow (r \leq o)$
- *StoreStoreEq:* $\forall_{w_1,w_2 \in (w,v,*,*,*)_{\mathcal{T}}} : (w_1 <_P w_2) \rightarrow (w_1 \leq w_2)$

PSO consistency is best explained by describing a possible computer architecture: Each process has a store buffer for each memory location which buffers writes before writing them to memory. If a process reads a location for which a write exists in the corresponding store buffer then it reads the latest write's value from that store buffer, otherwise it reads the value from memory.

An execution is PSO consistent if a process observes its own operations in program order and there exists a total order on all write operations which is observed by all processes regarding others' writes. Also a process may not observe all writes of other processes as these may be hidden by own writes in its store buffer.

At this point of time no view-based definition for PSO was found that looks promising enough to be presented.

Example The execution in Figure 2.12 is PSO consistent as $write(x, 1)$ can be buffered by p. For TSO, following writes would be buffered too and there-

fore it can clearly not be TSO consistent. Furthermore, it is also not GWO consistent as for all writes are ordered by write-read-write order.

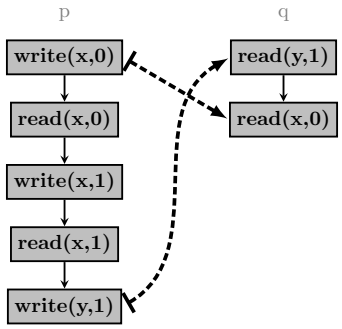

Figure 2.12.: *Example: PSO consistent executions that is neither TSO nor GWO consistent.*

Chapter 3

Paradigm: Testing as SAT

Contents

In a joint work [FMSS14; FMSS15] the program analysis approach testing is studied in context of view-based definitions. The testing problem checks whether sequences of operations of a concurrent program can be interleaved to a program execution that is consistent with a given weak memory model. This chapter covers the most important results of the cooperation in terms of our complexity analysis and presents the resulting SAT encoding as a representation of a memory-model aware testing problem.

3.1. Testing Problem

As first studied by Gibbons and Korach [GiKo97] in 1997 the *testing problem* is a core problem for verification. It asks if a given execution is an allowed execution as defined for sequential consistency. The problem can be extended to the *testing problem under weak memory* that checks the execution against an arbitrary memory model. The *testing problem under weak memory* is defined as follows: Given a concurrent execution, check whether this execution is a valid execution with respect to a given weak memory model.

As introduced in Definition 2.5, an execution is a sequence of read and write operations for each concurrent process and it is a valid execution if there exists an interleaving of the operations that satisfies the constraints of the weak memory model at hand. Other works might refer to executions as traces

as they talk about observed memory interactions when running a concurrent program (see also Definition 2.2 and Definition 2.5). The joint work with Furbach, Meyer and Schneider in [FMSS14; FMSS15] focused on the study of algorithms that solve the testing problem, so-called *testing algorithms*.

Testing algorithms have various applications in program analysis. In verification, over-approximation might result in counterexamples that can be proven to be infeasible with respect to the chosen memory model by a testing algorithm. Infeasible counterexamples might then be used for the refinement of the analysis.

Another application can be found in the best and worst case estimation of program execution times. Testing algorithms can identify paths that are not allowed with respect to the given memory model and thereby improve the analysis.

Only few publications exist on the design and complexity of algorithms for the testing problem. Gibbons and Korach [GiKo97] showed that the general problem is NP-complete for sequential consistency and linearizability. In 2005, Cantin, Lipasti, and Smith [CaLS05a] extended these results to several more models. In particular, they prove the testing problem to be NP-complete for TSO, PSO, relaxed memory ordering (RMO) [WeGe94], processor consistency (PC), release consistency [GLLG90], and the PowerPC memory model.

All of these approaches are tailored towards few specific memory models. In contrast, an approach is introduced that tackles the testing problem under different memory models in a *uniform* way.

The memory-model-aware testing problem $\textsc{Test}(\mathrm{M})$ is considered for every memory model M shown in Figure 2.1. The definition is as follows.

Definition 3.1 $\langle$ **Memory-model-aware Testing Problem** $\rangle$
Given a trace $\mathcal{T} \subseteq \mathcal{O}$.
Is there an execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ *that is valid under* M*?*

A test *succeeds under* M if there is a valid execution, otherwise it *fails under* M. In the Dekker mutual exclusion example, $\mathcal{T}_{Dekker}$ fails under SC but succeeds under SLOW.

We also consider restricted variants of the testing problem that admit more efficient algorithms: $\textsc{Test}_P(\mathrm{M})$ assumes a fixed number of processes in input tests, $\textsc{Test}_L(\mathrm{M})$ fixes the length of processes (number of operations), and $\textsc{Test}_V(\mathrm{M})$ studies the problem for a fixed number of variables.

3.2. Testing is NP-Hard for many models

Using a new reduction technique Furbach, Meyer, Schneider, and Senftleben [FMSS14; FMSS15] provide a reduction of the SAT decision problem to the testing problem for a wide range of memory models at the same time.

Given a weaker M_W and a stronger M_S memory consistency model, the set of allowed executions of M_W is a superset of the allowed executions of M_S. Also given a NP-hard decision problem Prob. Then, a $M_S \trianglelefteq M_W$*-range reduction of* Prob *to the testing problem* is a reduction of Prob to the testing problem that satisfies the following conditions. First, for each mapped testing problem that is valid for the weaker model M_W the corresponding instance of Prob succeeds as well. And second, for each problem instance of Prob that succeeds the mapped testing problem is valid for the stronger model M_S. This concept is illustrated in Figure 3.1.

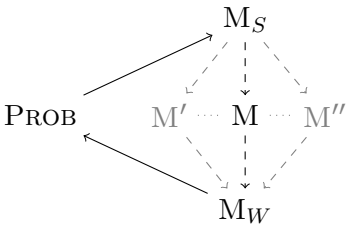

Figure 3.1.: *Illustration of the Range Reduction concept*

By construction of polynomial time computable $M_S \trianglelefteq M_W$-range reductions of a NP-hard problem it is shown that the testing problem for all models $M_S \trianglelefteq M \trianglelefteq M_W$ is NP-hard. The publication of Furbach, Meyer, Schneider, and Senftleben [FMSS14; FMSS15] presents four of these range reductions that can cover most of the memory models in Figure 2.1 and partitions them regarding their complexity results as seen in Figure 3.2. It also considers the influence of limiting some of the properties of the memory model like variable count, count of processes and process length.

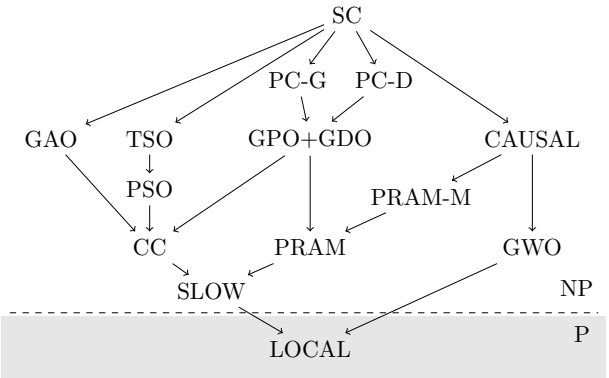

Figure 3.2.: *Complexity partitioning of memory models in terms of NP completeness of the corresponding testing problem. [FMSS14; FMSS15]*

More interesting in terms of this thesis is the other way around, a reduction of the testing problem to a well-known decision problem, the SAT problem.

3.3. Testing is in NP

This section shows that the memory-model-aware testing problem is in NP for all memory models in Figure 2.1. Using the formalism and results of Steinke and Nutt [StNu04] allows for a quite straightforward reduction. On the one hand their definition using partial orders is well suited to be expressed in a SAT encoding. And on the other hand their uniform definitions allow to only use one reduction for all considered memory models. The reduction yields a SAT formula consisting of two parameterized formulas that depend on the memory model.

3.3.1. Encoding: Components

The uniform encoding has to encode two different aspects: The existence of an execution for the given trace and the existence of a serial view for the given trace taking the execution of the first part into account. Both parts are expressed as propositional formulas in conjunctive normal form (CNF).

For a given trace $\mathcal{T}$, the existence of an execution is encoded in formula $\mathrm{EXE}(\mathcal{T})$. The encoding uses variables $ex_{w,r}$ for each pair of write and read operations $w, r \in \mathcal{T}$ that access the same variable and have the same value, $\mathrm{var}(w) = \mathrm{var}(r)$ and $\mathrm{data}(w) = \mathrm{data}(r)$.

Formula $\mathrm{EXE}(\mathcal{T})$ requires that every read has a write providing its value (3.1) and no read has two sources (3.2):

$$\bigwedge_{r \in \mathcal{T}} \bigvee_{\substack{w \in \mathcal{T} \\ \mathrm{var}(w)=\mathrm{var}(r) \\ \mathrm{data}(w)=\mathrm{data}(r)}} ex_{w,r} \tag{3.1}$$

$$\wedge \bigwedge_{\substack{r, w_1, w_2 \in \mathcal{T}, w_1 \neq w_2 \\ \mathrm{var}(w_1)=\mathrm{var}(w_2)=\mathrm{var}(r) \\ \mathrm{data}(w_1)=\mathrm{data}(w_2)=\mathrm{data}(r)}} (\neg ex_{w_1,r} \vee \neg ex_{w_2,r}) \tag{3.2}$$

Lemma 3.1 $\mathrm{EXE}(\mathcal{T})$ *is in CNF and cubic in the size of* $\mathcal{T}$*. Moreover,* $\mathrm{EXE}(\mathcal{T})$ *is satisfiable if and only if there is an execution* $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$*.*

The existence of a serial view of operations $\mathcal{O}'$ for an execution of $\mathcal{T}$ is encoded by formula $\mathrm{SV}(\mathcal{T}, \mathcal{O}', <)$. The formula takes as input a trace $\mathcal{T}$, a subset of operations $\mathcal{O}' \subseteq \mathcal{T}$, and a strict partial order $< \subseteq \mathcal{O}' \times \mathcal{O}'$. Serial views are defined relative to an execution. To access the execution determined

by $\mathrm{EXE}(\mathcal{T})$, formula $\mathrm{SV}(\mathcal{T}, \mathcal{O}', <)$ makes use of the variables $ex_{w,r}$ defined above.

Formally, a serial view is a strict total order $<_{sv} \subseteq \mathcal{O}' \times \mathcal{O}'$. It is encoded with variables sv_{o_1,o_2}, one for each pair of operations $o_1, o_2 \in \mathcal{O}'$. Intuitively, variable sv_{o_1,o_2} is set to true iff $o_1 <_{sv} o_2$ holds. The following exclusive-or ensures the serial view is total and asymmetric. The implication is transitivity. The exclusive-or is used as a macro for a conjunction and the implication as a macro for a disjunction so that the resulting formula is in conjunctive normal form:

$$\bigwedge_{\substack{o_1,o_2,o_3 \in \mathcal{O}' \\ o_1 \neq o_3 \\ o_1 \neq o_2 \neq o_3}} \Big[(sv_{o_1,o_2} \oplus sv_{o_2,o_1}) \quad \wedge \quad (sv_{o_1,o_2} \wedge sv_{o_2,o_3} \rightarrow sv_{o_1,o_3}) \Big] \tag{3.3}$$

Definition 2.6 requires that $<_{sv}$ refines $<$ to a total order:

$$\bigwedge_{\substack{o_1,o_2 \in \mathcal{O}' \\ o_1 < o_2}} sv_{o_1,o_2} \tag{3.4}$$

The next formula requires that for every pair $w \mapsto r$ we have $w <_{sv} r$ and that no write to the variable is placed in between:

$$\bigwedge_{\substack{w,r \in \mathcal{O}' \\ \mathrm{var}(w)=\mathrm{var}(r) \\ \mathrm{data}(w)=\mathrm{data}(r)}} \left[(\neg ex_{w,r} \vee sv_{w,r}) \wedge \bigwedge_{\substack{w' \in \mathcal{O}' \\ \mathrm{var}(w')=\mathrm{var}(r)}} (\neg ex_{w,r} \vee \neg sv_{w,w'} \vee \neg sv_{w',r}) \right] \tag{3.5}$$

Formula $\mathrm{SV}(\mathcal{T}, \mathcal{O}', <)$ is the conjunction of the Formulas (3.3) to (3.5). To state the relationship between *SerialView* $(\mapsto, \mathcal{O}', <)$ in Definition 2.6 and $\mathrm{SV}(\mathcal{T}, \mathcal{O}', <)$, the satisfying assignments are restricted to the propositional variables. An assignment *respects* $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ if $o_1 \mapsto o_2$ holds if and only if ex_{o_1,o_2} is set to true.

Lemma 3.2 $\mathrm{SV}(\mathcal{T}, \mathcal{O}', <)$ *is in CNF and cubic in its input. There is a strict total order* $<_{sv}$ *with* $<_{sv} =$ *SerialView* $(\mapsto, \mathcal{O}', <)$ *if and only if* $\mathrm{SV}(\mathcal{T}, \mathcal{O}', <)$ *has a satisfying assignment that respects* $\mapsto$.

3.3.2. Encoding: Uniform Reduction of Testing to SAT

Now, the above formulas are instantiated to solve the testing problem for the memory models defined using our formalism.

The procedure is illustrated by using the testing problem under *SLOW* consistency as an example and by its reduction to SAT.

Computing an execution is equivalent to determining a satisfying assignment for $\mathrm{EXE}(\mathcal{T})$. To make sure that the required serial views exist, the formula $\mathrm{SV}(\mathcal{T}, \bullet, \bullet)$ is instantiated with appropriate parameters:

$$\mathrm{EXE}(\mathcal{T}) \wedge \bigwedge_{\substack{p \in \mathcal{P} \\ x \in \mathcal{V}}} \mathrm{SV}(\mathcal{T}, (*, x, *, p, *)_{\mathcal{T}} \cup (w, x, *, *, *)_{\mathcal{T}}, <_P) \tag{3.6}$$

Test $\mathcal{T}$ succeeds under SLOW iff this formula is satisfiable. Note that $\mathrm{EXE}(\mathcal{T})$ ensures that the assignment to the execution variables matches an execution. And following Lemma 3.2, a satisfying assignment resembles an execution and all required serial views.

Therefore, $\textsc{Test}(SLOW)$ can be reduced to SAT in polynomial time and hence is in NP. Furthermore, as the procedure can be used for all models M defined via serial views, it can be concluded that for all these models $\textsc{Test}(\mathrm{M})$ is in NP.

3.4. Experiments

The introduced propositional formulas can be encoded for SAT solvers to tackle the testing problem. Consider the following trace $\mathcal{T}_\alpha$:

$$(w, x, 1).(r, x, 2) \parallel (w, x, 2).(r, x, 1)$$

The sequential consistency testing problem $\textsc{Test}(SC)$ for instance $\mathcal{T}_\alpha$ can be reduced to the following encoding:

$$EXE(\mathcal{T}_\alpha) \wedge SV(\mathcal{T}_\alpha, <_P)$$

The propositional formula can be encoded in the SMT-LIBv2 [BaFT17] language as shown in Appendix A.1. Using the Z3 SMT solver [MoBj08a] proves the propositional formula to be unsatisfiable as expected.

Similarly, the PRAM testing problem $\textsc{Test}(PRAM)$ for instance $\mathcal{T}_\alpha$ can be reduced to the following encoding:

$$\begin{aligned} EXE(\mathcal{T}_\alpha) \wedge SV_p((*, *, *, p, *)_{\mathcal{T}_\alpha} \cup (w, *, *, *, *)_{\mathcal{T}_\alpha}, <_P) \\ \wedge SV_q((*, *, *, q, *)_{\mathcal{T}_\alpha} \cup (w, *, *, *, *)_{\mathcal{T}_\alpha}, <_P) \end{aligned}$$

The respective encoding in the SMT-LIBv2 language is given in Appendix A.2.

While these examples were solvable in a fraction of a second, no big example problems have been encoded because at that time no tool for encoding was ready to be used. Because of that and since the focus of our work on elaborating modelling techniques rather than tools, this thesis will not provide detailed benchmark results with bigger examples.

3.5. Results and Future Work

The joint work [FMSS14; FMSS15] revealed important complexity results for the testing problem under weak memory. Besides the before-mentioned NP-hardness for many models and the NP-affiliation for all models, the work also proved that the testing problem is in P for some models.

More important, a representation of the testing problem for weak memory as a propositional formula was introduced. This representation allows to solve the testing problem by means of the SAT problem. While the reduction might not be the most compact representation, it still allows to solve the problem using well-known and established tools.

Further research into the testing problem and its use to identify porting bugs is shown in [LFHM17].

Although there are use cases for modelling traces and thereby for the testing problem, the more interesting questions target programs and all their possible traces. This resulted in further research on different approaches to express weak memory consistency for programs rather than traces.

Chapter 4

Paradigm: Temporal Logic

Contents

While temporal logics have been proven to be well suited to describe concurrent behavior, they have not been employed to describe weak consistent memory behavior until today. In the following we introduce a formalism to describe several of these models with linear temporal logic (LTL). This section provides a short introduction to LTL and explains the general idea of the specification. Furthermore, the memory system requirements and minimal properties for the specifications are explained. Finally, the properties are composed to a full specification for several memory models.

4.1. Linear temporal logic (LTL)

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 are 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.

The temporal operators used in this thesis are:

- $\mathsf{G}\varphi$ (Globally): φ holds in the current state and all future states.
- $\mathsf{F}\varphi$ (Finally): φ eventually holds at least once (now or in future).
- $\mathsf{X}\varphi$ (Next): φ holds in the next state of the path.
- $[\psi \,\mathsf{U}\, \varphi]$ (Until): φ holds until the first time when ψ holds, and ψ has to hold eventually.
- $\left[\psi \,\overline{\mathsf{U}}\, \varphi\right]$ (Weak Until): φ holds until the first time when ψ holds, and ψ may never hold (in which case φ holds ad infinitum).

For more information about temporal logics, see e.g., [Emer90; MaPn92; BaKa08; ClGP99; Schn03].

4.2. State Variables: Read/Write/Observed Events

In general, a multiprocessor system can be modeled by a set of processes communicating with a central memory system using a well-defined interface. This section will introduce the memory interface used for the LTL specifications that are presented in this chapter. The processes and the memory system interact via certain events, namely write events, and read events.

Each write operation $w = (w, v, d, p, i)$ as defined in Definition 2.1, issued by process p with issue index i and writing d to variable v, results in a write event $\langle w \rangle$ in the current state. Similar to Definition 2.1, a process p issuing a read of variable v with issue index i and the memory system answering that read request with value d results in the read event $\langle r \rangle$ with $r = (r, v, d, p.i)$. A process is expected to only issue a single operation at a given time, i.e., at most one event per process can occur at a given time. Furthermore, the issue index for operations of a single process are expected to be strictly increasing. Note that only $\mathrm{data}(r)$ with $r \in (r, *, *, *, *)_{\mathcal{O}}$ is an output of the memory system and everything else is considered as an input.

In addition to the mentioned write and read events and not part of the interface, the specifications use an observation event $\langle q, w \rangle$ which reflects that process q has observed write event $w \in (w, *, *, *, *)_{\mathcal{O}}$. A process is assumed to observe at most one write at a time.

4.3. Feasible Event Restrictions

To provide a well-defined context for the specifications, the environment has to fulfill certain assumptions.

Process First, quite obviously a process is assumed to only issue one operation at a time (4.1). Next, the same process event should not occur more than once (4.2), i.e., if an event of process p with issue index i occurred then no such event may occur in the future. Furthermore, for each process the issue index should start at 1 (4.3) and increase by 1 for each further operations issued (4.4).

$$\mathsf{G}\big((\langle o_1\rangle \wedge \langle o_2\rangle) \rightarrow \big(\mathrm{proc}(o_1) \neq \mathrm{proc}(o_2) \vee o_1 = o_2\big)\big) \tag{4.1}$$

$$\mathsf{G}\Big(\langle o_1\rangle \rightarrow \mathsf{XG}\big(\langle o_2\rangle \rightarrow \big(\mathrm{proc}(o_1) \neq \mathrm{proc}(o_2) \vee \mathrm{index}(o_1) \neq \mathrm{index}(o_2)\big)\big)\Big) \tag{4.2}$$

$$\mathsf{G}\big(\langle o\rangle \rightarrow \mathrm{index}(o) \geq 1\big) \tag{4.3}$$

$$\forall_{i>1} : \big(\neg\langle(w,v,d,p,i)\rangle\big)\ \overline{\mathsf{U}} \bigvee_{o'\in\mathcal{C},v'\in\mathcal{V},d'\in\mathcal{D}} \langle(o',v',d',p,i-1)\rangle \tag{4.4}$$

Together, the equations from Equation 4.1 through 4.4 constrain the behavior of the processes to expected behavior.

Memory On the memory side, 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 $\langle q,w\rangle$ may only occur if there was a corresponding write event $\langle w\rangle$ before (4.5). Second, a process should observe each write event only once (4.6).

$$\mathsf{G}\big[\big(\neg\langle q,(w,v,d,p,i)\rangle\big)\ \overline{\mathsf{U}}\ \langle(w,v,d,p,i)\rangle\big] \tag{4.5}$$

$$\mathsf{G}\big(\langle q,w\rangle \rightarrow \mathsf{XG}\neg\langle q,w\rangle\big) \tag{4.6}$$

The combination of both equations limits observation events to at most one per process and corresponding write.

The most interesting property so far is the semantics of a read operation. Read operations should either return the default value 0 as long as there was no observed write to that location (4.7) or the value of the latest observed write event (4.8).

$$\forall_{r\in(r,v,*,p,*)_{\mathcal{O}}} : \big(\langle r\rangle \rightarrow \mathrm{data}(r) = 0\big)\ \overline{\mathsf{U}} \bigvee_{w\in(w,v,*,q,*)_{\mathcal{O}}} \langle q,w\rangle \tag{4.7}$$

$$\forall_{w\in(w,v,d,*,*)_{\mathcal{O}},r\in(r,v,*,q,*)_{\mathcal{O}}} :$$
$$\mathsf{G}\Big(\langle q,w\rangle \rightarrow \big[(\langle r\rangle \rightarrow data(r) = d)\quad \overline{\mathsf{U}} \bigvee_{\substack{w\in(w,v,*,*,*)_{\mathcal{O}}\\ w\neq w'}} \langle q,w'\rangle\big]\Big) \tag{4.8}$$

4.4. Weak Consistency Properties

In the following, the additional properties that are needed to specify the different memory models will be presented. This section will progress from the weakest model which requires the least restrictions to the stronger models.

4.4.1. Local consistency (LOCAL)

Local consistency as introduced in Definition 2.14 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. In other words, a write event requires the issuing process to immediately observe its own write.

This property can be expressed in LTL as follows:

$$\mathsf{G}\big(\langle(w,v,d,p,i)\rangle \rightarrow \langle p,(w,v,d,p,i)\rangle\big) \tag{4.9}$$

4.4.2. Slow consistency (SLOW)

SLOW as introduced in Definition 2.13 extends local consistency by requiring processes to observe the writes of another process to the same location in the order they were issued. 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.

The corresponding LTL representation of that property is as follows:

$$\mathsf{G}\Big(\langle q,(w,l,d,p,i)\rangle \rightarrow \mathsf{XG} \bigwedge_{\substack{j\leq i\\ d'\in\mathcal{D}}} \neg\big[\langle q,(w,l,d',p,j)\rangle\big]\Big) \tag{4.10}$$

4.4.3. Pipelined RAM consistency (PRAM)

PRAM as introduced in Definition 2.11 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. That is, if a process observes a write operation, then it has to observe all earlier writes of the originating process beforehand. Therefore, the PRAM specification extends slow consistency by an additional property:

$$\begin{aligned}\forall_{j<i}: \quad &\Big(\big[\mathsf{F}(\langle(w,v,d,p,i)\rangle)\big] \wedge \big[\mathsf{F}(\langle(w,v',d',p,j)\rangle)\big]\\ &\rightarrow \big[\neg\langle q,(w,v,d,p,i)\rangle \,\overline{\mathsf{U}}\, \langle q,(w,v',d',p,j)\rangle\big]\Big)\end{aligned} \tag{4.11}$$

4.4.4. Cache consistency (CC)

CC consistency as introduced in Definition 2.12 is stronger than slow consistency and requires that if a process observes two writes to the same location, then if another process observes them as well they have to be in the same order. This can be expressed with the following property:

$$\Big[\mathrm{var}(w_1)=\mathrm{var}(w_2)\wedge\mathsf{F}\big(\langle q,w_1\rangle\wedge\mathsf{F}(\langle q,w_2\rangle)\big)\Big]\rightarrow\mathsf{G}\big(\langle p,w_2\rangle\rightarrow\mathsf{G}\neg\langle p,w_1\rangle\big) \tag{4.12}$$

4.4.5. Total store ordering (TSO)

TSO as defined in Definition 2.15 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 behavior allows us to describe TSO as well. The additional event $\underline{\overline{w}}$ models that the write w corresponding to $\langle w \rangle$ left the store buffer. This allows us to express that a value should be available to other processes. In general $\underline{\overline{w}}$ can not be expressed using $\langle q, w \rangle$ straightforward 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.

$$[\neg \underline{\overline{w}} \,\mathsf{U}\, \langle w \rangle] \wedge \mathsf{G}\Big[\underline{\overline{w}} \wedge \underline{\overline{w'}} \rightarrow w = w'\Big]$$

$$\wedge\, \mathsf{G}\Big[\underline{\overline{w}} \rightarrow \mathsf{XG}\big(\underline{\overline{w'}} \wedge \mathrm{proc}(w) = \mathrm{proc}\big(w'\big) \rightarrow \mathrm{index}\big(w'\big) > \mathrm{index}(w)\big)\Big] \tag{4.13}$$

$$\mathsf{G}\Big[\langle q, w \rangle \wedge \big(\mathrm{proc}(w) \neq q\big) \rightarrow \underline{\overline{w}}\Big] \tag{4.14}$$

$$\mathsf{G}\Big(\langle (w, x, v, p, i) \rangle \rightarrow \Big[\neg \langle p, (w, x, v', p', i') \rangle \,\mathsf{U}\, \underline{\overline{(w, x, v, p, i)}}\Big]\Big) \tag{4.15}$$

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 (4.13). A write may only be observed by another process at the time it hits the memory (4.14). Last, a write will eventually leave the store buffer and until then the process may not observe other processes' writes to that location (4.15).

Further work might investigate if a specification for TSO actually requires the additional set of events. If that is the case then it may be of interest why defining TSO requires more effort than other models.

4.4.6. Sequential consistency (SC)

Sequential consistency (SC) as introduced in Definition 2.7 defines a behavior that may occur if programs are executed on a single processor. It requires all processes to agree upon a single sequential total ordering of all write operations. The first required property (Totality) can be expressed as:

$$\mathsf{G}\Big[\langle w \rangle \rightarrow \big(\mathsf{F}\langle q, w \rangle\big)\Big] \tag{4.16}$$

That means, whenever a write event occurs, then each process has to observe that write operation some time in the future. The other property to ensure a unique sequential representation is as follows:

$$\Big[\mathsf{F}(\langle p, w_1 \rangle \wedge \mathsf{F}\langle p, w_2 \rangle)\Big] \rightarrow \Big[\mathsf{F}(\langle q, w_1 \rangle \wedge \mathsf{F}\langle q, w_2 \rangle)\Big] \tag{4.17}$$

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

4.5. Property Verification

Given a multithreaded environment, a *safety property* is an assertion that should hold in all reachable states. For example, multithreaded programs often come along with code parts that require mutually exclusive access to some variables to achieve the correct behavior, i.e., they contain a critical section. Mutual exclusion can be expressed as a safety property for example by having each process increase and decrease the same variable by 1 in their critical section. If that variable is ever 2 or bigger then the mutual exclusion was violated. An example with the given constraints is Peterson's mutual exclusion protocol [Pete81] shown in Figure 4.1.

The reachable states depend on the input program as well as the memory system. The properties defined in the previous section allow to abstract from the actual memory system implementation and directly use the memory specification for verification.

Given a representation of a multithreaded program that uses a distinct set of variables to indicate a memory event corresponding to a memory operation as defined in Definition 2.1. Then the verification works as follows. The program yields a set of traces covering every possible interleaving of processes and memory reads retrieving every possible combination of values. Now, the LTL specification limits the verification of the safety property to traces which have a valid execution for the considered memory model. Last, the safety property is checked for all states of this subset of traces.

This can be written as follows:

$$(\text{Process modules}) \vDash (\text{Memory Specification}) \rightarrow (\text{Safety Property})$$

Multithreaded programs can be implemented as parallel modules in SMV [CCGR99] and the properties can be expressed in LTL. This allows to utilize different LTL model checking tools that use the SMV input language for verification.

4.6. Experiments

Following the before mentioned approach, safety properties can be verified with tools like NuSMV [CCGR99] or NuXMV [CCDG14]. These tools use either an BDD approach for verification or search for counterexamples using a SAT-based BMC approach. To this end, the environment and specifications described in the previous section have been implemented in the SMV [CCGR99] input language (Example in Appendix B). For the process abstraction, the environment follows an assembler representation of the processes or uses specific designed test cases. Because SMV does not allow for any form of quantification a preprocessor that supports file inclusion and quantification has been developed. Some of the experiments are included in the appendix but only in their simplified form as their unrolled form is too lengthy to be included in here.

In the following, several algorithms are described and analyzed for their minimal required memory models. The considered algorithms are the well-known mutual exclusion algorithm due to Peterson, followed by a chained computation scenario, and closing with a simple consumer-producer algorithm.

4.6.1. Peterson's Mutual Exclusion Algorithm

First, Peterson's mutual exclusion algorithm (see Figure 4.1) will be shown to work as expected with SC, but fail with weaker consistency models.

Procedure p	**Procedure** q
1 x = 1;	1 y = 1;
2 t = 0;	2 t = 1;
3 **while** *y=1 & t=0* **do** ;	3 **while** *x=1 & t=1* **do** ;
`// Begin Critical Section`	`// Begin Critical Section`
4 d = d + 1;	4 d = d + 1;
5 d = d - 1;	5 d = d - 1;
`// End Critical Section`	`// End Critical Section`
6 x = 0;	6 y = 0;

Figure 4.1.: *Peterson mutual exclusion protocol [Pete81] in pseudo code. Both processes p and q set their flag to express their intention to enter the critical section. In case of a tie, the turn variable t is used to determine which process is allowed to go first.*

Peterson's mutual exclusion algorithm works as follows: Whenever a process wants to enter the critical section, it sets its own flag, either x or y. Then, it sets the turn variable t to either 0 or 1 respectively. 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 contains operations to increment and then decrement a data variable d. Following the mutual exclusive execution of the critical sections the value of the data variable d should always be either 0 or 1.

To examine the memory behavior of that algorithm, it has to be translated in a representation which reveals the individual load-store instructions as shown in Figure 4.2. The representation uses memory locations already present in the high-level implementation (see Figure 4.1): x, y, t, and d. Consider the non-atomic instructions $d = d + 1$ and $d = d - 1$. Assuming the initial value of d is 0, then d 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, d 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 they add unexpected observable behavior if the

mutual exclusion is not ensured. In this case, we would like to verify that the location d is 0 or 1 at all times.

```
1  write(x, 1)
2  write(t, 0)
3  reg = read(y)
4  if (reg=0) then goto 7
5  reg = read(t)
6  if (reg=0) then goto 3
7  reg = read(d)
8  write(d, reg + 1)
9  reg = read(d)
10 write(d, reg - 1)
11 write(x, 0)
12 goto 12
```

```
1  write(y, 1)
2  write(t, 1)
3  reg = read(x)
4  if (reg=0) then goto 7
5  reg = read(t)
6  if (reg=1) then goto 3
7  reg = read(d)
8  write(d, reg + 1)
9  reg = read(d)
10 write(d, reg - 1)
11 write(y, 0)
12 goto 12
```

Figure 4.2.: *Peterson's mutual exclusion algorithm in assembler for two processes. The register reg is used for storing the loaded value locally.*

To verify the safety property, the instructions depicted in Figure 4.2 are modelled on two processor modules in the SMV input language (Appendix B.2). Then, an LTL specification which reads $(\mathit{MemorySpecification}) \rightarrow (G\ (d = 0 \vee d = 1)))$ is added. Whenever a path of the state transition system satisfies the specification of the memory model, then the safety property has to be checked, i.e., in those cases it is asserted that it will never be the case that d is a value different from 0 or 1. Using NuSMV, the safety property has been shown to be valid for SC, and to be invalid for the other models described in this section: LOCAL, SLOW, CC, PRAM, and TSO consistency by providing counterexamples.

4.6.2. Chained Computation

Another scenario analyzed covers the concept of a chained computation: One process computes a value, another uses that value, and a third collects both the intermediate and the final values. This is depicted in Figure 4.3: 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 postprocessing.

SC 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 $(\mathit{MemorySpecification}) \rightarrow (G\ ((\mathit{done} = 1) \rightarrow (v1 = v_P \wedge v2 = v_Q)))$, the algorithm was shown to work as expected for SC using NuSMV. While counterexamples could be found for PRAM, and CC, the results showed that CAUSAL is

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

Procedure p	**Procedure** q	**Procedure** r
1 int local; // Calc: local 2 dataP = local; 3 flag = 1;	1 int local; 2 **while** *f!=1* **do** ; 3 local = dataP; // Calc: local 4 dataQ = local; 5 flag = 2;	1 int v1,v2; 2 int local; 3 **while** *f!=2* **do** ; 4 v1 = dataP; 5 v2 = dataQ; // Postprocessing 6 done = 1;

Figure 4.3.: *A chained computation scenario in assembler code. Process p produces a value that is read by process q. Then, process q will store an updated value to another location. Finally, process r reads both locations.*

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 behavior.

4.6.3. Producer Consumer

The last algorithm analyzed is a simple producer-consumer algorithm as seen in Figure 4.4. 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 *ready* to 1 and c signals that it has read the value by setting *ready* to 0 again.

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

Procedure p	**Procedure** c
1 **for** *i=0..N-1* **do** 2 | data = i; 3 | ready = 1; 4 | **while** *ready* **do** ; 5 **end**	1 int[] result[N]; 2 **for** *i=0..N-1* **do** 3 | **while** *!ready* **do** ; 4 | local[i] = data; 5 | ready = 0; 6 **end** 7 done = 1;

Figure 4.4.: *Producer-Consumer Algorithm in pseudo code. The producer p sets ready to signal data availability, and consumer q resets the value to indicate that it retrieved the value.*

Clearly, the depicted producer-consumer algorithm works as expected for SC: Whenever c observes *ready* to be 1, the corresponding write to *data* is visible to c, too. Therefore, there is only a single possible outcome for the

values of the result registers in a sequentially consistent environment.

Just like the scenario mentioned before, it was proven that the correctness property $(MemorySpecification) \rightarrow (G\ ((done = 1) \rightarrow \bigwedge_{i=0}^{N-1}(result[i] = i)))$ does not hold for LOCAL, SLOW and CC, but in fact holds for SC as expected but more interesting for PRAM as well. Following that, costly synchronization may be saved for producer-consumer algorithms in distributed scenarios (as PRAM is a distributed concept).

Closing remark

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 behavior.

Verifying against multiple models is as easy as replacing the LTL specification. Neither the processor representation nor the property description have to be changed.

Using the described technique, more small litmus-test-alike examples have been verified to hold for different memory models and fail for weaker models. Concluding, the mentioned technique allows to determine the minimal required consistency guarantees - and consistency models - to ensure that a given property holds for the analyzed examples.

4.7. Results and Future Work

This chapter introduced a novel approach to specify weak memory systems using temporal logic. Using temporal logic allowed to describe the behavior 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 the resulting LTL specifications, so that established tools can be used to verify multithreaded programs. Moreover, the approach allows 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 non-trivial examples like Peterson's mutual exclusion introduced in Figure 4.1 already require several Gigabyte of memory to finish. Using NuSMV's BMC with reasonable bounds allows to inspect more examples, but inevitable the tool will run out of memory for more complex examples as well.

Some effort was spend to express the specifications using computational tree logic (CTL) in order to utilize different tools and techniques that might be less

expensive in space, but to no avail. For some of the properties, the unsuccessful attempts were concluded with a reasonably confident conclusion that they are not expressible in CTL at all. While there are more expressive logics used in specification like CTL* or μ-Calculus, they get harder to grasp and therefore the initial benefit of using LTL as a comprehensible representation is lost.

In the future, more models could be described using temporal logics. Furthermore, finding better representations would be of interest, especially for a possibility to get rid of the write index. 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. Moreover, the relationship between different models could be verified using our representations (e.g., that SC implies PRAM). Finally, as the proposed verification approach has to build the full state space before it restricts it to valid executions using the LTL specification, there might exist another approach that directly uses the specification to build the state space and thereby requires less memory.

Chapter 5

Paradigm: Operational Semantics

Contents

This section contains an operational 'architectural' characterizations of most introduced weak memory models. To this end, reference machines for each one of these memory models are provided. A reference machine is an abstract memory architecture with a well-defined load/store interface that is build in a modern system-level language. In addition to the architectural composition, the correctness and completeness of these machines are discussed. This means that the reference machines can only execute computations that belong to the considered memory model (*correctness*), and that the reference machine can execute *all* computations that belong to the considered memory model (*completeness*).

In order to discuss these reference machines, some common basic components are introduced in the next section. Then, the reference machines are presented and their correctness and completeness are briefly discussed. Finally, further details on the actual implementation in the synchronous language Quartz [Schn09] are given.

5.1. Basic Components

This section introduces some basic components that are used by most of the reference machines that will be shown later. Of course, there exist plenty of different components that might be more suitable for the implementation of specific memory models. To ease comparability, the reference machines use as few different structural elements as possible.

FIFO

The FIFO component is a first-in-first-out buffer which buffers memory operations as tuples. It holds the operation type (read or write), the issuing process' id, the memory address, and in case of a write operation the value to be written. The component's interface is defined as follows:

```
module FIFO (
    event ?pop,
    event ?push,
    event !isempty,
    event isfull,
    // input : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) ?inp,
    // output : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) !outp
)
```

The outputs `isempty` and `isfull` signal the current state of the buffer. Both data channels `inp` and `outp` consist of a valid flag, the id of the originating processor, the memory location to write to and the actual value to write. Adding an entry to the buffer is handled by input signal `push` while providing the data to `inp`. Similarly, removing the first entry of the buffer is handled by input signal `pop` and reading data from `outp`.

A Quartz implementation is shown in Listing C.1.1. There exists a slightly different version that extends entries by one additional clock value and another version that extends entries by N clocks, one for each process. A clock value is a natural number that expresses observation progress. The additional versions can be found in Listings C.1.2 and C.1.3.

BAG

The BAG component shares the same interface as the FIFO component but slightly differs in its semantics: While the FIFO component will always return and remove the oldest entry when signal `pop` is set, the BAG component may non-deterministically return and remove any stored entry.

MEM

The memory unit MEM stores the latest write to a location and in case of a read operations returns the most recently written value of a location. It may contain values for several or just a single location and distributes read values to the corresponding components.

Distributor

A distributor DIST is responsible for distributing memory operations to a selection of the connected components. Depending on the model it may just route the operation or duplicate it to several destinations.

Receiver

Similarly, a receiver REC receives read results from multiple components and delivers them to the corresponding process. Both DIST and REC are actually more routing structures than active components.

Arbiter

The arbiter is one of the characterizing components of a memory model. It determines the order in which memory operations are passed from the connected components based on the model.

Each step, the arbiter either selects a component to deliver an operation if it is not empty, or chooses to idle. While this behaviour is not desirable from a performance point of view, it is required to achieve completeness results.

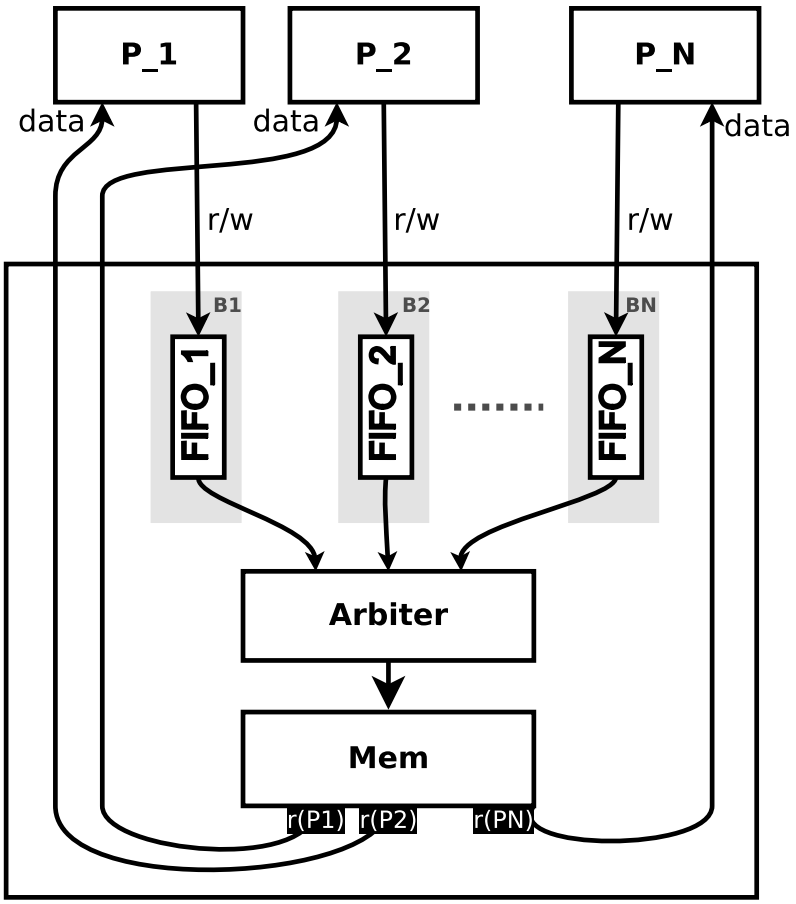

Figure 5.1.: *Reference Machine for Sequential consistency.*

5.2. Reference Machines

In this sections, the reference machines for most introduced consistency models are presented. To that end, the architecture of the reference machine is described using the above mentioned basic components. After this, the correctness and completeness of the given reference machine is discussed briefly, where *correctness* means that all computations of our reference machine belong to the considered weak memory consistency model, and conversely, *completeness* means that our reference machine can simulate all possible executions of the considered weak memory consistency model. Hence, the reference machines exactly characterize the weak memory consistency model in an operational/architectural manner.

5.2.1. Sequential consistency (SC)

Architecture: Figure 5.1 shows an implementation of a reference machine for SC. It consists of a FIFO buffer for each connected process, which is directly connected to their process interface, an arbiter which selects non-deterministically from all FIFOs and passes the operations to the memory unit or otherwise idles. The memory unit passes processed reads to the process that issued the read operation.

Correctness: Using FIFO buffers ensures by construction that the read and write operations of each process are kept in order (maintaining $<_P$). The arbiter generates a serialization of all memory operations while maintaining the program order and therefore satisfies sequential consistency.

Completeness: If an arbitrary execution is sequentially consistent, then a serial view exists for all memory operations which respects $<_P$. If the arbiter uses this view to make its non-deterministic choices, then the resulting behavior is equivalent to the considered execution. Consequently, all sequentially consistent executions are covered by the given reference machine.

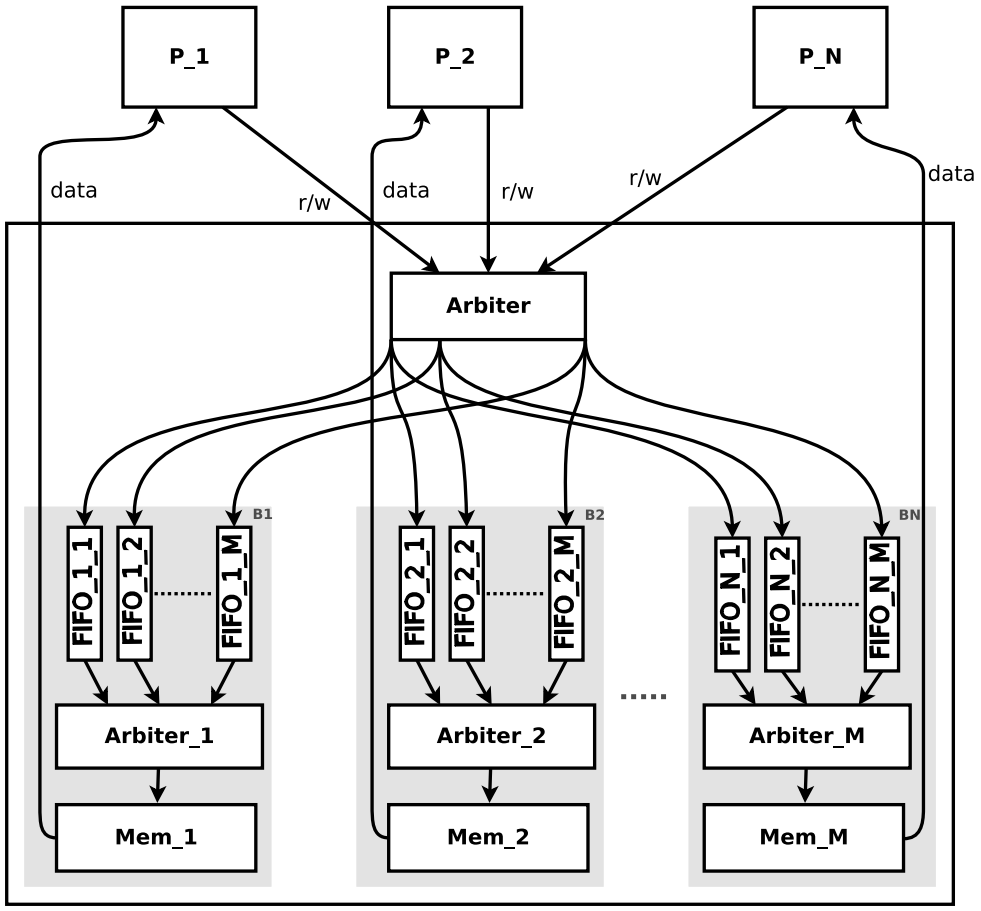

Figure 5.2.: *Reference Machine for Processor consistency.*

5.2.2. Processor consistency (Goodman) (PC-G)

Architecture: The structure of the PC-G reference machine is shown in Figure 5.2 for a given set $\mathcal{P}$ of n processes. For each process $P_i \in \mathcal{P}$, the memory system has M FIFO buffers $FIFO_{i,x}$, one for each memory cell x, an arbiter $Arbiter_i$ and a memory unit Mem_i. The system has an additional arbiter which non-deterministically takes memory commands from the processes and inserts them into the corresponding FIFOs. Reads from process i to memory cell x are inserted into the process' $FIFO_{i,x}$ only, writes to memory cell x are inserted into all $FIFO_{i,x}, i \in P$. The main arbiter keeps a clock vector $t_i \in \mathbb{N}^n, i \in P$ and tags each write from process i with a tuple (i, t_i) and increases t_i after distributing the write. The processes' arbiters $Arbiter_i$ keep each a clock vector $t_{i,j} \in \mathbb{N}^n, i, j \in P$, too. They select non-deterministically one of the FIFOs to read from but only pop an element from the selected FIFO if its tag's clock is the next element to be processed for that process: $clock\big((k,t)\big) = t_{i,k} + 1$

Correctness: The order in which the main arbiter passes commands to its corresponding FIFOs cells corresponds to a total order $<_v$ on all memory commands regarding this memory address. All of a process' write operations are tagged with a steadily increasing counter. As this counter reflects their order in $<_P$ and the operations are only passed to the memory units in that particular order, the $<_P$ is maintained. Therefore, the processes' arbiters construct each a serial view on all read operations of their corresponding process and all processes' write operations which respects $<_P$ and $\bigcup_{v \in V} <_v$ and as a consequence the executions of the reference machine are PC-G.

Completeness: Consider a PC-G consistent execution (by Definition 2.8):

$$\forall_{v \in \mathcal{V}} \exists_{<_v} : <_v = SerialView\left(<_P \mid (*, v, *, *, *)_{\mathcal{T}}\right)$$
$$\wedge\ \forall_{p \in P} \exists_{<_{sv}} : <_{sv} = SerialView\left((\cup_{v \in \mathcal{V}} <_v) \bigcup <_P \mid (*, *, *, p, *)_{\mathcal{T}} \cup (w, *, *, *, *)\right)$$

If the main arbiter uses the orders $<_v$ as selection criteria (it only selects an operation if its predecessor in $<_v$ has already been passed to the FIFOs before) then the FIFOs maintain $<_v$. As the processes' arbiters use an operations' tag as selection criteria they maintain $<_P$ by the way the tags are generated. Hence, the execution is covered by the machine. The assumption must be incorrect.
Therefore, every PC-G consistent execution is covered by the reference machine.

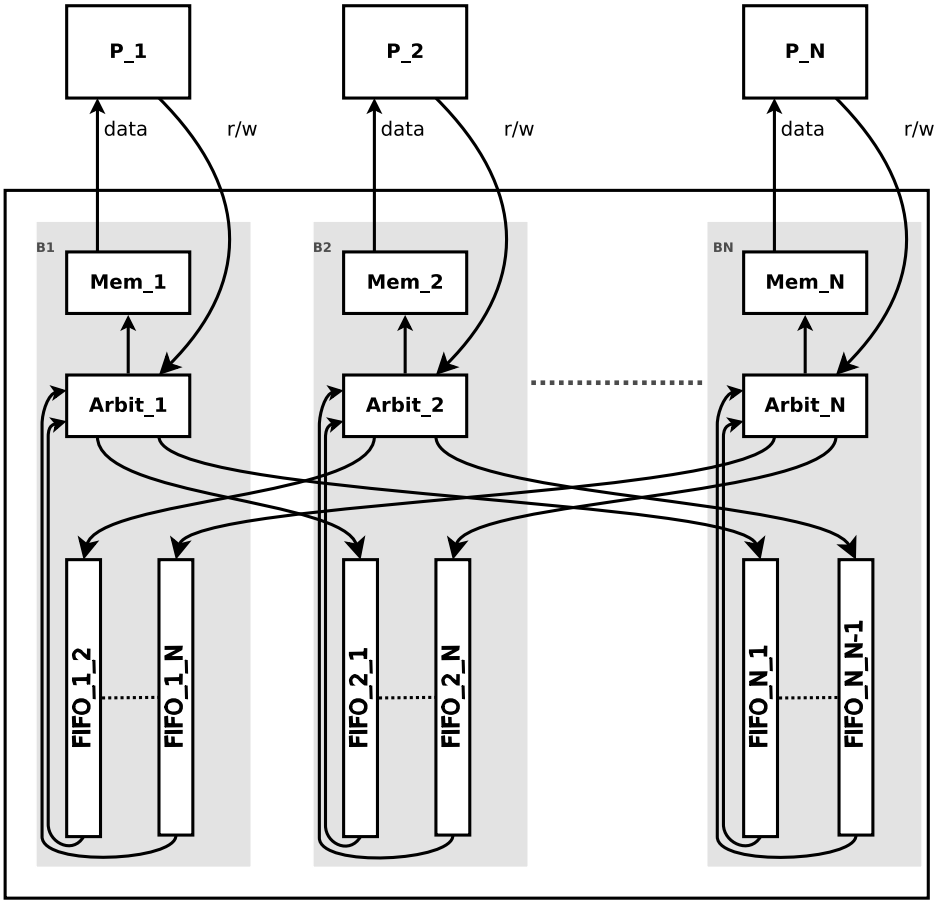

Figure 5.3.: *Reference Machine for Causal consistency.*

5.2.3. Causal consistency (CAUSAL)

CAUSAL requires the reference machine to keep track of causal dependencies created by writes following reads. This tracking is achieved by introducing clocks which represents the progress a process already observed from other processes.

Architecture: The structure of the CAUSAL reference machine is shown in Figure 5.3 for a given set $\mathcal{P}$ of n processes. It is based on the 'Simple Algorithm' described by Ahamad et. al [ABHN91]. For each process $P_i \in \mathcal{P}$, the memory system has an arbiter $Arbit_i$, a memory unit Mem_i and $n-1$ FIFO buffers $FIFO_{i,j}, j \in \{1 \dots m\}, i \neq j$. The arbiters hold a clock vector $t_i \in \mathbb{N}^n$ which is used to determine the execution order of received writes and is appended to the writes sent to other processes. Before 'sending' writes to the other processes, the arbiter increases the clock vector's value of the entry corresponding to its process $t_i[i]$. Upon 'retrieving' a write from another process, the arbiter updates the clock vector's value of the entry corresponding to the sending process $t_i[j]$. A write is only retrieved from $FIFO_{i,j}$ if its clock is lower or equal than the arbiters clock with the writes clock entry replaced: $t_w[k] \leq t_i[k], k \neq j$

Correctness: The order in which the operations are passed to a process memory unit provides a serial view on all writes and the process' own reads.

Assume that an execution produced by the machine is not causal correct. Then, the given serial view must violate either $<_P$ or $<_{wrw}$. As the arbiter forwards a process' own operations in-order to its memory location, the serial view clearly respects $<_i$. The others processes' writes are inserted in-order into FIFOs and passed to the memory unit in-order from the FIFOs. Therefore, the serial view respects $<_P$ as well. If the serial view violates $<_{wrw}$, then there exists a write w_1 which writes to a read r local ordered before another write w_2 with the serial view ordering w_2 before w_1. If process P_i's w_1 was read by the process P_j before it issues w_2 then P_j increased its clock value $t_j[i]$ and w_2 was sent with a clock vector which contained the new value. Another process P_k can only read w_2 if its clock vector value is greater or equal to the value w_2 was tagged with: $t_k[i] \geq t_j[i]$. On the one hand, only writes received from P_i can increase $t_k[i]$ and writes are tagged with increasing clock values regarding local order. And on the other hand, writes appear in local order in the serial view as shown before. Then, $<_{wrw}$ could not have been violated. ↯ Therefore, the assumption must be incorrect and all executions are CAUSAL.

Completeness: Assuming there exists a CAUSAL execution $(\mathcal{P}, \mathcal{V}, \mathcal{O}, <_P, \mapsto)$ which cannot be covered by the machine. As the execution is CAUSAL the following holds (by Definition 2.10):
$\forall_{p \in \mathcal{P}} \exists_{<_{sv}} : <_{sv} = SerialView\,(<_p \cup <_P \cup <_{wrw} | (*, *, *, p, *)_{\mathcal{T}} \cup (w, *, *, *, *)_{\mathcal{T}})$
If each arbiter uses the serial view of its memory location (which exists according to the definition) as selection order, then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. It remains to show that the selection order is a valid one. Since the arbiter may choose non-deterministically, the only restriction is its clock vector. If it is not a valid selection order for process i then the operation to select next would need to be a write from another process as all own operations may be selected without checking the clock vector. Without loss of generality, we may say that this operation is a write from process j called w_1. The write w_1 would need to have a non native clock value which is greater than process i's corresponding clock vector value: $t_{w_1}[k] > t_i[k], k \neq j$. As clock vector values are only increased if a write is performed, that would imply that process j observed a write w_2 from process k before issuing w_1 and that process i did not observe w_2 before w_1. But that would contradict $<_{wrw}$ and therefore the assumption must be incorrect. ↯ Therefore, every CAUSAL execution is covered by the reference machine.

Alternative implementation The presented machine could be modified to write new clock values to a temporary clock vector instead of updating clock values directly and to update corresponding clock values with their temporary counterpart when a location is read. However, this modification might allow the retrieval of some writes that would not have been retrieved in the former machine.

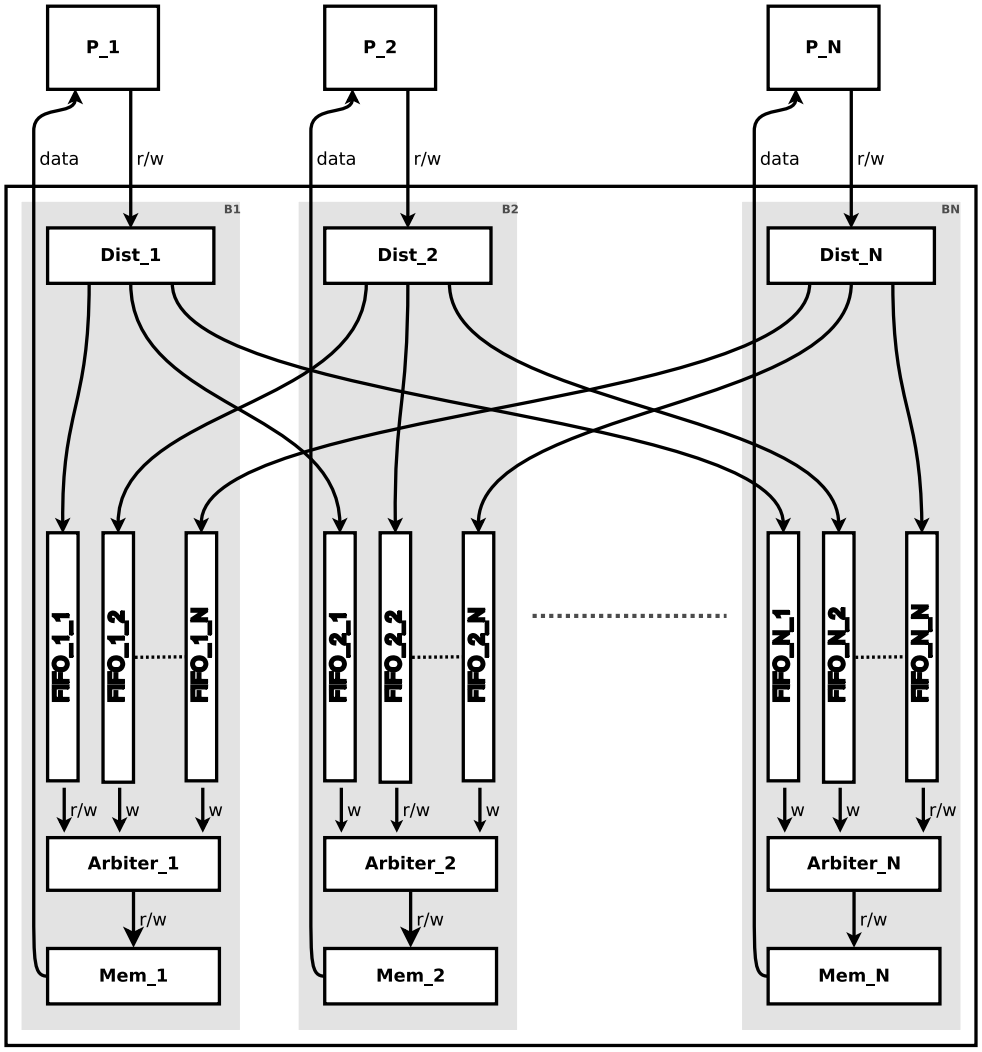

Figure 5.4.: *Reference Machine for Pipelined RAM consistency.*

5.2.4. Pipelined RAM consistency (PRAM)

As can be seen in Figure 5.4, the reference machine for PRAM provides a single memory for every process. Intuitively, this structure reflects a memory model that is more likely found in distributed computing.

Architecture: The implementation of the reference machine for PRAM consistency is shown in Figure 5.4 for a given set $\mathcal{P}$ of n processes. For each process $P_i \in \mathcal{P}$, the memory system has a distributor $Dist_i$, an arbiter $Arbiter_i$, a memory unit Mem_i, and n different buffers $FIFO_{i,j}$ for $j \in \{1, \ldots, n\}$. A distributor $Dist_i$ broadcasts received writes to all corresponding $FIFO_{i,j}$ for $j \in \{1, \ldots, n\}$, and sends all received reads to its $FIFO_{i,i}$. The arbiters choose non-deterministically from the connected FIFOs.

Correctness: Using FIFO buffers ensures by construction that the read and write operations of each process are kept in order (maintaining $<_P$). The arbiter takes elements from the top of a FIFO buffer and issues the operation to the memory unit. Therefore, the arbiter constructs a serial view on write operations of all processes and the read operations of its corresponding process.

Completeness: Consider an arbitrary PRAM execution. If each arbiter selects its actions according to the execution's serial view corresponding to its process, then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. Hence, as no writes are lost and an arbiter can always wait until the required value is available, every PRAM consistent execution is covered by the reference machine.

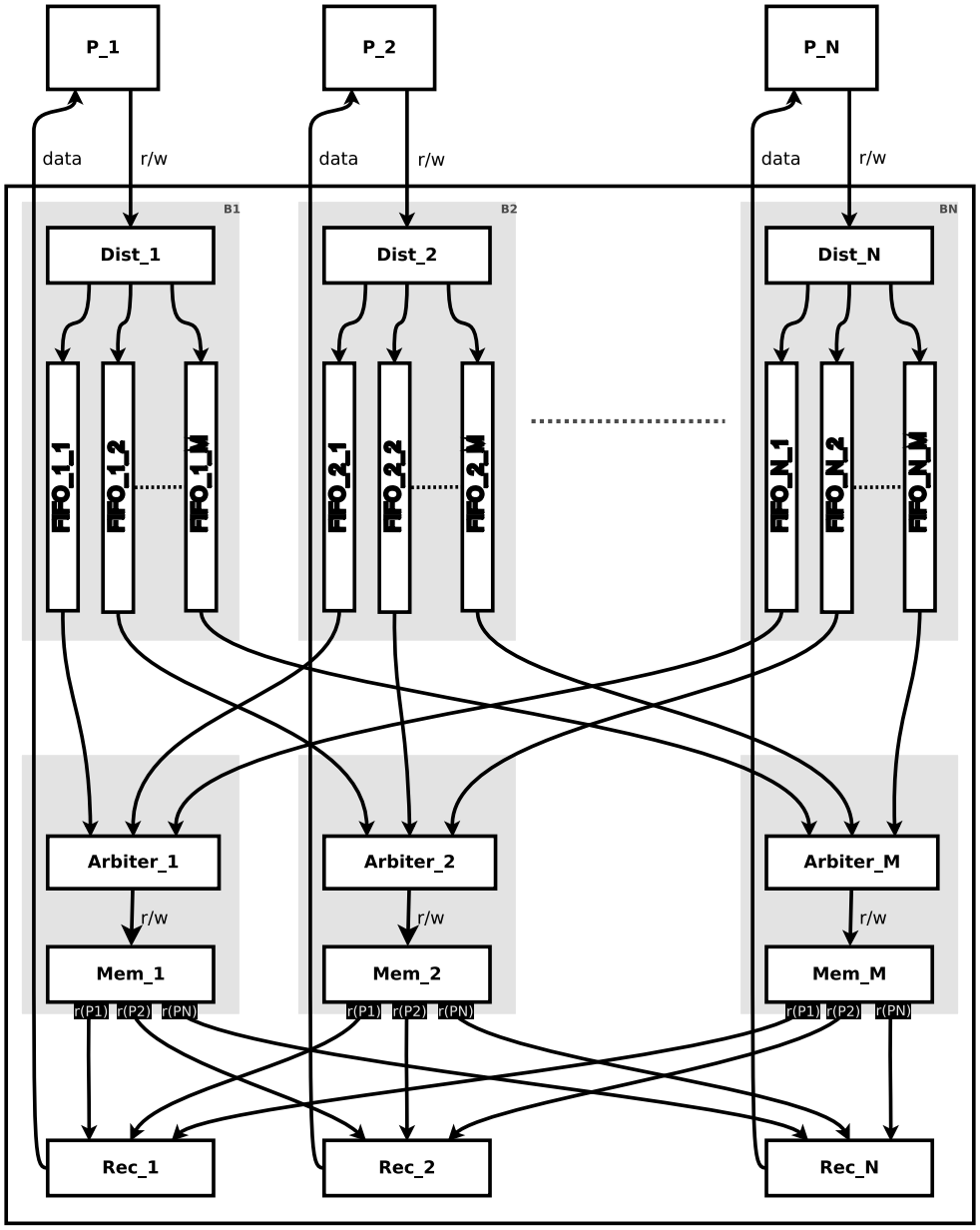

Figure 5.5.: *Reference Machine for Cache consistency.*

5.2.5. Cache consistency (CC)

Architecture: The implementation of the reference machine for CC is shown in Figure 5.5 for a given set $\mathcal{P}$ of n processes. For each process $P_i \in P$, the memory system has a distributor $Dist_i$, a receiver Rec_i, and m different FIFO buffers $FIFO_{i,j}$ for $j \in \{1, \ldots, m\}$. For each memory cell M_j, the memory system provides a memory unit Mem_j and an arbiter $Arbiter_j$. A distributor $Dist_i$ passes the received memory command for memory cell M_j to the corresponding $FIFO_{i,j}$. The arbiters choose non-deterministically from the connected FIFOs to read from. The memory unit returns the result of a read operation to the receiver Rec_i of process P_i. The receiver Rec_i receives reads for its process and returns them to the process' data interface.

Correctness: The use of FIFO buffers ensures by construction that the read and write operations regarding a specific memory location of each process are kept in order (maintains $<_P$ per variable). Therefore, each arbiter $Arbiter_j$ constructs a serial view on all read and write operations regarding its memory location j.

Completeness: Consider now an arbitrary CC execution. If each arbiter selects its action according to the executions' serial view corresponding to its memory location, then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. As no memory operations are lost, and the serial views adhere to the local order, it cannot be the case that the next required value is stuck behind another value in one of the FIFOs. Therefore, each arbiter can idle until eventually the next required operation will be available at the head of one of the connected FIFOs.

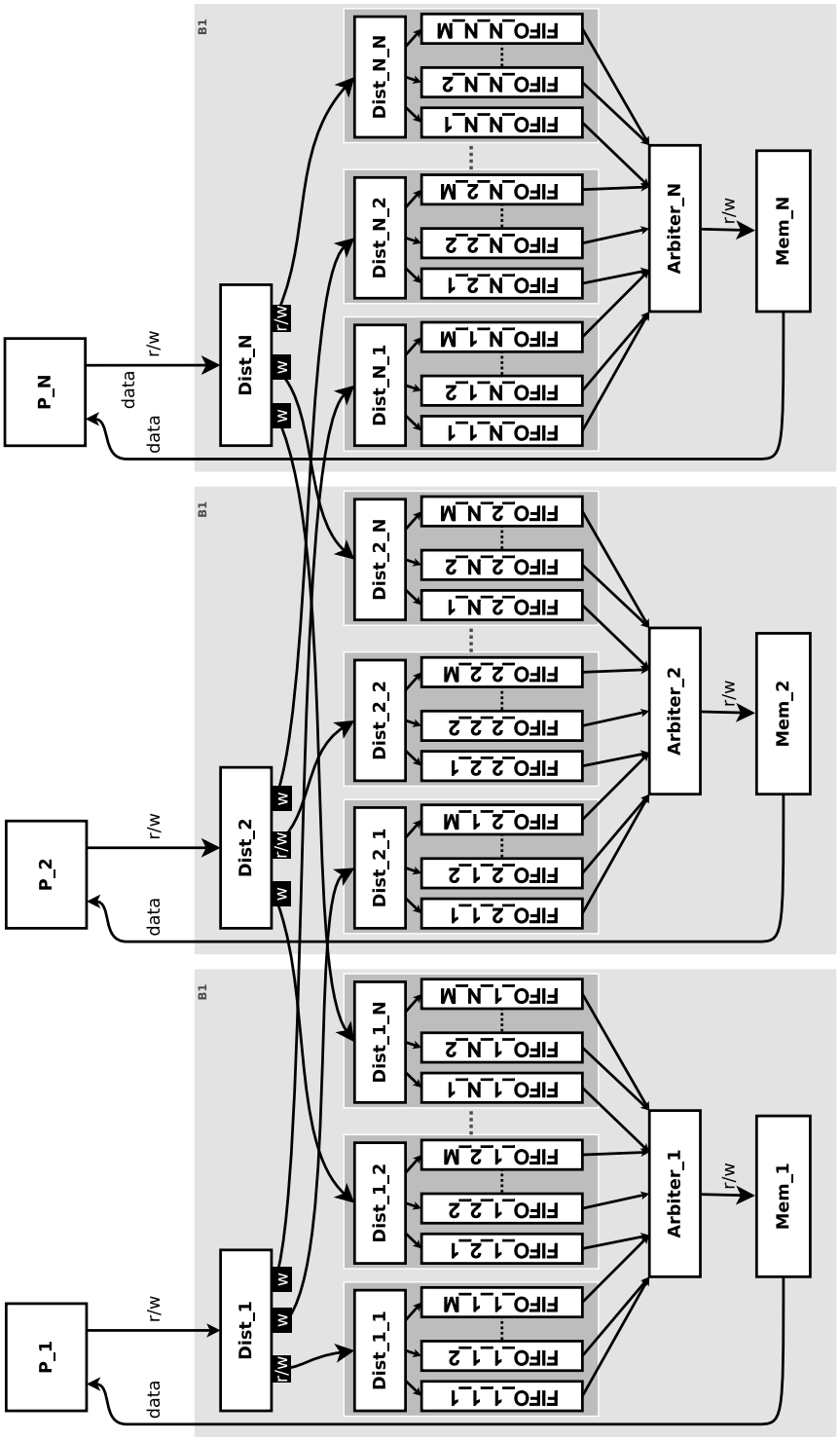

Figure 5.6.: *Reference Machine for Slow consistency.*

5.2.6. Slow consistency (SLOW)

Architecture: The structure of the SLOW reference machine is shown in Figure 5.6 for a given set $\mathcal{P}$ of n processes and m memory locations. For each process $P_i \in \mathcal{P}$, the memory system has a distributor $Dist_i$ and n different distributors $Dist_{i,j}, j \in \{1 \dots n\}$, an arbiter $Arbiter_i$, a memory unit Mem_i and $n \times m$ different FIFO buffers $FIFO_{i,j,k}, j \in \{1 \dots n\}, k \in \{1 \dots m\}$. A distributor $Dist_i$ broadcasts received writes to all corresponding distributors $Dist_{j,i}, j \in \{1 \dots n\}$, and sends all received reads to the distributor $Dist_{i,i}$. A sub-distributor $Dist_{i,j}$ broadcasts received memory commands for memory cell k to the corresponding FIFO buffer $FIFO_{i,j,k}$. The arbiters choose non-deterministically to read from one of the connected FIFOs.

Correctness The distributor fills the FIFOs in the order of the incoming memory operations. Usage of FIFO buffers ensures by construction that the read and write operations of each process are kept in order for each memory location (maintains $<_P |(*, v, *, p, *)_{\mathcal{T}} \cup (w, v, *, *, *)_{\mathcal{T}})$. The arbiter takes elements from the top of a FIFO buffer and issues the operation to the memory unit. Therefore, the memory unit has a serial view on all processes' write operations and the read operations of its corresponding process, and because this property holds for each memory unit, the execution, consisting of $(\mathcal{P}, \mathcal{V}, \mathcal{O}, <_P, \mapsto)$, is SLOW according to Definition 2.13.

Completeness Given a SLOW execution (by Definition 2.13):
$\forall_{p \in \mathcal{P}} \forall_{v \in \mathcal{V}} \exists_{<_{sv}} : <_{sv} = SerialView\,(<_P|\,(*, v, *, p, *)_{\mathcal{T}} \cup (w, v, *, *, *)_{\mathcal{T}})$
By definition serial views exists for each process. Each arbiter can use the serial view of its process as selection order (as it chooses non-deterministically). Then, the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. Therefore, every slowly consistent execution is covered by the reference machine.

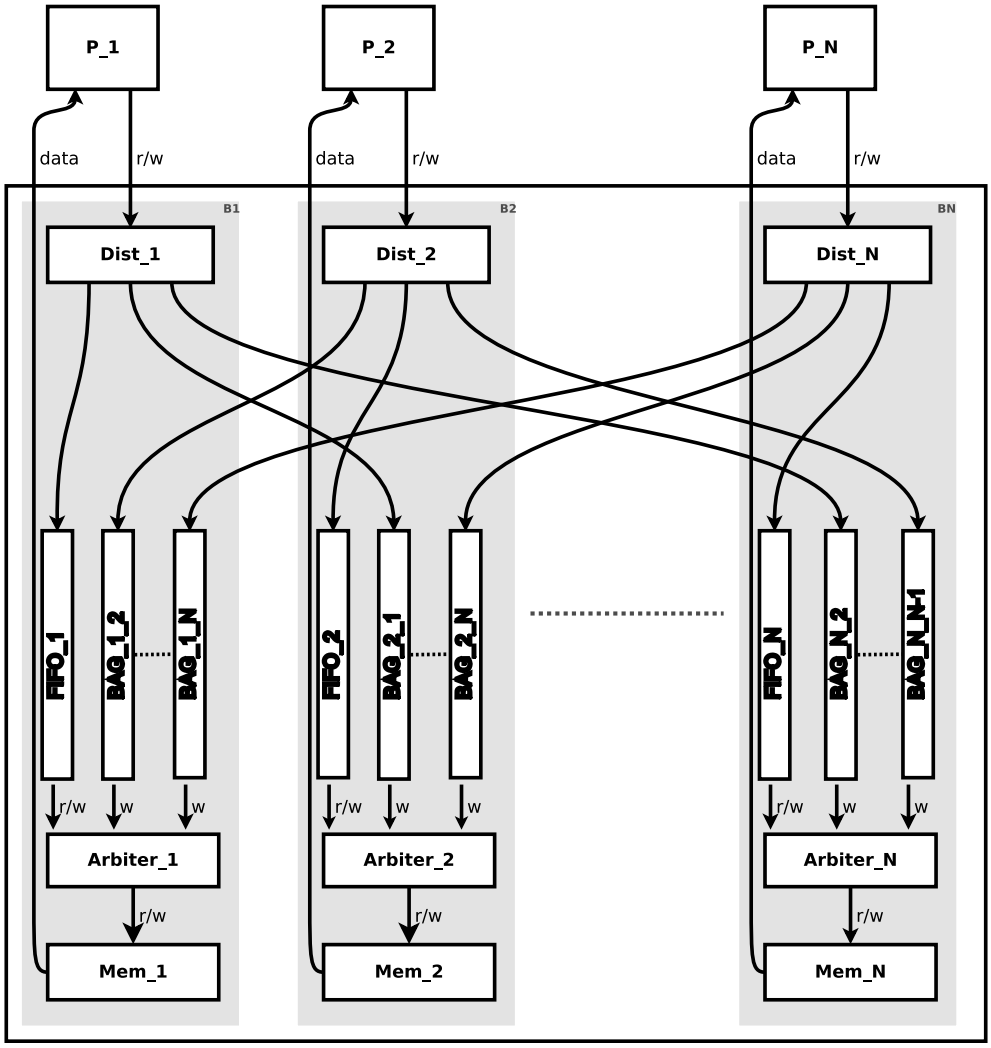

Figure 5.7.: *Reference Machine for Local consistency.*

5.2.7. Local consistency (LOCAL)

Architecture: The implementation of the reference machine for LOCAL is shown in Figure 5.7 for a given set $\mathcal{P}$ of n processes and m memory locations. For each process $P_i \in \mathcal{P}$, the memory system has a distributor $Dist_i$, an arbiter $Arbiter_i$, a memory unit Mem_i, a FIFO buffer $FIFO_i$, and $n-1$ different BAG structures $BAG_{i,j}$ with $j \in \{1, \ldots, n\}, j \neq i$. A distributor $Dist_i$ broadcasts received writes to its $FIFO_i$, and all corresponding $BAG_{j,i}$, $j \in \{1, \ldots, n\}, j \neq i$, and sends all received reads to its $FIFO_i$. The arbiters non-deterministically decide to idle or to non-deterministically read an operation from the connected FIFO and BAG structures. Any operation that is read from the selected FIFO or BAG is forwarded to the memory unit.

Correctness: By construction, a process' own memory operations are kept in order in the FIFO maintaining local order $<_p$. The arbiters generate a serial view covering all own ordered operations and all others' write operations.

Completeness: Given an arbitrary LOCAL execution. According to its definition, a serial view exists for each process. Now, the arbiter can choose to read from the BAG/FIFO structures as the order of the serial view suggests, or to idle as long as the next required value is not yet available. The given architecture allows to wait until the required values are available and therefore covers the required behavior.

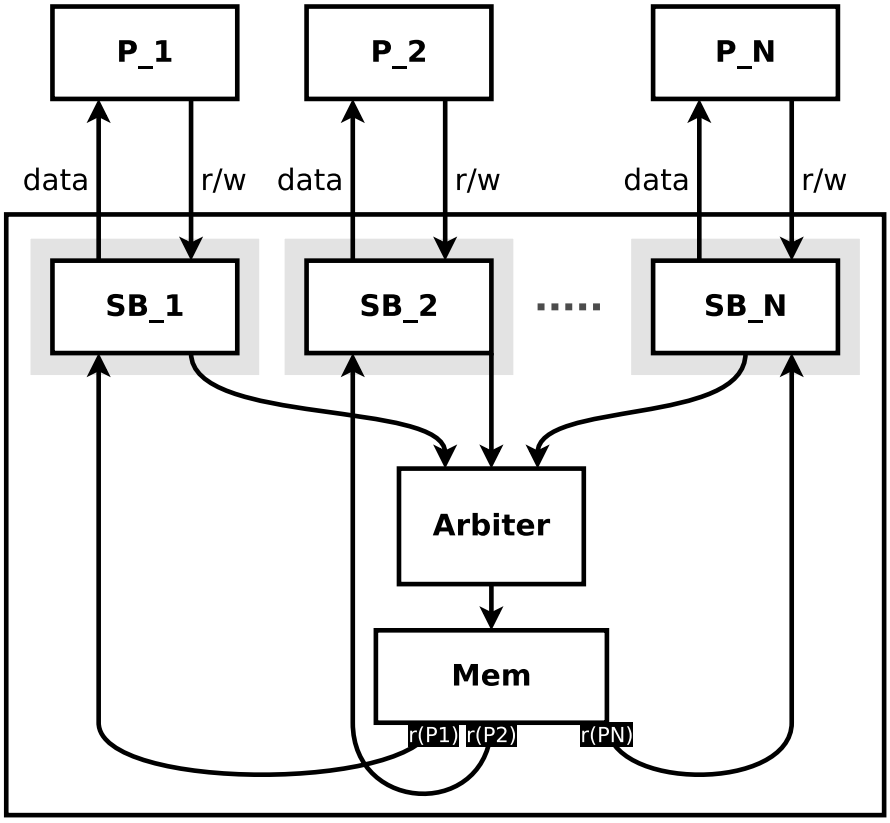

Figure 5.8.: *Reference Machine for Total store ordering.*

5.2.8. Total store ordering (TSO)

Architecture: The structure shown in Figure 5.8 illustrates the reference machine for TSO. The reference machine consists of a store buffer SB_i for each connected process, an arbiter and a memory unit. The store buffers receive operations from the processes and return read results back to them. The arbiter selects non-deterministically between the store buffers and writes the next buffered value back to the main memory unit. If the store buffer wants to issue a read, the read may (chosen non-deterministically) be processed before the next store buffer entry.

Correctness: The reference machine is constructed analogously to the 'TSO Model of Memory' structure as shown in [Spar92, Appendix H]. Therefore, correctness by construction follows.

Completeness: A TSO consistent execution is characterized by a sequence of different events: Either a write is put into the buffer, a write is moved from the head of the buffer to the memory, or a read operation is processed (see Section 2.3.8). As the arbiter can choose non-deterministically to either write back the head of one of the buffers or process a pending read, all execution sequences can be modelled.

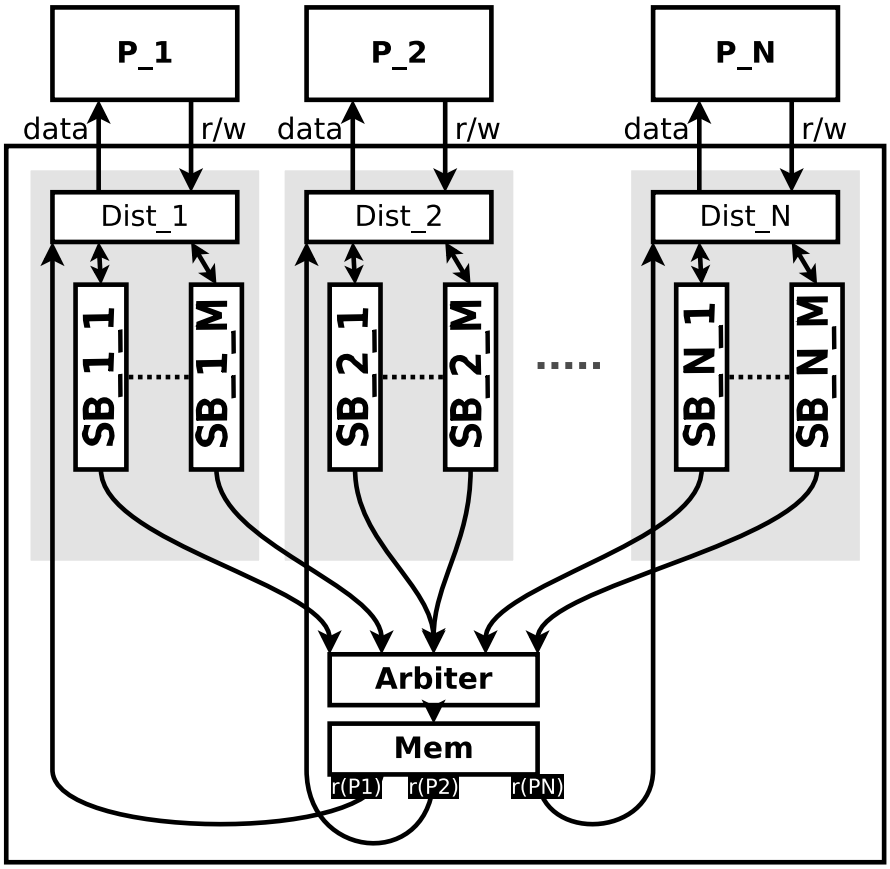

Figure 5.9.: *Reference Machine for Partial store ordering.*

5.2.9. Partial store ordering (PSO)

Architecture: Figure 5.9 illustrates the reference machine for PSO consistency. The reference machine consists of M store buffers $SB_{i,j}$, one for each memory location, and one distributor for each connected process, an arbiter, and a memory unit. The distributors pass the memory operations to the corresponding store buffer. The arbiter selects non-deterministically between the store buffers and writes the next buffered value back to the main memory unit. If the store buffer wants to issue a read, the read may non-deterministically be processed before the next store buffer entry.

Correctness: The reference machine is constructed analogously to the 'PSO Model of Memory' structure as shown in [Spar92, Appendix H]. Therefore, correctness by construction follows.

Completeness: A PSO consistent execution is characterized by a sequence of different events: Either a write is put into the buffer, a write is moved from the head of the buffer to the memory, or a read operation is processed (see Section 2.3.9). As the arbiter can choose non-deterministically to either write back the head of one of the buffers or process a pending read, all execution sequences can be modelled.

5.3. Implementation

All discussed reference machines have been implemented in the synchronous programming language Quartz [Schn09]. The full implementations can be found in the Appendix C and were first published in [Senf13].

The implementations of the reference machines have been tested for correctness with programs like Dekker's mutual exclusion protocol and programs based on memory model litmus test suites.

While the introduced reference machines require unbounded buffers and true non-determinism to guarantee the completeness of the memory model, their implementations in a system description language like Quartz have to specify bounds for such structures. Clearly, since the required buffer sizes can be determined for each finite execution, it is still possible to ensure completeness. For simulation purposes, one can resort to randomizing the non-deterministic choices, and for verification or coverage checking, those have to be handled by oracle inputs which are controlled by the underlying tools.

5.4. Results and Future Work

This chapter presented reference machines to characterize weak memory consistency models in an operational manner. The reference machines have been implemented in the synchronous language Quartz in order to precisely determine their behaviors by the formal semantics of Quartz. All reference machines were implemented by means of some basic components that clearly reflect the intention of the considered memory model. The resulting reference machines are useful for simulation and verification, and can serve as a comprehensive specification that can be used as a programming model.

The correctness and completeness of our reference machines have been shown, i.e., that the reference machines can only perform computations that belong to the weak memory model (correctness), and that all possible computations of the memory model can also be performed by the reference machines (completeness). Hence, the reference machines characterize the memory models in an operational manner.

As stated before, the provided implementations aimed to be both correct and complete following the corresponding definitions. As a result, their structure is more complex than a real implementation would be, including both unboundedness and non-determinism which are impossible in real implementations. Real processor implementation will lack non-determinism but rather have an internal obscure selection procedure.

In future work "reference" machines that are closer to real implementations should be investigated. In fact, a supervised thesis of one of our students started to develop such machines, that are still correct but not necessarily complete anymore.

This thesis observes the behavior of programs developed for sequential machines on weak memory models. Therefore, the memory models are all analyzed and defined without synchronization operations. But, most multicore processors offer synchronization operations for enforcing a desired behavior if needed. Hence, it might be of interest to also include synchronization operations like fences in the proposed architectures.

Chapter 6

Paradigm: State Transition System

Contents

Another approach for verification is exhaustive simulation or where not applicable bounded model checking. A simulation approach requires a well defined state transition system which covers the semantics of a programming language as well as the properties of the memory model. To this end, the semantics of a small programming language MiniC and the weak memory transition rules for memory are provided as structural operational semantics (SOS). In order to cover several memory models at once, the state and especially the memory state are designed in a way that allows to express different

restrictions. The state transition system for distinct weak memory models will then only differ in the rules of the SOS that relate to memory operations. Next, the minimalistic programming language MiniC is introduced.

6.1. MiniC Programming Language

MiniC is a minimalistic programming language developed in our research group and used in education and research [BhSS15]. It is close to C-like languages, but sticks to a small number of instructions and data types. MiniC as used in this thesis features a reasonable complete instruction set, that excludes all sorts of redundant instructions. Undeniably, the familiarity with the language was one of the reasons to chose it over most other known minimalistic languages. But another reason is the required detailed knowledge of languages semantics and compiler decisions regarding memory operations. For example, while a language could ask a value to only be loaded once for an expression, the processor architecture might not have enough registers to hold all required values and therefore has to deviate in its behavior. Furthermore, the evaluation order of an expression is important as well, e.g., if a Boolean conjunction & expression first evaluates both operands or only one operand before handling the operator.

6.1.1. Syntax

$$
\begin{aligned}
\texttt{PROGRAM} &:= [\texttt{DECL}^+]\ \texttt{THREAD}^+ \\
\texttt{THREAD} &:= \texttt{thread ID \{ DECL}^*\ \texttt{STMT \}} \\
\texttt{STMT} &:= \texttt{STMT STMT} \mid \texttt{nothing;} \mid \texttt{ID = EXPR;} \\
&\quad \mid \texttt{if(EXPR) \{STMT\}} \mid \texttt{if(EXPR) \{STMT\} else \{STMT\}} \\
&\quad \mid \texttt{while(EXPR) \{STMT\}} \\
\texttt{EXPR} &:= \texttt{ID} \mid \texttt{VAL} \mid \texttt{!EXPR} \mid \texttt{abs(EXPR)} \\
&\quad \mid \texttt{EXPR} \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{>=}, \texttt{<=}\}\ \texttt{EXPR} \\
&\quad \mid \texttt{EXPR} \{\texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{->}, \texttt{<->}\}\ \texttt{EXPR} \\
&\quad \mid \texttt{EXPR} \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}\ \texttt{EXPR} \\
\texttt{DECL} &:= \texttt{TYPE ID} \\
\texttt{TYPE} &:= \texttt{bool} \mid \texttt{int} \\
\texttt{ID} &:= \{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}, \texttt{\_}\}^+ \\
\texttt{VAL} &:= \texttt{TRUE} \mid \texttt{FALSE} \mid [\texttt{-}]\{\texttt{0}, \ldots, \texttt{9}\}^+
\end{aligned}
$$

Figure 6.1.: *Syntax of the MiniC language. The nonterminals are given on the left-hand side. On the right-hand side, the square brackets stand for optionality, the raised plus sign for one or more times, the braces for choosing one element of many, and the bar for alternatives.*

As can be seen in Figure 6.1, a MiniC program consists of a set of global variables and several threads. A thread can consist of assignments, conditional and loop statements. MiniC is considered to only provide two data types: Boolean and Integer.

6.1.2. Expressions

A MiniC expression is composed of variables, values, and both arithmetic and Boolean expressions (Figure 6.1).

An expression can only be evaluated after all variables have been read from memory. This is technically not required for all operators, e.g., in case of a Boolean conjunction & the evaluation could be finished as soon as one of the operands can be evaluated to FALSE. However, the MiniC semantics expect all variables to be evaluated first to get a deterministic number of load operations for a given expression. Moreover, MiniC semantics require variable instances to be loaded one by one. This design decision is justified by the insights that expressions yield several machine code instructions and depending on the processor architecture and the size of the expression a variable may be loaded several times. Both insights result in additional possible interleavings of parallel processes that should be covered by the semantics.

The 'next' variable which has to be read in order to evaluate an expression is expressed by NextVar(Expr). The function will return $\perp$ if all variables in that expression have been evaluated, otherwise it returns the leftmost variable in the expression tree.

Function NextVar(Expr e)

```
1 switch e do
2 |  case VAL x do ⊥;
3 |  case ID i do i;
4 |  case !e do NextVar(e);
5 |  case abs(e) do NextVar(e);
6 |  case e1 ⊙ e2 do
7 |  |  if NextVar(e1) ≠ ⊥ then NextVar(e1) else NextVar(e2);
8 end
```

To express that a variable's value has been read from memory, we define a substitution on expressions $e\big|_v^x$ that replaces the *first* occurrence of a given variable x with the value v. The *first* occurrence is defined analogously to the NextVar function as the recursively left-most first instance of that variable.

After all variable instances have been replaced by values, i.e., NextVar(e)=$\perp$, the expression can recursively be evaluated to a value. For this purpose, all values in Boolean operators are treated as TRUE if they are not equal to zero, or FALSE otherwise.

When evaluating an expression that has no further variables that need to be read, function Eval(Expr) is used to determine the resulting value the expression evaluates to.

```
Function Eval(Expr e)
1  switch e do
2  |  case TRUE do 1;
3  |  case FALSE do 0;
4  |  case VAL x do x;
5  |  case !e do
6  |  |  if Eval(e) = 0 then 1 else 0;
7  |  case abs(e) do
8  |  |  if Eval(e) < 0 then -Eval(e) else Eval(e);
9  |  case e1 ⊙ e2 | ⊙ ∈ {+, −, ∗, /, %} do
10 |  |  Eval(e1) ⊙ Eval(e2);
11 |  case e1 ⊙ e2 | ⊙ ∈ {==, ! =, <, >, >=, <=} do
12 |  |  if Eval(e1) ⊙ Eval(e2) then 1 else 0;
13 |  case e1 ⊙ e2 | ⊙ ∈ {&, |, ' − >, < − >} do
14 |  |  if (Eval(e1) ≠ 0) ⊙ (Eval(e2) ≠ 0) then 1 else 0;
15 end
```

6.2. System State

The state transition system has to describe the semantics of the programming language as well as the behavior of the memory system depending on the memory consistency model in terms of state transitions. The proposed system defines a state as the combination of a statement and the internal variable state per process as well as a global memory state.

Without loss of generality, it can be assumed that each variable has a unique name, e.g., by prepending variables with a process specific prefix. The variable state of all variables can be expressed as one environment function $\mathcal{E} : \mathtt{ID} \to \mathbb{Z}$ that covers local and global variables. In the initial state, the environment $\mathcal{E}$ returns 0 for all variables, i.e., $\forall_{v \in \mathcal{V}}\ \mathcal{E}(v) \to 0$. Following that, a state is composed of the environment and a statement for each process.

$$\Sigma = \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_i, \ldots, \mathcal{S}_n \rangle$$

In the initial state, the processes' statements correspond to the full input program, i.e., the statement part of the particular threads.

In the following, the data structure for the environment is introduced.

6.2.1. Environment: Memory State

The environment $\mathcal{E}$ that maps each declared variable to an integer value is divided in an environment $\mathcal{E}_l$ for local variables and another environment $\mathcal{E}_g$ for global variables. A local variable is exclusive to a single process and as stated before if multiple processes share the same variable names for local variables the names can always be made exclusive by prepending them with a process specific prefix. Therefore, the set of variables $\mathcal{V}$ can be split in the two distinct subsets $\mathcal{V}_l$ for local variables and $\mathcal{V}_g$ for global variables.

The local environment $\mathcal{E}_l : \mathcal{V}_l \to \mathbb{Z}$ only stores the last written value and can be implemented as a simple array (Figure 6.2) as it does not depend on the memory model.

x	y	z
3	0	2

Figure 6.2.: *Structure: State representation of the local environment as an array. Each process will always read its latest write for a local variable. Therefore, there is no need to store any additional information.*

Global variables on the other hand may hold different values for different processes at a given time. The environment $\mathcal{E}_g : x, p \to \mathbb{Z}$, that returns a variable $x \in \mathcal{V}_g$ current value for a given process $p \in \mathcal{P}$, is expected to return values which are consistent to a given memory model.

Most of the time, a memory system should at least be CC to feasible be used for multithreaded software. This assumption is based on the fact that all model that are shown to be convergent, i.e., eventually offer the same view after idling long enough, are shown to be at least CC consistent [MMSG16]. Therefore, we propose a state representation for global locations that is suitable to express different memory models that are at least CC. Figure 6.3 shows a state representation that maps each location to a queue of written values. An initial state would return the value zero for each location. Furthermore, the state representation has pointers for each process that indicate their *progress* in the queue. If a process issues a read operation it will return the value at the position indicated by its pointer.

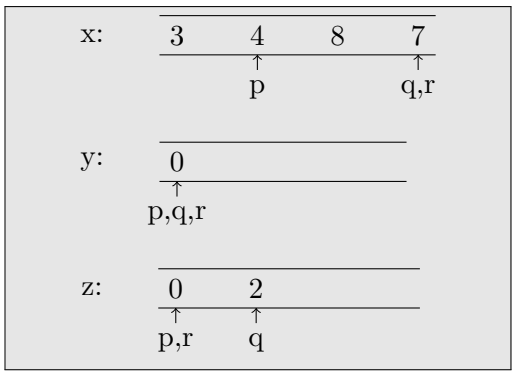

Figure 6.3.: *Structure: State representation of the global environment using queues and pointers. Processes may not yet observe the same values. Therefore, a history of values and some additional information may have to be stored.*

In the given example state in Figure 6.3, the current value of x for process p would be 4 ($\mathcal{E}_g(x, p) = 4$), while q and r would read 7 ($\mathcal{E}_g(x, q) = 7 = \mathcal{E}_g(x, r)$). If no more writes to x occur, then the value 3 will not be readable anymore, while process p may progress to read 8 or 7 in subsequent states.

6.3. Structural Operational Semantics

In the following, the control flow of MiniC will be outlined. To this end, a set of SOS rules will be given that define the MiniC semantics.

6.3.1. Parallelism

As explained in the following, it is sufficient to allow only alternating execution of processes and to not consider true parallel execution.

If processes do not access global variables they can not influence one another, then it is of no difference if one of them is executed before the other or both at the same time. In the first case we would get an intermediate state Σ_a or Σ_b followed by the concluding Σ_{ab}, in the later case only Σ_{ab} (Figure 6.4).

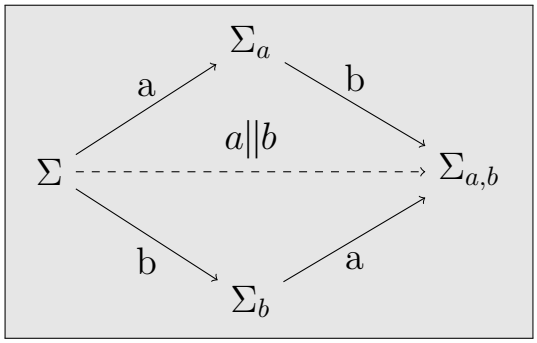

Figure 6.4.: *Small step parallel execution without global variables*

For global variables, depending on the implementation, a parallel access on the same location could result in undefined results. MiniC expects the memory system to order memory operations even if they are executed in parallel. This means if two values are written to the same location the resulting state should be the same as if one of the values was written after the other (Figure 6.5).

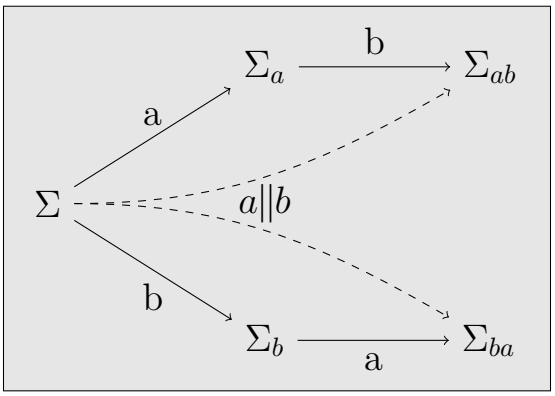

Figure 6.5.: *Small step parallel execution with global variables*

Following that, the resulting states are the same if the memory actions were executed one after the other. Therefore, only allowing interleaved executions

does not remove any states from the reachable state space. This allows for shorter SOS rules, as they can be limited to the effect of one statement at a time.

6.3.2. Atomic statements

- **Nothing:** A preceding nothing statement in a sequence statement can be dropped.

$$\frac{\varnothing}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{nothing; } \mathcal{S}_a, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_a, \ldots, \mathcal{S}_n \rangle}$$

6.3.3. Control-flow statements

- **Sequence:** The evaluation of the left part of a sequence is independent of its right part.

$$\frac{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_a, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_a\texttt{'}, \ldots, \mathcal{S}_n \rangle}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_a\texttt{; } \mathcal{S}_b, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_a\texttt{ ; } \mathcal{S}_b, \ldots, \mathcal{S}_n \rangle}$$

- **If-Then:** An If-Then statement can be expressed with an Else part that contains a Nothing statement.

$$\frac{\varnothing}{\begin{array}{l}\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(c)\{ } \mathcal{S}\texttt{\}}, \ldots, \mathcal{S}_n \rangle \\ \mapsto \\ \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(c)\{ } \mathcal{S}\texttt{\}else\{ nothing; \}}, \ldots, \mathcal{S}_n \rangle\end{array}}$$

- **If-Then-Else:** An If-Then-Else statement with an evaluated condition can be reduced to either its Then or Else statement.

$$\frac{c \in \mathbb{Z} \;\wedge\; c \neq 0}{\begin{array}{l}\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(c)\{ } \mathcal{S}_a\texttt{\} else \{ } \mathcal{S}_b\texttt{\}}, \ldots, \mathcal{S}_n \rangle \\ \mapsto \\ \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_a, \ldots, \mathcal{S}_n \rangle\end{array}}$$

$$\frac{\varnothing}{\begin{array}{l}\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(0) \{ } \mathcal{S}_a\texttt{\} else \{ } \mathcal{S}_b\texttt{\}}, \ldots, \mathcal{S}_n \rangle \\ \mapsto \\ \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_b, \ldots, \mathcal{S}_n \rangle\end{array}}$$

- **While:** A While statement can be recursively evaluated using If-Then statements.

$$\frac{\varnothing}{\begin{array}{l}\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{while(c)\{ } \mathcal{S}\texttt{\}}, \ldots, \mathcal{S}_n \rangle \\ \mapsto \\ \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(c)\{ } \mathcal{S}\texttt{; while(c)\{ } \mathcal{S}\texttt{\}\}}, \ldots, \mathcal{S}_n \rangle\end{array}}$$

6.3.4. Expression Evaluation

- **If-Then-Else:** If the condition of a If-Then-Else statement does not contain any variables, then it can be evaluated to an Integer value using the *Eval* function.

$$\frac{\text{NextVar}(e) = \bot}{\begin{array}{l}\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(e)\{ } \mathcal{S}_a \texttt{\} else \{ } \mathcal{S}_b \texttt{\}}, \ldots, \mathcal{S}_n \rangle \\ \mapsto \\ \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{if(Eval(e))\{ } \mathcal{S}_a \texttt{\} else \{ } \mathcal{S}_b \texttt{\}}, \ldots, \mathcal{S}_n \rangle\end{array}}$$

- **Assignment:** Analogously, if the righthand side of an Assignment statement does not contain any more variables, then it can be evaluated to an Integer value using the *Eval* function.

$$\frac{\text{NextVar}(e) = \bot}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{v=e;}, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{v=Eval(e);}, \ldots, \mathcal{S}_n \rangle}$$

6.3.5. Memory Reads

Memory reads only consider the current state as the process pointers always determine the current view of a processor for each memory location. Also, they only change a processes state and do not influence the current memory state. Hence, read operations are independent of the memory model.

With the rules introduced so far, only two situations remain that require to evaluate an expression and therefore may require a read: Either the evaluation of the condition of a conditional statement or the right side of an assignment.

If there exists a variable in the expression e of an If-Then-Else or assignment statement, i.e., $\text{NextVar}(e) \neq \bot$, then there exists a state transition that replaces the variable in the expression by the current value of the environment for the given process $\mathcal{E}(y, i) = v$.

- **Reading If-Then-Else condition:** The first occurring variable in the condition of an If-Then-Else statement can be substituted by the environment's current value of that variable for the selected process.

$$\frac{\text{NextVar}(e) = v \neq \bot \quad \wedge \quad \mathcal{E}(v, p) = d \quad \wedge \quad e' = e\big|_d^v}{\begin{array}{l}\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_p : \texttt{if(e)\{ } \mathcal{S}_a \texttt{ \} else \{} \mathcal{S}_b \texttt{\}}, \ldots, \mathcal{S}_n \rangle \\ \mapsto \\ \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}'_p : \texttt{if(e')\{} \mathcal{S}_a \texttt{\} else \{} \mathcal{S}_b \texttt{\}}, \ldots, \mathcal{S}_n \rangle\end{array}}$$

- **Reading Assignment expression:** Analogously, the first occurring variable of the righthand side of an Assignment statement can be substituted by the environment's current value of that variable for the selected process.

$$\frac{\text{NextVar}(e) = v \neq \bot \quad \wedge \quad \mathcal{E}(v, p) = d \quad \wedge \quad e' = e\big|_d^v}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}_p : \texttt{x = e}, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \mathcal{S}'_p : \texttt{x = e'}, \ldots, \mathcal{S}_n \rangle}$$

6.3.6. Local Memory Update

While the local memory updates are independent of the memory model, the global memory updates may be different for each memory model.

Let $Upd(\mathcal{E}, v, d)$ be the environment $\mathcal{E}$ after the value of the local variable v was replaced by data value d. Then, a write to a local variable can be expressed with the following rule:

$$\frac{d \in \mathbb{Z} \quad \wedge \quad v \in \mathcal{V}_l \quad \wedge \quad \mathcal{E}' = Upd(\mathcal{E}, v, d)}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{v=d;}, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, \texttt{nothing;}, \ldots, \mathcal{S}_n \rangle}$$

6.3.7. Global Memory Update

The global environment however can change due to insertion of a new value in a queue or by incrementing one of the pointers. To express these transitions as SOS rules some additional functions are defined as follows.

- $Ins(\mathcal{E}, v, d, i)$ depicts the environment $\mathcal{E}$ after inserting a new value d at position i of queue v.
- $Len(\mathcal{E}, v)$ is the length of the queue for v.
- $Pos(\mathcal{E}, p, v)$ is the current numerical position of process p's pointer in the queue of v where the first position has index 0.
- $Mov(\mathcal{E}, p, v, i)$ is the environment $\mathcal{E}$ after the pointer of p in queue v was moved to position i.
- $MovAll(\mathcal{E}, v, i)$ is the environment $\mathcal{E}$ after all pointers of queue v which are at a position before i were moved to position i.

Each memory model defines restrictions how a write may be inserted in the corresponding queue. The weakest rules would allow to insert a write at every possible position and pointers to be moved arbitrarily.

$$\frac{d \in \mathbb{Z} \quad \wedge \quad v \in \mathcal{V}_g \quad \wedge \quad 0 < i < Len(\mathcal{E}, v) \quad \wedge \quad \mathcal{E}' = Ins(\mathcal{E}, v, d, i)}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{v=d;}, \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, \texttt{nothing;}, \ldots, \mathcal{S}_n \rangle}$$

$$\frac{v \in \mathcal{V}_g \quad \wedge \quad p \in \mathcal{P} \quad \wedge \quad 0 < c < Len(\mathcal{E}, v) \quad \wedge \quad \mathcal{E}' = Mov(\mathcal{E}, p, d, c)}{\langle \mathcal{E}, (\mathcal{S}_1, \ldots, \mathcal{S}_n) \rangle \mapsto \langle \mathcal{E}', (\mathcal{S}_1, \ldots, \mathcal{S}_n) \rangle}$$

The only guarantee these rules could provide would be that the only values that can be read are values that have been written at some time before. Obviously, these rules do not offer any reasonable restrictions and only serve a exemplary purpose.

Depending on the memory model both the insertion of writes as well as the movement of the memory pointers may be restricted in several ways.

6.4. Memory Models

In this section, the SOS rules for the global memory update of different memory models are introduced.

6.4.1. Sequential consistency (SC)

For SC a write adds a value at the end of the queue and moves all pointers to the new value. As all processes have the same pointer locations they have the same view on memory at all time. As the pointers are always at the end of the queue no environment update rule is required for SC.

$$\frac{\begin{array}{c} d \in \mathbb{Z} \quad \wedge \quad v \in \mathcal{V}_g \quad \wedge \quad Len(\mathcal{E}, v) = i \\ \wedge \quad \mathcal{E}' = Ins(MovAll(\mathcal{E}, v, i+1), v, d, i+1) \end{array}}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, \texttt{v=d;}\, , \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, \texttt{nothing;}\, , \ldots, \mathcal{S}_n \rangle}$$

6.4.2. Cache consistency (CC)

For CC a write adds a value to the queue after its process' current position and moves the process' pointer to the new entry.

$$\frac{\begin{array}{c} d \in \mathbb{Z} \quad \wedge \quad v \in \mathcal{V}_g \quad \wedge \quad Pos(\mathcal{E}, p, v) + 1 = i \\ \wedge \quad \mathcal{E}' = Ins(Mov(\mathcal{E}, p, v, i), v, d, i) \end{array}}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, S_p : \texttt{v=d;}\, , \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, S_p : \texttt{nothing;}\, , \ldots, \mathcal{S}_n \rangle}$$

A process may observe newer reads to a specific location but never older ones, i.e., pointers may only be moved forward.

$$\frac{\begin{array}{c} v \in \mathcal{V}_g \quad \wedge \quad p \in \mathcal{P} \quad \wedge \quad Pos(\mathcal{E}, p, v) + 1 = i < Len(\mathcal{E}, v) \\ \wedge \quad \mathcal{E}' = Mov(\mathcal{E}, p, v, i) \end{array}}{\langle \mathcal{E}, (\mathcal{S}_1, \ldots, \mathcal{S}_n) \rangle \mapsto \langle \mathcal{E}', (\mathcal{S}_1, \ldots, \mathcal{S}_n) \rangle}$$

6.4.3. Partial store ordering (PSO)

The environment for PSO consistency allows to insert a write at any following position in the corresponding queue. The pointer of the writing process will be set to the new entry.

$$\frac{\begin{array}{c} d \in \mathbb{Z} \quad \wedge \quad v \in \mathcal{V}_g \quad \wedge \quad Pos(\mathcal{E}, p, v) < i \leq Len(\mathcal{E}, x) + 1 \\ \wedge \quad \mathcal{E}' = Ins(Mov(\mathcal{E}, p, v, i), v, d, i) \end{array}}{\langle \mathcal{E} \,|\, \mathcal{S}_1, \ldots, S_p : \texttt{v=d;}\, , \ldots, \mathcal{S}_n \rangle \mapsto \langle \mathcal{E}' \,|\, \mathcal{S}_1, \ldots, S_p : \texttt{nothing;}\, , \ldots, \mathcal{S}_n \rangle}$$

Flush: Select a position, move all pointers that are lower than that to that position.

$$\frac{0 < i < Len(\mathcal{E}, v) \quad \wedge \quad \mathcal{E}' = MovAll(\mathcal{E}, v, i)}{\langle \mathcal{E}, (\mathcal{S}_1, \ldots, \mathcal{S}_n) \rangle \mapsto \langle \mathcal{E}', (\mathcal{S}_1, \ldots, \mathcal{S}_n) \rangle}$$

The described environment actually models PSO as outlined in the following. A PSO execution can be described by two different types of events: a write event $write(p, x, v)$ that stores the written value in a buffer which is specific to the given variable and process, and a flush event $flush(p, x, v)$ that

takes the oldest value out of one of the buffers and writes the value to memory. A PSO consistent execution consists of a sequence of such correlating events: a write will eventually be followed by the corresponding flush.

6.4.4. Other models

Some memory models like PRAM or TSO require to track the order among memory operations of a single process that affect multiple variables. As the proposed structure separates all operations by location, it can not be used to express these memory models as it is.

By enhancing the queue entries with the originating process and either an increasing write id or a pointer to the previous write these models could be expressed as well. However, these modifications would be specific to single models and there already exist representations that are optimized for specific memory models, e.g., [Cali16] for TSO.

6.5. Results and Future Work

This section introduced a state transition system for programs written in MiniC that complies with a given weak memory model. The reachable state space models the set of traces that are valid with respect to a given memory consistency model.

Consequently, the state transition system can be implemented as a tool for verification of MiniC programs under weak memory. Depending on the verification goal, the state space exploration might use state approximation to reduce the state space.

The state size could be reduced dynamically. In the given representation the queues will keep old values even though they might never be referenced any more. In order to save space, an implementation could adapt the representation and rules to drop an element if all pointers have moved past it.

The chosen representation was a compromise of structural simplicity and coverage of memory models. Meaning, there exist representations that require less memory space, but therefore can only cover specific memory models. For example, SC can be modelled with a global environment that just stores the latest value. On the other hand, storing more meta data, e.g., the source of a write and its issue index, would allow the coverage of more models but increase both the size of each state as well potentially increasing the state space as well.

Chapter 7

Conclusions

The multitude of weak memory models calls for techniques to uniformly address different models for verification of multithreaded programs.

To this end, this thesis introduced a multitude of different paradigms to express various aspects of weak memory consistency. Each of the introduced paradigms allows to inspect several memory models. Hence, making it possible to reflect properties between different models, for example safety properties in programs that do not hold in weaker models. Each of the proposed approaches has its own benefits and drawbacks as summarized in Figure 7.1.

	coverage	**simulation**	**state**	**state space**
SAT	trace	no	-	-
LTL	program	no	min	full
OpS	program	yes	full	optimized
STS	program	yes	small	optimized

Figure 7.1.: *Comparison of the introduced modeling approaches.*

First, the testing problem, that asks whether a given trace is valid with respect to a memory model, is analyzed. Besides important complexity results, an uniform encoding as SAT problem is developed. The encoding directly reflects the constraints defined by the view-based definition of the memory models. Due to the similar definitions of the memory models, the encoding is able to reuse the same encoding principle parameterized for different models. The resulting SAT encodings have been encoded for and solved with the Z3 SMT solver for some examples. The biggest drawback is the restricted set of applications. While there exist relevant applications that need to analyze traces, most applications require an examination of all possible traces that can not be provided this way.

The second introduced approach uses linear temporal logic to describe the constraints of a memory consistency model. The constraints are described in context of memory read and write events as well as auxiliary observation events. Based on the history of events the LTL specification determines the result values of read events. The LTL specification is composed of a set of

LTL formulas where in general stronger models have to incorporate more constraints than weaker models. For verification purposes of safety properties, the program is encoded as several parallel modules that communicate using a memory interface. Using NuSMV or NuXMV the set of traces is checked for valid traces for the given memory model that do not satisfy the safety property. However, the approach does not limit read results to reasonable values from the start but only checks if they are valid afterwards. Likewise, every possible write and auxiliary event has to be covered only to be dismissed by the specification. Therefore, the approach suffers from the state explosion problem, requiring a huge amount of memory. Only very small examples were considered in the experiments as larger examples quickly stressed out the testing environment. While the pursued verification approach proved unsuitable, the encoding of weak memory consistency properties in LTL is a novel concept that might be suitable for other tools and verification approaches.

While working on the teaching processor architectures of our chair, the idea emerged to develop operational semantics for weak memories as well. Like the preceding approach, the processors communicate with the memory system using a well defined interface for read and write actions. The memory systems are composed of common basic components allowing to compare and compose different memory models. To actually coincide with the definition of the weak memory models, the corresponding memory systems have to be both correct and complete. While correctness is simply achieved by restricting the internals to reflect only valid traces, the completeness requires to capture every possible trace. On that account, all of its decisions are based on non-deterministic oracles and all dynamic structures like buffers are considered unbounded. However, in practice implementations will not offer completeness, but bounds can be chosen adequately for given examples. The resulting reference machines can be used for simulation and verification purposes either using randomized values or oracles for the non-deterministic choices.

The only drawback with the operational semantics was that they were implemented on a bit-level close to hardware similar to the related processor architecture. While this approach is well suited for teaching, most processor details and bit-level specifics could be abstracted for verification purposes. Following that, an approach is proposed that is tailored for verification based on state exploration. The approach consists of a state transition system that covers the semantics of a minimal C-like language in presence of different weak memory consistency models. To this end, the semantics of the programming language is given as a set of SOS rules and a state structure is proposed that allows to represent different memory models: SC, PSO, and CC. The actual memory semantics for the different memory models are given as additional SOS rules that use the state structure to model the observations of the different processes. By now, the proposed state transition system has not been implemented yet.

7.1. Future Works

With newer processor architectures there will be more memory consistency models making it even more important to understand the subtle differences on how they influence the expected behavior of multithreaded programs.

There remains work to be done as listed in the following:

- More models should be expressed using the introduced representations. For example the SPARC and PowerPC memory models as well as the x86 TSO implementation deserve more work due to their prevalence.
- For verification using the proposed temporal logics representation, there might be a better approach to check the correctness of a safety property. The proposed approach constructs the full state space and restricts it using the specification afterwards. It remains to search if there exists an approach that could derive legitimate states directly from specification.
- The operational semantics are quite complex due to their aim for completeness. Some work in the direction of more realistic and therefore not complete memory system implementations was already started. The resulting reference machines look promising but comparisons of the different implementations are not yet available.
- The state transition system proposed in the last chapter should be implemented in a suitable programming language. Later, with a first implementation up and running, approximations based on the memory state could be considered to decrease the required memory for the state exploration.

Bibliography

[AlCM16] J. Alglave, P. Cousot, and L. Maranget. *Syntax and semantics of the weak consistency model specification language cat.* arXiv Report arXiv:1608.07531. Cornell University Library, 2016.

[AdGh96] S.V. Adve and K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial". In: *IEEE Computer* 29.12 (1996), pp. 66–76.

[AdHi93] S.V. Adve and M.D. Hill. "A unified formalization of four shared-memory models". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 4.6 (1993), pp. 613–624.

[ABHN91] M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neigher. "Causal Memory". In: *International Workshop on Distributed Algorithms*. Ed. by S. Toueg, P.G. Spirakis, and L.M. Kirousis. Vol. 579. LNCS. Delphi, Greece: Springer, 1991, pp. 9–30.

[ABJK93] M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger. "The power of processor consistency". In: *Symposium on Parallel Algorithms and Architectures (SPAA)*. Ed. by L. Snyder. Velen, Germany: ACM, 1993, pp. 251–260.

[Algl10] J. Alglave. "A Shared Memory Poetics". PhD. PhD thesis. Paris, France: Université Paris 7-Denis Diderot, 2010.

[Algl12] J. Alglave. "A formal hierarchy of weak memory models". In: *Formal Methods in System Design (FMSD)* 41.2 (2012), pp. 178–210.

[BaBe97a] J. Bataller and J.M. Bernabéu-Aubán. "Synchronized DSM Models". In: *International Euro-Par Conference (Euro-Par)*. Ed. by C. Lengauer, M. Griebl, and S. Gorlatch. Vol. 1300. LNCS. Passau, Germany: Springer, 1997, pp. 468–475.

[BaFT17] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard*. 2017.

[BaKa08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. Cambridge, Massachusetts, USA: MIT Press, 2008.

[BrMo17] R. Bruni and U. Montanari. *Models of Computation*. Texts in Theoretical Computer Science. Springer, 2017.

[BoPe09] G. Boudol and G. Petri. "Relaxed memory models: an operational approach". In: *Principles of Programming Languages (POPL)*. Ed. by Z. Shao and B.C. Pierce. Savannah, Georgia, USA: ACM, 2009, pp. 392–403.

[BhSS15] N. Bhardwaj, M. Senftleben, and K. Schneider. "Abacus – A Processor Family for Education". In: *Workshop on Embedded and Cyber-Physical Systems Education (WESE)*. Ed. by M. Törngren and M.E. Grimheden. New Delhi, India: ACM, 2015, 2:1–2:8.

[Cali16] G.I. Calin. "Verification Techniques for TSO-Relaxed Programs". PhD. PhD thesis. Department of Computer Science, University of Kaiserslautern, 2016.

[CCDG14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. "The nuXmv Symbolic Model Checker". In: *Computer Aided Verification (CAV)*. Ed. by A. Biere and R. Bloem. Vol. 8559. LNCS. Vienna, Austria: Springer, 2014, pp. 334–342.

[ClGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. "NUSMV: A New Symbolic Model Verifier". In: *Computer Aided Verification (CAV)*. Ed. by N. Halbwachs and D. Peled. Vol. 1633. LNCS. Trento, Italy: Springer, 1999, pp. 495–499.

[CaLS05a] J.F. Cantin, M.H. Lipasti, and J.E. Smith. "The Complexity of Verifying Memory Coherence and Consistency". In: *IEEE Transactions on Parallel and Distributed Systems* 16.7 (2005), pp. 663–671.

[Dijk68b] E.W. Dijkstra. "Cooperating sequential processes". In: *Programming Languages: NATO Advanced Study Institute*. Ed. by F. Genuys. Academic Press, 1968, pp. 43–112.

[Emer90] E.A. Emerson. "Temporal and Modal Logic". In: *Handbook of Theoretical Computer Science*. Ed. by J. van Leeuwen. Vol. B: Formal Models and Semantics. Elsevier, 1990. Chap. 16, pp. 995–1072.

[FGPS16] S. Flur, K.E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. "Modelling the ARMv8 architecture, operationally: concurrency and ISA". In: *Principles of Programming Languages (POPL)*. Ed. by R. Bodik and R. Majumdar. St. Petersburg, FL, USA: ACM, 2016, pp. 608–621.

[FMSS14] F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. "Memory Model-Aware Testing - A Unified Complexity Analysis". In: *Application of Concurrency to System Design (ACSD)*. Tunis La Marsa, Tunisia: IEEE Computer Society, 2014, pp. 92–101.

[FMSS15] F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. "Memory-Model-aware Testing – A Unified Complexity Analysis". In: *Transactions on Embedded Computing Systems (TECS)* 14.4 (2015), 63:1–63:25.

[GLLG90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors". In: *International Symposium on Computer Architecture (ISCA)*. Seattle, Washington, USA: IEEE Computer Society, 1990, pp. 15–26.

[GiKo97] P.B. Gibbons and E. Korach. "Testing Shared Memories". In: *SIAM Journal on Computing* 26.4 (1997), pp. 1208–1244.

[Good91a] J.R. Goodman. *Cache consistency and sequential consistency.* Technical Report 1006. Computer Sciences Department, University of Wisconsin-Madison, Feb. 1991.

[HuAh90] P.W. Hutto and M. Ahamad. "Slow Memory: Weakening Consistency to Enchance Concurrency in Distributed Shared Memories". In: *International Conference on Distributed Computing Systems (ICDCS)*. Paris, France: IEEE Computer Society, 1990, pp. 302–309.

[HiKV98] L. Higham, J. Kawash, and N. Verwaal. *Weak memory consistency models – Part I: Definitions and Comparisons.* Technical Report 98/612/03. Department of Computer Science, University of Calgary, 1998.

[HePa03] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A quantitative Approach.* 3rd ed. Morgan Kaufmann, 2003.

[HeSi92] A. Heddaya and H. Sinha. *Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing.* Technical Report BU-CS-92-004. Department of Computer Science, Boston University, 1992.

[Lamp78] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM (CACM)* 21.7 (1978), pp. 558–565.

[Lamp79] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers (T-C)* 28.9 (1979), pp. 690–691.

[Lawr98] R. Lawrence. *A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors.* 1998.

[LLGW92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. "The Stanford Dash Multiprocessor". In: *IEEE Computer* 25.3 (1992), pp. 63–79.

[LCCM06] P. Loewenstein, S. Chaudhry, R. Cypher, and C. Manovit. "Multiprocessor memory model verification". Automated Formal Methods Workshop (AFM). 2006.

[LiSa88] R.J. Lipton and J.S. Sandberg. *PRAM: A Scalable Shared Memory*. Technical Report CS-TR-180-88. Princeton University, 1988.

[Lust15] D. Lustig. "Specifying, Verifying, and Translating Between Memory Consistency Models". PhD. PhD thesis. Princeton University, USA, 2015.

[MMSM12] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M.M.K. Martin, P. Sewell, and D. Williams. "An Axiomatic Memory Model for POWER Multiprocessors". In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and S.A. Seshia. Vol. 7358. LNCS. Berkeley, California, USA: Springer, 2012, pp. 495–512.

[MoBj08a] L. Mendonc ca de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by C.R. Ramakrishnan and J. Rehof. Vol. 4963. LNCS. Budapest, Hungary: Springer, 2008, pp. 337–340.

[McKe10] P.E. McKenney. *Memory Barriers: A Hardware View for Software Hackers*. http://www.rdrop.com/users/paulmck. 2010.

[Mire14] L.P. Miret. "Consistency models in modern distributed systems – An approach to Eventual Consistency". Master. MA thesis. Universitat Politècnica de València, Spain, 2014.

[Mosb93] D. Mosberger. "Memory consistency models". In: *ACM SIGOPS: Operating Systems Review* 27.1 (1993), pp. 18–26.

[Mosb93a] D. Mosberger. *Memory Consistency Models*. Technical Report TR 93/11. Tucson, Arizona, USA: Department of Computer Science, The University of Arizona, 1993.

[MaPn92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[MMSG16] F.D. Munoz-Escoí, J.R.G. de Mendívil, J.S. Sendra-Roig, J.-R. García-Escrivá, and J.M. Bernabéu-Aubán. *Notes on Eventual Consistency*. Technical Report TR-IUMTI-SIDI-2016/002. València, Spain: Instituto Universitario Mixto Tecnológico de Informática Universitat Politècnica de València, 2016.

[NSSS11] F.Z. Nardelli, P. Sewell, J. Ševčík, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave. "Relaxed memory models must be rigorous". In: *Exploiting Concurrency Efficiently and Correctly (EC2)*. Snowbird, Utah, USA, 2011.

[OwSS09] S. Owens, S. Sarkar, and P. Sewell. "A Better x86 Memory Model: x86-TSO". In: *Theorem Proving in Higher Order Logics (TPHOL)*. Ed. by S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel. Vol. 5674. LNCS. Munich, Germany: Springer, 2009, pp. 391–407.

[Pete81] G.L. Peterson. “Myths about the Mutual Exclusion Problem”. In: *Information Processing Letters* 12.3 (1981), pp. 115–116.

[LFHM17] H. Ponce de León, F. Furbach, K. Heljanko, and R. Meyer. “Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models”. In: *Static Analysis Symposium (SAS)*. Ed. by F. Ranzato. Vol. 10422. LNCS. New York, NY, USA: Springer, 2017, pp. 299–320.

[Pugh00] W. Pugh. “The Java memory model is fatally flawed”. In: *Concurrency: Practice and Experience* 12.6 (2000), pp. 445–455.

[Schn03] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[Schn09] K. Schneider. *The Synchronous Programming Language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, 2009.

[Senf13] M. Senftleben. “Operational Characterization of Weak Memory Consistency Models”. Master. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, 2013.

[SiFC92] P.S. Sindhu, J.-M. Frailong, and M. Cekleov. “Formal Specification of Memory Models”. In: *Scalable Shared Memory Multiprocessors*. Ed. by M. Dubois and S.S. Thakkar. Kluwer, 1992, pp. 25–41.

[StNu04] R.C. Steinke and G.J. Nutt. “A unified theory of shared memory consistency”. In: *Journal of the ACM (JACM)* 51.5 (2004), pp. 800–849.

[Spar92] SPARC International, Inc. *The SPARC Architecture Manual-Version 8*. Prentice-Hall, 1992.

[SeSc16] M. Senftleben and K. Schneider. “Specifying Weak Memory Consistency with Temporal Logic”. In: *Verification and Evaluation of Computer and Communication Systems (VECoS)*. Ed. by M. Ghazel and M. Jmaiel. Vol. 1689. CEUR Workshop Proceedings. http://ceur-ws.org/Vol-1689/. Tunis, Tunisia: Sun SITE Central Europe, 2016, pp. 107–122.

[SeSc18] M. Senftleben and K. Schneider. “Operational Characterization of Weak Memory Consistency Models”. In: *International Conference on Architecture of Computing Systems (ARCS)*. Ed. by M. Berekovic, R. Buchty, H. Hamann, D. Koch, and T. Pionteck. Vol. 10793. LNCS. Braunschweig, Germany: Springer, 2018, pp. 195–208.

[SeSc18a] M. Senftleben and K. Schneider. “Using Temporal Logics for Specifying Weak Memory Consistency Models”. In: *International Journal of Critical Computer-Based Systems (IJCCBS)* 8.2 (2018), pp. 214–229.

[WeGe94] D.L. Weaver and T. Germond, eds. *The SPARC Architecture Manual-Version 9*. Prentice-Hall, Inc., 1994.

Appendix A

Test Encoding in SMT2

Contents

A.1. Encoding of $\mathcal{T}_\alpha$ for Sequential consistency

```
(declare-fun ex (Int Int) Bool) ; Varaible ex_i,j
(declare-fun sv (Int Int) Bool) ; Variable sv_i,j
; ### Execution ##########################################
; Exists write for read
(assert (ex 0 3)) ; wx1 -> rx1
(assert (ex 2 1)) ; wx2 -> rx2
; Only one write for read (same variable & data)
; > none, as only one write matches read
; ### SerialViews ########################################
; ## SV: forall (*,*,*,*,*), respecting <_PO
; Total Order & AntiSymmetry
(assert (xor (not (sv 0 1) ) (not (sv 1 0))))
(assert (xor (not (sv 0 2) ) (not (sv 2 0))))
(assert (xor (not (sv 0 3) ) (not (sv 3 0))))
(assert (xor (not (sv 1 2) ) (not (sv 2 1))))
(assert (xor (not (sv 1 3) ) (not (sv 3 1))))
(assert (xor (not (sv 2 3) ) (not (sv 3 2))))
; Transitivity
(assert (=> (and (sv 0 1) (sv 1 2)) (sv 0 2) ))
(assert (=> (and (sv 0 1) (sv 1 3)) (sv 0 3) ))
(assert (=> (and (sv 0 2) (sv 2 1)) (sv 0 1) ))
(assert (=> (and (sv 0 2) (sv 2 3)) (sv 0 3) ))
(assert (=> (and (sv 0 3) (sv 3 1)) (sv 0 1) ))
(assert (=> (and (sv 0 3) (sv 3 2)) (sv 0 2) ))
(assert (=> (and (sv 1 0) (sv 0 2)) (sv 1 2) ))
(assert (=> (and (sv 1 0) (sv 0 3)) (sv 1 3) ))
(assert (=> (and (sv 1 2) (sv 2 0)) (sv 1 0) ))
(assert (=> (and (sv 1 2) (sv 2 3)) (sv 1 3) ))
(assert (=> (and (sv 1 3) (sv 3 0)) (sv 1 0) ))
(assert (=> (and (sv 1 3) (sv 3 2)) (sv 1 2) ))
(assert (=> (and (sv 2 0) (sv 0 1)) (sv 2 1) ))
(assert (=> (and (sv 2 0) (sv 0 3)) (sv 2 3) ))
(assert (=> (and (sv 2 1) (sv 1 0)) (sv 2 0) ))
```

```
(assert (=> (and (sv 2 1) (sv 1 3)) (sv 2 3) ))
(assert (=> (and (sv 2 3) (sv 3 0)) (sv 2 0) ))
(assert (=> (and (sv 2 3) (sv 3 1)) (sv 2 1) ))
(assert (=> (and (sv 3 0) (sv 0 1)) (sv 3 1) ))
(assert (=> (and (sv 3 0) (sv 0 2)) (sv 3 2) ))
(assert (=> (and (sv 3 1) (sv 1 0)) (sv 3 0) ))
(assert (=> (and (sv 3 1) (sv 1 2)) (sv 3 2) ))
(assert (=> (and (sv 3 2) (sv 2 0)) (sv 3 0) ))
(assert (=> (and (sv 3 2) (sv 2 1)) (sv 3 1) ))
; SV refines <_PO
(assert (sv 0 1)) ; wx1 <_PO rx2
(assert (sv 2 3)) ; wx2 <_PO rx1
; Writes-To implies SV & no intermediate write
; wx1 -> rx1
(assert (or (not (ex 0 3)) (sv 0 3)))
(assert (or (not (ex 0 3)) (not (sv 0 0)) (not (sv 0 3))))
(assert (or (not (ex 0 3)) (not (sv 0 2)) (not (sv 2 3))))
; wx2 -> rx2
(assert (or (not (ex 2 1)) (sv 2 1)))
(assert (or (not (ex 2 1)) (not (sv 2 0)) (not (sv 0 1))))
(assert (or (not (ex 2 1)) (not (sv 2 2)) (not (sv 2 1))))
(check-sat)
```

A.2. Encoding of $\mathcal{T}_\alpha$ for Pipelined RAM consistency

```
(declare-fun ex (Int Int) Bool)  ; Variable ex_i,j
(declare-fun svP (Int Int) Bool) ; Variable sv_i,j for process p
(declare-fun svQ (Int Int) Bool) ; Variable sv_i,j for process q
; ### Execution ########################################
; Existence of Write for read
(assert (ex 0 3)) ; wx1 -> rx1
(assert (ex 2 1)) ;: wx2 -> rx2
; Only one write for read (same variable & data)
; > none, as only one write matches read
; ### SerialViews ######################################
; ## SV_P: forall (*,*,*,p,*)u(w,*,*,*,*), respecting <_PO
; Total Order & AntiSymmetry
(assert (xor (not (svP 0 1) ) (not (svP 1 0))))
(assert (xor (not (svP 0 2) ) (not (svP 2 0))))
(assert (xor (not (svP 1 2) ) (not (svP 2 1))))
; Transitivity
(assert (=> (and (svP 0 1) (svP 1 2)) (svP 0 2) ))
(assert (=> (and (svP 0 2) (svP 2 1)) (svP 0 1) ))
(assert (=> (and (svP 1 0) (svP 0 2)) (svP 1 2) ))
(assert (=> (and (svP 1 2) (svP 2 0)) (svP 1 0) ))
(assert (=> (and (svP 2 0) (svP 0 1)) (svP 2 1) ))
(assert (=> (and (svP 2 1) (svP 1 0)) (svP 2 0) ))
; SV_p refines <_PO
(assert (svP 0 1)) ; wx1 <_PO rx2
; Writes-To implies SV_p & no intermediate write
; wx2 -> rx2
(assert (or (not (ex 2 1)) (svP 2 1)))
(assert (or (not (ex 2 1)) (not (svP 2 0)) (not (svP 0 1))))
(assert (or (not (ex 2 1)) (not (svP 2 2)) (not (svP 2 1))))
; ## SV_Q: forall (*,*,*,q,*)u(w,*,*,*,*), respecting <_PO
; Total Order & AntiSymmetry
(assert (xor (not (svQ 0 2) ) (not (svQ 2 0))))
(assert (xor (not (svQ 0 3) ) (not (svQ 3 0))))
(assert (xor (not (svQ 2 3) ) (not (svQ 3 2))))
; Transitivity
(assert (=> (and (svQ 0 2) (svQ 2 3)) (svQ 0 3) ))
(assert (=> (and (svQ 0 3) (svQ 3 2)) (svQ 0 2) ))
(assert (=> (and (svQ 2 0) (svQ 0 3)) (svQ 2 3) ))
(assert (=> (and (svQ 2 3) (svQ 3 0)) (svQ 2 0) ))
(assert (=> (and (svQ 3 0) (svQ 0 2)) (svQ 3 2) ))
(assert (=> (and (svQ 3 2) (svQ 2 0)) (svQ 3 0) ))
; SV_q refines <_PO
(assert (svQ 2 3)) ; wx2 <_PO rx1
; Writes-To implies SV_q & no intermediate write
; wx1 -> rx1
(assert (or (not (ex 0 3)) (svQ 0 3)))
(assert (or (not (ex 0 3)) (not (svQ 0 0)) (not (svQ 0 3))))
(assert (or (not (ex 0 3)) (not (svQ 0 2)) (not (svQ 2 3))))
(check-sat)
```

Appendix B

LTL encoding in SMV

Contents

The encoding uses a SMV preprocessor that allows to use define constants (#define), to include other files (#include), and to repeat expressions based on constants to allow for quantification ([[repeatline]], [[forall]]).

First, the encoded invariants and LTL specification are given as seperate files, then the encoding of some examples are given in the SMV preprocessor format.

B.1. Invariants and Specifications

B.1.1. Invariants

```
-----------------------------------------------------------------------
-- INVARIANTS --

-- read and write only occur if processor is active
[[repeatline %p=MIN_P..MAX_P]]
    INVAR (!step[%p]) -> (!read[%p] & !write[%p]);
[[/repeatline]]

-- read and write never occur at the same time
[[repeatline %p=MIN_P..MAX_P]]
    INVAR !(write[%p] & read[%p]);
```

```
[[/repeatline]]

-- readValue is undefined while no read occurs
[[repeatline %i=MIN_P..MAX_P]]
    INVAR (!read[%i]) -> (readValue[%i]=UNDEFINED);
[[/repeatline]]

-----------------------------------------------------------------------------
-- Fairness: step[p] has to hold infinitely often (for all processors p)
[[repeatline %p=MIN_P..MAX_P]]
    JUSTICE step[%p];
[[/repeatline]]
-----------------------------------------------------------------------------
```

B.1.2. LTL Specification: Base

```
-----------------------------------------------------------------------------
-- Processing Causality
(
[[forall %p=MIN_P..MAX_P]][[forall %q=MIN_P..MAX_P]]
[[forall %i=MIN_ID..MAX_ID]][[forall %l=MIN_LOC..MAX_LOC]]
        ( (!(proc[%p] & (procProcess[%p] = %q)
        & (procId[%p] = %i) & (procLocation[%p] = %l)))
        U (write[%q] & (writeId[%q] = %i) & (writeLocation[%q] = %l)) )
        | G(!(proc[%p] & (procProcess[%p] = %q)
        & (procId[%p] = %i) & (procLocation[%p] = %l)))
[[/forall]][[/forall]]
[[/forall]][[/forall]]
)
-----------------------------------------------------------------------------
-- Processing Uniqueness
& (G
    [[forall %p=MIN_P..MAX_P]][[forall %q=MIN_P..MAX_P]]
    [[forall %i=MIN_ID..MAX_ID]]
        (proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %i))
        -> X G !(proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %i))
    [[/forall]]
    [[/forall]][[/forall]])

-- Read Initial
& ([[forall %p=MIN_P..MAX_P]][[forall %l=MIN_LOC..MAX_LOC]]
        ( (read[%p] & (readLocation[%p]=%l) -> (readValue[%p]=UNDEFINED))
        U (proc[%p] & procLocation[%p]=%l) )
    | G(read[%p] & (readLocation[%p]=%l) -> (readValue[%p]=UNDEFINED))
[[/forall]][[/forall]])
-----------------------------------------------------------------------------
-- Read Causality
& (G
    [[forall %q=MIN_P..MAX_P]][[forall %i=MIN_ID..MAX_ID]]
    [[forall %l=MIN_LOC..MAX_LOC]][[forall %v=MIN_VAL..MAX_VAL]]
        (write[%q] & (writeId[%q]=%i)
    & (writeLocation[%q]=%l) & (writeValue[%q]=%v))
        -> G [[forall %p=MIN_P..MAX_P]]
                        (proc[%p] & (procProcess[%p] = %q) & (procId[%p]=%i))
                        -> ( ( ( ( read[%p] & (readLocation[%p]=%l))
                    -> readValue[%p]=%v )
                U (proc[%p] & (procLocation[%p]=%l)
                    & ( (procProcess[%p] != %q) | (procId[%p] != %i) )) )
                | ( G ( ( read[%p] &
                        (readLocation[%p]=%l)) -> readValue[%p]=%v ) ) )
                [[/forall]]
    [[/forall]][[/forall]]
    [[/forall]][[/forall]])
-----------------------------------------------------------------------------
```

B.1.3. LTL Specification: Local

```
-------------------------------------------------------------------------
-- Include Base Spec
(
#include(spec-base.ltl)
)
-- Local Causality
& ([[forall %p=MIN_P..MAX_P]]
        G ((write[%p]) ->
        (proc[%p] & (procProcess[%p]=%p) & (procId[%p]=writeId[%p])))
[[/forall]])
-------------------------------------------------------------------------
```

B.1.4. LTL Specification: Slow

```
-------------------------------------------------------------------------
-- Include Local Spec
(
#include(spec-local.ltl)
)
-- Slow
& (G [[forall %p=MIN_P..MAX_P]][[forall %q=MIN_P..MAX_P]]
    [[forall %i=MIN_ID..MAX_ID]][[forall %l=MIN_LOC..MAX_LOC]]
        (proc[%p] & (procProcess[%p] = %q)
        & (procId[%p] = %i) & (procLocation[%p] = %l))
    -> (X G
        [[forall %j=MIN_ID..MAX_ID]]
            !((%j <= %i) & proc[%p] & (procProcess[%p] = %q)
                & (procId[%p] = %j) & (procLocation[%p] = %l))
        [[/forall]])
[[/forall]][[/forall]]
[[/forall]][[/forall]])
-------------------------------------------------------------------------
```

B.1.5. LTL Specification: CC

```
-------------------------------------------------------------------------
-- Include Slow Spec
(
#include(spec-slow.ltl)
)
-- CC --
& [[forall %p=MIN_P..MAX_P]][[forall %q=MIN_P..MAX_P]]
[[forall %r=MIN_P..MAX_P]][[forall %i=MIN_ID..MAX_ID]]
[[forall %j=MIN_ID..MAX_ID]][[forall %l=MIN_LOC..MAX_LOC]]
        (F ( (proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %i)
        & (procLocation[%p] = %l))
    & (X F (proc[%p] & (procProcess[%p] = %r) & (procId[%p] = %j)
        & (procLocation[%p] = %l))) ) )
        -> G [[forall %pp=MIN_P..MAX_P]]
                    (proc[%pp] & (procProcess[%pp] = %r) & (procId[%pp] = %j))
                    -> G !(proc[%pp] & (procProcess[%pp] = %q)
                & (procId[%pp] = %i))
                [[/forall]]
[[/forall]][[/forall]]
[[/forall]][[/forall]]
[[/forall]][[/forall]]
-------------------------------------------------------------------------
```

B.1.6. LTL Specification: PRAM

```
-------------------------------------------------------------------------
-- Include Slow Spec
```

```
(
#include(spec-slow.ltl)
)
-- PRAM --
& [[forall %p=MIN_P..MAX_P]][[forall %q=MIN_P..MAX_P]]
[[forall %i=MIN_ID..MAX_ID]]
        (F (proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %i)))
        -> [[forall %j=MIN_ID..MAX_ID]]
        (%j < %i) ->
        ( (F (proc[%p]& (procProcess[%p] = %q) & (procId[%p] = %j)))
        & G ( (proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %i))
        -> G !(proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %j)) ) )
    [[/forall]]
[[/forall]]
[[/forall]][[/forall]]
```

B.1.7. LTL Specification: SC

```
-- Include Local Spec
(
#include(spec-local.ltl)
)
--SC: Global Total Order
-- TOTAL
& G [[forall %q=MIN_P..MAX_P]][[forall %i=MIN_ID..MAX_ID]]
(write[%q] & (writeId[%q]=%i)) ->
    [[forall %p=MIN_P..MAX_P]]
        F (proc[%p] & (procProcess[%p]=%q) & (procId[%p]=%i))
    [[/forall]]
[[/forall]][[/forall]]
-- GLOBAL ORDER
& G
[[forall %p=MIN_P..MAX_P]][[forall %q=MIN_P..MAX_P]]
[[forall %r=MIN_P..MAX_P]][[forall %i=MIN_ID..MAX_ID]]
[[forall %j=MIN_ID..MAX_ID]]
        (F ((proc[%p] & (procProcess[%p] = %q) & (procId[%p] = %i))
    & (F (proc[%p] & (procProcess[%p] = %r) & (procId[%p] = %j)))))
        ->
        [[forall %pp=MIN_P..MAX_P]]
                (F ((proc[%pp] & (procProcess[%pp] = %q) & (procId[%pp] = %i))
        & (F (proc[%pp] & (procProcess[%pp] = %r) & (procId[%pp] = %j)))))
        [[/forall]]
[[/forall]]
[[/forall]][[/forall]]
[[/forall]][[/forall]]
```

B.2. Petersons Mutual Exclusion Algorithm

```
-------------------------------------------------------------------------
-- Petersons Mutual Exclusion Algorithm ---------------------------------
-------------------------------------------------------------------------

-- LOCATIONS: flag0, flag1, turn, data

-- 00:  write (0,1)                              flag[self] <- T
-- 01:  write (2,1)                              turn <- other
-- 02:  reg = read (1)                    if (!flag[other]) goto 6
-- 03:  if(reg=0) goto 6                  "
-- 04:  reg = read (2)                    if (turn=other) goto 2
-- 05:  if(reg=1) goto 2                  "
-- 06:  reg = read (3)                    data++;
-- 07:  write (3,reg+1)                "
-- 08:  write (0,0)                              flag[self] <- F
-- 09:  goto 9

-------------------------------------------------------------------------

#define MIN_P                   0
#define MAX_P                   1
#define MIN_LOC         0
#define MAX_LOC         3
#define MIN_VAL         0
#define MAX_VAL         2
#define UNDEFINED       0
#define MIN_ID                  0
#define MAX_ID                  10

-------------------------------------------------------------------------

-- Processor --
MODULE Processor(id, step, write, writeLocation, writeValue, read,
                 readLocation, readValue)
VAR
        reg : MIN_VAL..MAX_VAL;
        pc : 0..9;
        other: MIN_P..MAX_P;
ASSIGN
        other := ((id=0)?1:0);
        init(pc) := 0;
        next(pc) :=
                case
                        !step : pc;
                        (pc = 3) & (reg = 0) : 6;
                        (pc = 5) & (reg != id) : 2;
                        pc >= 9 : 9;
                        TRUE : (pc+1);
                esac;
        init(reg) := UNDEFINED;
        next(reg) :=
                case
                        !step : reg;
                        pc = 2 : readValue;
                        pc = 4 : readValue;
                        pc = 6 : readValue;
                        TRUE : reg;
                esac;
        write :=
                case
                        !step: FALSE;
                        TRUE: (pc = 0 | pc = 1 | pc = 7 | pc = 8);
                esac;
```

```
        writeLocation :=
                case
                        !step: 0;
                        pc = 0 : id;
                        pc = 1 : 2;
                        pc = 7 : 3;
                        pc = 8 : id;
                        TRUE : 0;
                esac;
        writeValue :=
                case
                        !step: 0;
                        pc = 0 : 1;
                        pc = 1 : other;
                        pc = 7 : (reg>=MAX_VAL)?reg:(reg+1);
                        pc = 8 : 0;
                        TRUE : 0;
                esac;
        read :=
                case
                        !step: FALSE;
                        TRUE: (pc = 2 | pc = 4 | pc = 6);
                esac;
        readLocation :=
                case
                        !step: 0;
                        pc = 2 : other;
                        pc = 4 : 2;
                        pc = 6 : 3;
                        TRUE : 0;
                esac;
-----------------------------------------------------------------------
MODULE main
VAR
        step: array MIN_P..MAX_P of boolean;
        write: array MIN_P..MAX_P of boolean;
        writeId: array MIN_P..MAX_P of MIN_ID..MAX_ID;
        writeLocation: array MIN_P..MAX_P of MIN_LOC..MAX_LOC;
        writeValue: array MIN_P..MAX_P of MIN_VAL..MAX_VAL;
        read: array MIN_P..MAX_P of boolean;
        readValue: array MIN_P..MAX_P of MIN_VAL..MAX_VAL;
        readLocation: array MIN_P..MAX_P of MIN_LOC..MAX_LOC;
        proc: array MIN_P..MAX_P of boolean;
        procProcess: array MIN_P..MAX_P of MIN_P..MAX_P;
        procId: array MIN_P..MAX_P of MIN_ID..MAX_ID;
        procLocation: array MIN_P..MAX_P of MIN_LOC..MAX_LOC;
VAR
[[repeatline %i=MIN_P..MAX_P]]
    p%i : Processor(%i, step[%i], write[%i], writeLocation[%i],
            writeValue[%i], read[%i], readLocation[%i], readValue[%i]);
[[/repeatline]]
ASSIGN
[[repeatline %i=MIN_P..MAX_P]]  init(writeId[%i]) := 0;[[/repeatline]]
[[repeatline %i=MIN_P..MAX_P]]
    next(writeId[%i]) :=
      (write[%i] & (writeId[%i]<MAX_ID)) ? (writeId[%i]+1) : writeId[%i];
[[/repeatline]]

-----------------------------------------------------------------------

#include(invariants)

-----------------------------------------------------------------------
-- First Test: CC
LTLSPEC
(
```

```
#include(spec-cc.ltl)
)
-- Property to check
->
( G  !((p0.pc=6) & (p1.pc=6)) )
;
--EXPECTED: FALSE (counterexample <=> possible behavior)
--PROVEN (BMC CounterExample)
------------------------------------------------------------------------
-- Second Test: PRAM
LTLSPEC
(
#include(spec-pram.ltl)
)
-- Property to check
->
( G  !((p0.pc=6) & (p1.pc=6)) )
;
--EXPECTED: FALSE (counterexample <=> possible behavior)
--PROVEN (BMC CounterExample)
------------------------------------------------------------------------
-- Third Test: SC
LTLSPEC
(
#include(spec-sc.ltl)
)
-- Property to check
->
( G  !((p0.pc=6) & (p1.pc=6)) )
;
--EXPECTED: TRUE (no counterexample)
--PROVEN (BMC Depth 27)
------------------------------------------------------------------------
```

Appendix C

Reference Machine Quartz Implementations

Contents

The mechanism for explicit non-determinism in Quartz ('choose') was replaced by oracle variables which are added to the modules interface. This has been done for verification purposes and to enable the reproducibility of simulation results.

C.1. Shared Modules

C.1.1. FIFO

This is a FIFO buffer with elements in the following form:

$$(writeFlag, originProcess, memoryTarget, value)$$

The tuple field *writeFlag* determines if the entry is a write or read operation, *originProcess* contains the ID of the process which issued the operation, *memoryTarget* is the memory address the operation operates on and *value* contains the value to be written to memory in case of a write operation.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth = 8;
macro MemSize = 8;

macro BufferSize = 6;

module FIFO(
    event ?pop,
    event ?push,
    event !isempty,
    event isfull,
    // input : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) ?inp,
    // output : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) !outp
    ) {

    [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) fifo;
    nat{BufferSize} head;
    nat{BufferSize} nxxt;
    bool empty;

    empty = true;

    always {
        if(empty) {
            emit(isempty);
        }
        if((head==nxxt) & !empty) {
            emit(isfull);
        }

        if(!empty) {
            outp = fifo[head];
        }

        if(pop & !empty) {
            if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

            if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
                if(!(push & !isfull)) {
                    next(empty) = true;
                }
            }
        }

        if(push & !isfull) {
            next(empty) = false;
```

```
            next(fifo[nxxt]) = inp;

            if(nxxt!=BufferSize-1) {
                next(nxxt) = nxxt+1;
            } else {
                next(nxxt) = 0;
            }
        }
    }
}
```

C.1.2. FIFOwClock

This is a FIFO buffer with elements in the following form:

$$(writeFlag, originProcess, memoryTarget, value, clock)$$

The interface of FIFO was extended by adding field *clock* which holds a clock value (natural number). The other fields behave like the counterpart in module FIFO.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth = 8;
macro MemSize = 8;

macro MaxClock = 127;

macro BufferSize = 6;

module FIFOwClock(
    event ?pop,
    event ?push,
    event !isempty,
    event isfull,
    // input : writeCommand & target & value & clock
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * nat{MaxClock}) →
        ?inp,
    // output : writeCommand & target & value & clock
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * nat{MaxClock}) →
        !outp
    ) {

    [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
        nat{MaxClock}) fifo;
    nat{BufferSize} head;
    nat{BufferSize} nxxt;
    bool empty;

    empty = true;

    always {
        if(empty) {
            emit(isempty);
        }
        if((head==nxxt) & !empty) {
            emit(isfull);
        }

        if(!empty) {
            outp = fifo[head];
        }
```

```
        if(pop & !empty) {
            if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

            if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
                if(!(push & !isfull)) {
                    next(empty) = true;
                }
            }
        }

        if(push & !isfull) {
            next(empty) = false;
            next(fifo[nxxt]) = inp;

            if(nxxt!=BufferSize-1) {
                next(nxxt) = nxxt+1;
            } else {
                next(nxxt) = 0;
            }
        }
    }
}
```

C.1.3. FIFOwClocks

This is a FIFO buffer with elements in the following form:

$$(writeFlag, originProcess, memoryTarget, value, clocks)$$

The interface of FIFO was extended by adding field *clocks* which holds a tuple of clock values (natural numbers), one for each process. The other fields behave like the counterpart in module FIFO.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth = 8;
macro MemSize = 8;

macro MaxClock = 127;

macro BufferSize = 6;

module FIFOwClocks(
    event ?pop,
    event ?push,
    event !isempty,
    event isfull,
    // input : writeCommand & origin & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
        [ProcessCount]nat{MaxClock}) ?inp,
    // output : writeCommand & origin & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
        [ProcessCount]nat{MaxClock}) !outp
    ) {

    [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
        [ProcessCount]nat{MaxClock}) fifo;
    nat{BufferSize} head;
    nat{BufferSize} nxxt;
    bool empty;
```

```
    empty = true;

    always {
        if(empty) {
            emit(isempty);
        }
        if((head==nxxt) & !empty) {
            emit(isfull);
        }

        if(!empty) {
            outp = fifo[head];
        }

        if(pop & !empty) {
            if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

            if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
                if(!(push & !isfull)) {
                    next(empty) = true;
                }
            }
        }

        if(push & !isfull) {
            next(empty) = false;
            next(fifo[nxxt]) = inp;

            if(nxxt!=BufferSize-1) {
                next(nxxt) = nxxt+1;
            } else {
                next(nxxt) = 0;
            }
        }
    }
}
```

C.1.4. FIFOwReadForwarding

This is a FIFO buffer with elements in the following form:

$$(writeFlag, originProcess, memoryTarget, value)$$

The fields behave like the counterpart in module FIFO.

This buffer additionally offers an interface and mechanisms to retrieve the most recent write's value if available for a given memory address.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth = 8;
macro MemSize = 8;

macro BufferSize = 6;

module FIFOwReadForwarding(
    event ?pop,
    event ?push,
    event !isempty,
    event isfull,
    // input : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) ?inp,
    // output : writeCommand & target & value
```

```
event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) !outp,

// Read Forward Mechanisms
// readIn : valid & address
event (bool * nat{MemSize}) ?readIn,
// readOut : success & value
event (bool * bv{DataWidth}) !readOut
) {

// FIFO variables
    [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) fifo;
    nat{BufferSize} head;
    nat{BufferSize} nxxt;
    bool empty;

// Read Forward Mechanism variables
    nat{BufferSize} tail;
    event [BufferSize] nat{BufferSize} readpos;
    event nat{BufferSize} headreadpos;
    event [BufferSize] bool readdone;

// Initialze FIFO empty
empty = true;

always {
    if(empty) {
        emit(isempty);
    }
    if((head==nxxt) & !empty) {
        emit(isfull);
    }

    if(!empty) {
        outp = fifo[head];
    }

    if(pop & !empty) {
        if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

        if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
            if(!(push & !isfull)) {
                next(empty) = true;
            }
        }
    }

    if(push & !isfull) {
        next(empty) = false;
        next(fifo[nxxt]) = inp;

        if(nxxt!=BufferSize-1) {
            next(nxxt) = nxxt+1;
        } else {
            next(nxxt) = 0;
        }
    }


    // Read Forward Mechanism //

    //tail = (nxxt==0?BufferSize-1:nxxt-1);
    if(nxxt<=0) {
        tail = BufferSize-1;
    } else {
        tail = nxxt-1;
    }
```

```
        for(i = 0 .. BufferSize-1) do || {
            //readpos[i] = (tail>=i?tail-i:tail+BufferSize-i);
            if(tail>=i) {
                readpos[i] = tail-i;
            } else {
                readpos[i] = tail+BufferSize-i;
            }
        }
        if(readIn.0 & !empty) { // If read request and FIFO not empty
            for(i = 0 .. BufferSize-1) do || {
                let(pos = readpos[i])
                {
                    if(i == head) { headreadpos = pos; }

                    if(pos <= headreadpos) { // exclude invalid entries
                        if(pos == 0) { // start from nxt-1 % size
                            if(readIn.1 == fifo[i].2) {
                                emit(readdone[0]);
                                readOut = (true, fifo[i].3) ;
                            }
                        } else {
                            if((!readdone[pos-1]) & readIn.1 == fifo[i].2) {
                                emit(readdone[pos]);
                                readOut = (true, fifo[i].3) ;
                            } else {
                                readdone[pos] = readdone[pos-1];
                            }
                        }
                    }
                }
            }
        }
    }
}
```

C.1.5. MemUnit

This is a memory unit which processes memory operations and returns read results. The values are stored in a bit-vector array.

```
package Architecture.ConsistencyModels.Structure;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module MemUnit(
        // input : issue & (writeCommand & origin & target & value)
        event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) →
              ?arbiterOut,

        // output (doneRead & origin & value)
        event (bool * nat{ProcessCount} * bv{DataWidth}) !readResult
    ) {

    [MemSize]bv{DataWidth} Mem;

    always {
        immediate await(arbiterOut.0);

        if((arbiterOut.1).0) { // write
            Mem[(arbiterOut.1).2] = (arbiterOut.1).3;
```

```
        } else {
            readResult = (true, (arbiterOut.1).1, Mem[(arbiterOut.1).2]);
        }
    }
}
```

C.1.6. MemUnitSingleCell

A single cell memory unit which processes memory operations and returns read results of a single memory location. The value is stored in a bit-vector.

```
package Architecture.ConsistencyModels.Structure;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module MemUnitSingleCell(
        // input : issue & (writeCommand & origin & target & value)
        event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) →
            ?arbiterOut,

        // output (doneRead & origin & value)
        event (bool * nat{ProcessCount} * bv{DataWidth}) !readResult
    ) {

    bv{DataWidth} Mem;

    always {
        immediate await(arbiterOut.0);

        if((arbiterOut.1).0) { // write
            Mem = (arbiterOut.1).3;
        } else {
            readResult = (true, (arbiterOut.1).1, Mem);
        }
    }
}
```

C.2. Reference Machines

C.2.1. Sequential consistency Reference Machine

```
package Architecture.ConsistencyModels.RefSequential;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefSequential(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
```

```
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event nat{ProcessCount} ?oracle
) {

    // FIFO
        event [ProcessCount] bool FIFOpop;
        event [ProcessCount] bool FIFOpush;
        event [ProcessCount] bool FIFOisempty;
        event [ProcessCount] bool FIFOisfull;
        // input : writeCommand & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
             bv{DataWidth}) FIFOinp;
        // output : writeCommand & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
             bv{DataWidth}) FIFOoutp;

    // Mem
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount-1) do || {
         fifo : FIFO(FIFOpop[i], FIFOpush[i], FIFOisempty[i],
                     FIFOisfull[i], FIFOinp[i], FIFOoutp[i]);
        ||
        always {
            if(reqMem[i] & !FIFOisfull[i]) {
                emit(ackMem[i]);

                FIFOinp[i] = (writeMem[i], i, adrBus[i], dataBus[i]);
                emit(FIFOpush[i]);

                if(writeMem[i]) {
                    emit(doneMem[i]);
                }
            }
        }
    }
    ||
    always {
        let(o1 = oracle) { // Arbiter (Simply choose from N components)
        //choose(o1 = 0 .. ProcessCount-1) {
            if(!FIFOisempty[o1]) {
                memIn = (true, FIFOoutp[o1]);
                emit(FIFOpop[o1]);
            }
        //}
        }
    }
    ||
    memunit: MemUnit(memIn, readResult);
    ||
    always { // Distributor (distributes read results to corresponding process)
        if(readResult.0) {
            emit(doneMem[readResult.1]);
            dataBus[readResult.1] = readResult.2;
```

```
        }
    }
}
```

C.2.2. Slow consistency Reference Machine

```
package Architecture.ConsistencyModels.RefSlow;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefSlow(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event [ProcessCount] nat{ProcessCount} ?oracle,
    event [ProcessCount] nat{MemSize} ?oracle2
) {
    // FIFO
        event [ProcessCount][ProcessCount][MemSize] bool FIFOpop;
        event [ProcessCount][ProcessCount][MemSize] bool FIFOpush;
        event [ProcessCount][ProcessCount][MemSize] bool FIFOisempty;
        event [ProcessCount][ProcessCount][MemSize] bool FIFOisfull;
        // input : writeCommand & origin target & value
        event [ProcessCount][ProcessCount][MemSize] (bool * nat{ProcessCount} * →
            nat{MemSize} * bv{DataWidth}) FIFOinp;
        // output : writeCommand & origin target & value
        event [ProcessCount][ProcessCount][MemSize] (bool * nat{ProcessCount} * →
            nat{MemSize} * bv{DataWidth}) FIFOoutp;

        event [ProcessCount] bool someFIFOfull;

    // Memory Units
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount−1) do || {
        always {
            // Distributor: broadcasts writes to all connected FIFOs
                // and sends reads to own FIFO
            // SubDistributor (splits memory cells)
            if(reqMem[i] & !someFIFOfull[i]) {
                emit(ackMem[i]);
                if(writeMem[i]) {
                    emit(doneMem[i]);
```

```
                }
                for(j = 0 .. ProcessCount-1) {
                    if((j==i) | writeMem[i]) {
                        // REMARK: notify mixed indices.
                        FIFOpush[j][i][ adrBus[i] ] = true;
                        FIFOinp[j][i][ adrBus[i] ] =
                            (writeMem[i], i, adrBus[i], dataBus[i]);
                    }
                }
            }

        }
        ||
        for(j = 0 .. ProcessCount-1) do || { // FIFO
            for(k = 0 .. MemSize-1) do || {
                always {
                    if(FIFOisfull[i][j][k]) {
                        emit(someFIFOfull[j]);
                    }
                }
                ||
                fifo : FIFO(FIFOpop[i][j][k], FIFOpush[i][j][k], FIFOisempty[i][j][k],
                            FIFOisfull[i][j][k], FIFOinp[i][j][k], FIFOoutp[i][j][k]);
            }
        }
        ||
        always { // Arbiter (Simply choose from NxM components)
            let(o1 = oracle[i])
            let(o2 = oracle2[i])
            {
            //choose(o1 = 0 .. ProcessCount-1) choose(o2 = 0 .. MemSize-1) {
                if(!FIFOisempty[i][o1][o2]) {
                    memIn[i] = (true, FIFOoutp[i][o1][o2]);
                    emit(FIFOpop[i][o1][o2]);
                }
            //}
            }
        }
        ||
        memunit: MemUnit(memIn[i], readResult[i]);
        ||
        always {
            if(readResult[i].0) {
                emit(doneMem[i]);
                dataBus[i] = readResult[i].2;
            }
        }
    }
}
```

C.2.3. Local consistency Reference Machine

```
package Architecture.ConsistencyModels.RefLocal;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefLocal(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
```

```
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event [ProcessCount] nat{ProcessCount+1} ?oracle,
    event [ProcessCount] nat{2} ?oracle2
) {
    // FIFO
        event [ProcessCount][ProcessCount] bool FIFOpop;
        event [ProcessCount][ProcessCount] bool FIFOpush;
        event [ProcessCount][ProcessCount] bool FIFOisempty;
        event [ProcessCount][ProcessCount] bool FIFOisfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOinp;
        // output : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOoutp;

        event [ProcessCount] bool someFIFOfull;

    // FIFOloop
        event [ProcessCount] bool FIFOloopPop;
        event [ProcessCount] bool FIFOloopPush;
        event [ProcessCount] bool FIFOloopIsempty;
        event [ProcessCount] bool FIFOloopIsfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOloopInp;
        // output : writeCommand & origin & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOloopOutp;

    // Arbiter
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) arbiterTemp;

    // Mem
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount-1) do || {
        always { // Distributor (broadcasts write to all connected FIFOs and sends reads to →
            own FIFO)
            if(reqMem[i] & !someFIFOfull[i]) {
                emit(ackMem[i]);
                if(writeMem[i]) {
                    emit(doneMem[i]);
                }

                for(j = 0 .. ProcessCount-1) {
                    if((j==i) | writeMem[i]) {
                        // REMARK: notify mixed indices.
                        FIFOpush[j][i] = true;
```

```
                    FIFOinp[j][i] = (writeMem[i], i, adrBus[i], dataBus[i]);
                }
            }
        }

    }
    ||
    for(j = 0 .. ProcessCount-1) do || {
        always {
            if(FIFOisfull[j][i]) {
                emit(someFIFOfull[i]);
            }
        }
        ||
        fifo : FIFO(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j], FIFOisfull[i][j], →
            FIFOinp[i][j], FIFOoutp[i][j]);
    }
    ||
    fifoloop : FIFO(FIFOloopPop[i], FIFOloopPush[i], FIFOloopIsempty[i], →
        FIFOloopIsfull[i], FIFOloopInp[i], FIFOloopOutp[i]);
    ||
    always { // Arbiter ("Shuffle" others, pass own)
        let(o1 = oracle[i])
        let(o2 = oracle2[i])
        {
            //choose(o1 = 0 .. ProcessCount) {
                if(o1 < ProcessCount) { // read from FIFOs
                    if(!FIFOisempty[i][o1]) {
                        arbiterTemp[i] = (true, FIFOoutp[i][o1]);
                        emit(FIFOpop[i][o1]);
                    }
                } else { // read from feedback
                    if(!FIFOloopIsempty[i]) {
                        arbiterTemp[i] = (true, FIFOloopOutp[i]);
                        emit(FIFOloopPop[i]);
                    }
                }
            //}
            if(arbiterTemp[i].0) {
                //choose(o2 = 0 .. 1) {
                    if(o1 == i | o2 == 0) { // pass to Mem
                        memIn[i] = arbiterTemp[i];
                    } else { // feedback
                        if(!FIFOloopIsfull[i]) {
                            FIFOloopInp[i] = arbiterTemp[i].1;
                            emit(FIFOloopPush[i]);
                        }
                    }
                //}
            }
        }
    }
    ||
    memunit: MemUnit(memIn[i], readResult[i]);
    ||
    always {
        if(readResult[i].0) {
            emit(doneMem[i]);
            dataBus[i] = readResult[i].2;
        }
    }
  }
}
```

C.2.4. Pipelined RAM consistency Reference Machine

```
package Architecture.ConsistencyModels.RefPRAM;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefPRAM(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event [ProcessCount] nat{ProcessCount} ?oracle
) {
    // FIFO
        event [ProcessCount][ProcessCount] bool FIFOpop;
        event [ProcessCount][ProcessCount] bool FIFOpush;
        event [ProcessCount][ProcessCount] bool FIFOisempty;
        event [ProcessCount][ProcessCount] bool FIFOisfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOinp;
        // output : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOoutp;

        event [ProcessCount] bool someFIFOfull;

    // Arbiter
        // output : issue & (writeCommand & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) arbiterOut;

    // Memory Units
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount−1) do || {
        always {
            // Distributor: broadcasts writes to all connected FIFOs
                // and sends reads to own FIFO
            if(reqMem[i] & !someFIFOfull[i]) {
                emit(ackMem[i]);
                if(writeMem[i]) {
                    emit(doneMem[i]);
                }
```

```
                for(j = 0 .. ProcessCount-1) {
                    if((j==i) | writeMem[i]) {
                        // REMARK: notify mixed indices.
                        FIFOpush[j][i] = true;
                        FIFOinp[j][i] = (writeMem[i], i, adrBus[i], dataBus[i]);
                    }
                }
            }

        }
        ||
        for(j = 0 .. ProcessCount-1) do || { // FIFO
            always {
                if(FIFOisfull[j][i]) {
                    emit(someFIFOfull[i]);
                }
            }
            ||
            fifo : FIFO(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                        FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);
        }
        ||
        always { // Arbiter (Simply choose from N components)
            let(o1 = oracle[i]) {
            //choose(o1 = 0 .. ProcessCount-1) {
                if(!FIFOisempty[i][o1]) {
                    memIn[i] = (true, FIFOoutp[i][o1]);
                    emit(FIFOpop[i][o1]);
                }
            //}
            }
        }
        ||
        memunit: MemUnit(memIn[i], readResult[i]);
        ||
        always { // returns memory unit read result to connected process
            if(readResult[i].0) {
                emit(doneMem[i]);
                dataBus[i] = readResult[i].2;
            }
        }
    }
}
```

C.2.5. Cache consistency Reference Machine

```
package Architecture.ConsistencyModels.RefCache;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnitSingleCell;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefCache(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
```

```
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated flags
    [ProcessCount] bool ?terminated,

    // oracle (replacement for choose)
    event [MemSize] nat{ProcessCount} ?oracle
) {
    // FIFO
        event [MemSize][ProcessCount] bool FIFOpop;
        event [ProcessCount][MemSize] bool FIFOpush;
        event [MemSize][ProcessCount] bool FIFOisempty;
        event [ProcessCount][MemSize] bool FIFOisfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOinp;
        // output : writeCommand & origin & target & value
        event [MemSize][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOoutp;

        event [ProcessCount] bool someFIFOfull;

    // Memory Units
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event [MemSize] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event [MemSize] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount-1) do || {
        always { // Distributor (Split memory cells)
            if(reqMem[i] & !someFIFOfull[i]) {
                emit(ackMem[i]);
                if(writeMem[i]) {
                    emit(doneMem[i]);
                }

                FIFOpush[i][adrBus[i]] = true;
                FIFOinp[i][adrBus[i]] = (writeMem[i], i, adrBus[i], dataBus[i]);
            }
        }
        ||
        for(j = 0 .. MemSize-1) do || {
            always {
                if(FIFOisfull[i][j]) {
                    emit(someFIFOfull[i]);
                }
            }
            ||
            // REMARK: notify mixed indices.
            fifo : FIFO(FIFOpop[j][i], FIFOpush[i][j], FIFOisempty[j][i],
                        FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[j][i]);
        }
    }
    ||
    for(i = 0 .. MemSize-1) do || {
        always { // Arbiter (Simply choose from M components)
            let(o1 = oracle[i]) {
            //choose(o1 = 0 .. ProcessCount-1) {
                if(!FIFOisempty[i][o1]) {
                    memIn[i] = (true, FIFOoutp[i][o1]);
                    emit(FIFOpop[i][o1]);
                }
            //}
```

```
            }
        }
        ||
        memunit: MemUnitSingleCell(memIn[i], readResult[i]);
        ||
        always {
            if(readResult[i].0)  {
                emit(doneMem[readResult[i].1]);
                dataBus[readResult[i].1]  = readResult[i].2;
            }
        }
    }
}
```

C.2.6. Causal consistency Reference Machine

```
package Architecture.ConsistencyModels.RefCausal;

import Architecture.ConsistencyModels.Structure.FIFOwClocks;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

macro MaxClock = 127;

module RefCausal(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    event [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event [ProcessCount] nat{ProcessCount} ?oracle
) {
    // Clocks
        [ProcessCount][ProcessCount] nat{MaxClock} clocks;
        event [ProcessCount][ProcessCount] nat{MaxClock} tempclocks;
        event [ProcessCount] bool clockgreater;

    // Dist
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth} * [ProcessCount]nat{MaxClock})) distIn;
        event [ProcessCount][ProcessCount−1] (bool * (bool * nat{ProcessCount} * →
            nat{MemSize} * bv{DataWidth} * [ProcessCount]nat{MaxClock})) distOut;

    // FIFO
        event [ProcessCount][ProcessCount−1] bool FIFOpop;
        event [ProcessCount][ProcessCount−1] bool FIFOpush;
        event [ProcessCount][ProcessCount−1] bool FIFOisempty;
        event [ProcessCount][ProcessCount−1] bool FIFOisfull;
        // input : writeCommand & target & value
        event [ProcessCount][ProcessCount−1] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth} * [ProcessCount]nat{MaxClock}) FIFOinp;
```

```
    // output : writeCommand & target & value
    event [ProcessCount][ProcessCount−1] (bool ∗ nat{ProcessCount} ∗ nat{MemSize} ∗ →
        bv{DataWidth} ∗ [ProcessCount]nat{MaxClock}) FIFOoutp;

    event [ProcessCount] bool someFIFOfull;

// Mem
    // input: issue & (writeCommand & target & value)
    event [ProcessCount] (bool ∗ (bool ∗ nat{ProcessCount} ∗ nat{MemSize} ∗ →
        bv{DataWidth})) memIn;
    event [ProcessCount] (bool ∗ nat{ProcessCount} ∗ bv{DataWidth}) readResult;

for(i = 0 .. ProcessCount−1) do || {
    always { // Arbiter
        //choose(o1 = 0 .. ProcessCount−1) {
        let(o1 = oracle[i]) {
            if(o1 == i) { // check for r/w
                if(reqMem[i] & !someFIFOfull[i]) {
                    emit(ackMem[i]);

                    // pass memory command to memory
                    memIn[i] = (true, (writeMem[i], i, adrBus[i], dataBus[i]));

                    if(writeMem[i]) {
                        emit(doneMem[i]);

                        for(k = 0 .. ProcessCount−1) {
                            if(k==i) {
                                tempclocks[i][k] = clocks[i][k]+1;
                            } else {
                                tempclocks[i][k] = clocks[i][k];
                            }
                        }

                        // Send write to FIFOs
                        distIn[i] = (true,
                                     (writeMem[i], i, adrBus[i], dataBus[i],
                                      tempclocks[i]));
                        next(clocks[i]) = tempclocks[i];
                    } else {
                        // read : await memory response
                        immediate await(readResult[i].0);
                    }
                }
            } else if(o1 != i) { // check FIFO
                let(o1m = (o1>=i & o1!=0?o1−1:o1)) {
                    if(!FIFOisempty[i][o1m]) {
                        let(clock = FIFOoutp[i][o1m].4) {
                            // check if operations clock is leq than own clock
                            for(k = 0 .. ProcessCount−1) {
                                if(k!=o1) {
                                    if(clock[k] > clocks[i][k]) {
                                        emit(clockgreater[i]);
                                    }
                                }
                            }
                            if(!clockgreater[i]) {
                                emit(FIFOpop[i][o1m]);

                                // pass memory command to memory
                                memIn[i] = (true,
                                            (FIFOoutp[i][o1m].0, FIFOoutp[i][o1m].1,
                                             FIFOoutp[i][o1m].2, FIFOoutp[i][o1m].3));

                                // copy clock origin component to own clock
                                next(clocks[i][o1]) = clock[o1];
```

```
                                }
                            }
                        }
                    }
                }
            }
        }
        ||
        always { // DistBroadcastClockedW
            if(distIn[i].0) {
                for(j = 0 .. ProcessCount-1) {
                    if(i!=j) {
                        let(k = (j<=i & i!=0?i-1:i)) {
                            FIFOinp[j][k] = distIn[i].1;
                            emit(FIFOpush[j][k]);
                        }
                    }
                }
            }
        }
        ||
        for(j = 0 .. ProcessCount-2) do || {
            fifo : FIFOwClocks(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                               FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);

            if(FIFOisfull[i][j]) {
                emit(someFIFOfull[(i>j?j:j+1)]);
            }
        }
        ||
        memunit: MemUnit(memIn[i], readResult[i]);
        ||
        always { // return read results
            if(readResult[i].0) {
                emit(doneMem[i]);
                dataBus[i] = readResult[i].2;
            }
        }
    }
}
```

C.2.7. Processor consistency Reference Machine

```
package Architecture.ConsistencyModels.RefProcessor;

import Architecture.ConsistencyModels.Structure.FIFOwClock;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

macro MaxClock = 127;

module RefProcessor(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
```

```
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event nat{ProcessCount} ?oracle,
    event [ProcessCount] nat{MemSize} ?oracles2
) {

    // FIFO
        event [ProcessCount][MemSize] bool FIFOpop;
        event [ProcessCount][MemSize] bool FIFOpush;
        event [ProcessCount][MemSize] bool FIFOisempty;
        event [ProcessCount][MemSize] bool FIFOisfull;
        // input : writeCommand & origin & target & value & clock
        event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth} * nat{MaxClock}) FIFOinp;
        // output : writeCommand & origin & target & value & clock
        event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth} * nat{MaxClock}) FIFOoutp;

        event [MemSize] bool someFIFOfull;

    // Arbiter
        [ProcessCount] nat{MaxClock} mainArbiterClocks;
        [ProcessCount][ProcessCount] nat{MaxClock} subArbiterClocks;
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) subArbiterOut;

    // Mem
        event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    always { // MainArbiter
        //choose(oracle = 0 .. ProcessCount-1) {
            if(reqMem[oracle]) {
                emit(ackMem[oracle]);

                let(adr = adrBus[oracle])
                let(data = dataBus[oracle])
                let(write = writeMem[oracle])
                let(read = readMem[oracle])
                {
                    if(write) {
                        immediate await(!someFIFOfull[adr]);
                        for(i = 0 .. ProcessCount-1) do || {
                            FIFOinp[i][adr] =
                                (true, oracle, adr, data, mainArbiterClocks[oracle]);
                            emit(FIFOpush[i][adr]);
                        }
                        emit(doneMem[oracle]);
                        next(mainArbiterClocks[oracle]) = mainArbiterClocks[oracle]+1;
                    } else if(read) {
                        immediate await(!FIFOisfull[oracle][adr]);
                        FIFOinp[oracle][adr] =
                            (false, oracle, adr, data, mainArbiterClocks[oracle]);
                        emit(FIFOpush[oracle][adr]);
                    }
                }
            }
        //}
    }
    ||
    for(i = 0 .. ProcessCount-1) do || {
        for(j = 0 .. MemSize-1) do || {
            fifo : FIFOwClock(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
```

```
                    FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);
            ||
            always {
                if(FIFOisfull[i][j]) {
                    emit(someFIFOfull[j]);
                }
            }
        }
        ||
        always { // SubArbiter
            //choose(oracle2 = 0 .. ProcessCount-1) {
                let(oracle2 = oracles2[i])
                let(entry = FIFOoutp[i][oracle2])
                if(!FIFOisempty[i][oracle2]) {
                    if(entry.0) { // write
                        if(subArbiterClocks[i][entry.1] == entry.4) {
                            subArbiterOut[i] = (true, (entry.0,entry.1,entry.2,entry.3));
                            emit(FIFOpop[i][oracle2]);
                            next(subArbiterClocks[i][entry.1]) =
                                subArbiterClocks[i][entry.1]+1;
                        }
                    } else { // read
                        subArbiterOut[i] = (true, (entry.0,entry.1,entry.2,entry.3));
                        emit(FIFOpop[i][oracle2]);
                    }
                }
            //}
        }
        ||
        memunit: MemUnit(subArbiterOut[i], readResult[i]);
        ||
        always {
            if(readResult[i].0) {
                emit(doneMem[i]);
                dataBus[i] = readResult[i].2;
            }
        }
    }
}
```

C.2.8. Partial store ordering Reference Machine

```
package Architecture.ConsistencyModels.RefPSO;

import Architecture.ConsistencyModels.Structure.MemUnit;
import Architecture.ConsistencyModels.Structure.FIFOwReadForwarding;


macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefPSO(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,
```

```
    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event nat{ProcessCount+1} ?oracle,
    event nat{MemSize+1} ?oracle2
) {
    // FIFOwReadForwarding interface variables
        event [ProcessCount][MemSize] bool FIFOpop;
        event [ProcessCount][MemSize] bool FIFOpush;
        event [ProcessCount][MemSize] bool FIFOisempty;
        event [ProcessCount][MemSize] bool FIFOisfull;
        // input : writeCommand & target & value
        event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOinp;
        // output : writeCommand & target & value
        event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOoutp;

        // readIn : valid & address
        event [ProcessCount][MemSize] (bool * nat{MemSize}) FIFOreadIn;
        // readOut : success & value
        event [ProcessCount][MemSize] (bool * bv{DataWidth}) FIFOreadOut;

    // Arbiter variables
        // arbiterSelection : process which may proceed | arbiterSelection == ProcessCount →
            means idle
        nat{ProcessCount+1} arbiterSelection;
        // arbiterBufferSelection : process' buffer which may write back | →
            arbiterBufferSelection == MemSize means Read
        nat{MemSize+1} arbiterBufferSelection;

    // Mem interface variables
        // memIn : valid/issue & (writeCommand & target & value)
        event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) memIn;
        // readResult : valid & issuer & value)
        event (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount−1) do || {
        always { // StoreBuffer
            if(reqMem[i]) {
                emit(ackMem[i]);

                if(writeMem[i]) { // write operation
                    immediate await(!FIFOisfull[i][adrBus[i]]);
                    FIFOinp[i][adrBus[i]] = (true, i, adrBus[i], dataBus[i]);
                    emit(FIFOpush[i][adrBus[i]]);
                    emit(doneMem[i]);
                }

                if(readMem[i]) { // read operation
                    FIFOreadIn[i][adrBus[i]] = (true, adrBus[i]);
                    if(FIFOreadOut[i][adrBus[i]].0) {
                        dataBus[i] = FIFOreadOut[i][adrBus[i]].1;
                        emit(doneMem[i]);
                    } else {
                        immediate await(arbiterSelection == i
                                        & arbiterBufferSelection == MemSize);
                        memIn = (true, (false, i, adrBus[i], dataBus[i]));
                    }
                }
            }
        }
        ||
        always { // StoreBuffer write back / flush
            if(arbiterSelection == i & arbiterBufferSelection<MemSize) {
```

```
                if(!FIFOisempty[i][arbiterBufferSelection]) {
                    memIn = (true, (true, i, arbiterBufferSelection,
                                    FIFOoutp[i][arbiterBufferSelection].3));
                }
            }
        }
        ||
        for(j = 0 .. MemSize-1) do || {
            fifo : FIFOwReadForwarding(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                                       FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j],
                                       FIFOreadIn[i][j], FIFOreadOut[i][j]);
        }
    }
    ||
    always { // Arbiter
        //choose(oracle = 0 .. ProcessCount) {
            arbiterSelection = oracle;
        //}
        //choose(oracle = 0 .. ProcessCount) {
            arbiterBufferSelection = oracle2;
        //}
    }
    ||
    memunit: MemUnit(memIn, readResult);
    ||
    always { // Distribute Completed Read Operations to the corresponding process
        if(readResult.0) {
            dataBus[readResult.1] = readResult.2;
            emit(doneMem[readResult.1]);
        }
    }
}
```

C.2.9. Total store ordering Reference Machine

```
package Architecture.ConsistencyModels.RefTSO;

import Architecture.ConsistencyModels.Structure.MemUnit;
import Architecture.ConsistencyModels.Structure.FIFOwReadForwarding;


macro DataWidth = 8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefTSO(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event nat{ProcessCount+1} ?oracle,
    event bool ?oracle2
) {
```

```
// FIFOwReadForwarding interface variables
    event [ProcessCount] bool FIFOpop;
    event [ProcessCount] bool FIFOpush;
    event [ProcessCount] bool FIFOisempty;
    event [ProcessCount] bool FIFOisfull;
    // input : writeCommand & target & value
    event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
        bv{DataWidth}) FIFOinp;
    // output : writeCommand & target & value
    event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
        bv{DataWidth}) FIFOoutp;

    // readIn : valid & address
    event [ProcessCount] (bool * nat{MemSize}) FIFOreadIn;
    // readOut : success & value
    event [ProcessCount] (bool * bv{DataWidth}) FIFOreadOut;

// Arbiter variables
    // arbiterSelection : process which may proceed
        // | arbiterSelection == ProcessCount means idle
    nat{ProcessCount+1} arbiterSelection;
    // arbiterReadInsteadOfWb: should process read or do a WB
    bool arbiterReadInsteadOfWb;

// Mem interface variables
    // memIn : valid/issue & (writeCommand & target & value)
    event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) memIn;
    // readResult : valid & issuer & value)
    event (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

for(i = 0 .. ProcessCount-1) do || {
    always { // StoreBuffer insert or read
        if(reqMem[i]) {
            emit(ackMem[i]);

            if(writeMem[i]) { // write operation
                immediate await(!FIFOisfull[i]);
                FIFOinp[i] = (true, i, adrBus[i], dataBus[i]);
                emit(FIFOpush[i]);
                emit(doneMem[i]);
            }

            if(readMem[i]) { // read operation
                FIFOreadIn[i] = (true, adrBus[i]);
                if(FIFOreadOut[i].0) {
                    dataBus[i] = FIFOreadOut[i].1;
                    emit(doneMem[i]);
                } else {
                    immediate await((arbiterSelection == i) & arbiterReadInsteadOfWb);
                    memIn = (true, (false, i, adrBus[i], dataBus[i]));
                }
            }
        }
    }
    ||
    always { // StoreBuffer write back / flush
        immediate await(arbiterSelection == i & !arbiterReadInsteadOfWb
                        & !FIFOisempty[i]);
        memIn = (true, (true, i, FIFOoutp[i].2, FIFOoutp[i].3));
    }
    ||
    fifo : FIFOwReadForwarding(FIFOpop[i], FIFOpush[i], FIFOisempty[i],
                               FIFOisfull[i], FIFOinp[i], FIFOoutp[i], FIFOreadIn[i], →
                                   FIFOreadOut[i]);
}
||
```

```
    always { // Arbiter
        //choose(oracle = 0 .. ProcessCount) {
            arbiterSelection  = oracle;
        //}

        //choose {
        //  arbiterReadInsteadOfWb = true;
        //} else {
        //  arbiterReadInsteadOfWb = false;
        //}
        arbiterReadInsteadOfWb = oracle2;
    }
    ||
    memunit: MemUnit(memIn, readResult);
    ||
    always { // Distribute Completed Read Operations to the corresponding process
        if(readResult.0) {
            dataBus[readResult.1] = readResult.2;
            emit(doneMem[readResult.1]);
        }
    }
}
```

Appendix D

Curriculum Vitae

Berufserfahrung

2016–2018 **Wissenschaftlicher Mitarbeiter** TU Kaiserslautern
Fachbereich Informatik, Arbeitsgruppe Eingebettete Systeme

2011–2016 **Wissenschaftliche Hilfskraft** TU Kaiserslautern
Fachbereich Informatik, Arbeitsgruppe Eingebettete Systeme

2009–2014 **PHP Programmierer**
Euro Modul S.á r.l.

2008–2008 **Web-Programmierer**
KLUMAX Internet GmbH

Akademische Ausbildung

2011–2013 **Master of Science in Informatik** TU Kaiserslautern
Masterarbeit: *Operational Characterization of Weak Memory Consistency Models.*

2008–2011 **Bachelor of Science Informatik** TU Kaiserslautern
Bachelorarbeit: *Web-based Instruction-level Simulation of a Parameterized Dynamic Prozessor.*

Wehrdienst

2007.04–2007.12 **Grundwehrdienst**, Immendingen

Schulausbildung

1998-2007 **Abitur** Leibniz-Gymnasium, Neustadt a.d. Weinstraße
Leistungskurse: Mathematik, Physik, Englisch.
Admin-AG (Schulnetzwerkbetreuung), Robotik AG.