Chapter 5

Shared Memory Synchronization

Noemie Slaats, Bart Van Assche and Albert Hoogewijs

Summary

Explicitly parallel programs consist of several threads, where each thread is executed by a different processing unit. These threads all have access to a shared memory, and communicate by writing to or reading from the shared memory. Reads and writes of different threads execute uncoordinatedly. Threads can wait for other threads by using synchronization. Although reading and writing the shared memory is similar in all shared memory systems, most shared memory systems have their own set of synchronization instructions. The semantics of the memory access and synchronization instructions together is called a *memory model*, which is usually specified informally or using a formalism specific to the memory model. We present a unified formalization of shared memory models both in traditional and in VDM notation. We also show how the Mural tool helps in writing the VDM specifications and in generating the corresponding formal theory. A proof constructed with Mural shows that even basic properties of this formal theory can be nontrivial to prove.

5.1 Introduction

A shared memory system has an address space common to all processors using the shared memory. Such a shared memory system can have one of the following implementations: a shared memory multiprocessor, a hardware distributed shared memory multiprocessor, or a network of workstations with distributed shared memory software. A shared memory multiprocessor has a physically shared memory, while in the other two systems the memory is fully distributed over the processors. In all three systems every processor has a local memory with a copy of a part of the shared memory. This memory is called a cache memory of the shared memory. To preserve the semantics of a single shared memory, these cache memories have to be kept consistent. The generic structure of a shared memory is shown in figure 5.1.

Every processor runs one thread, or actually a uniprocessor program associated with the thread. The different uniprocessor programs together are the explicitly parallel program. While running a thread, a processor issues read and write operations to the shared memory. These read and write operations are the only interaction of the processor with the shared memory. We call the sequence of operations that result from running a thread the *execution* of that thread. Since the order of the operations in the execution is derived from the order of instructions in the uniprocessor program running, we call this order the *program order* of the operations issued by the corresponding processor.

In a shared memory system without local memories, every processor has the same view of the shared memory. When duplicating or caching the contents of the shared memory in local memories, however, every processor potentially can have a different view of the shared memory. We model a processor's view of the shared memory by specifying the history of the changes applied to that view. We represent this history by the order in which operations of all processors have been observed by the current processor. We call this order the *memory order* relation. It is a partial order relation, able to model concurrent operations.

Although the memory orders are relations over more operations than the program order, there is a strong relation between both. When the memory order as observed by a processor includes the program order of the same processor, that processor obeys the *uniprocessor correctness* property. This means that a one-processor program will execute correctly on that processor.

To cooperate in a deterministic way, threads must be able to wait for one another or to synchronize. Synchronization of two or more threads is a way to guarantee an ordering between memory operations of different threads. There are two kinds of synchronization operations: low-level and high-level. *Low-level synchronization operations* are read and write operations whose ordering is guaranteed to be the same in all memory order relations. *High-level synchronization operations* are the acquire-, release- and barrier operations. They are defined in section 5.2.4.

This chapter is organized as follows: in section 5.2 we give an informal description of a shared memory synchronization model. The VDM specification associated with this model is given in section 5.3. In section 5.4 the formal theory for the memory model is discussed. To generate this formal theory the Mural tool [4] is used. In a last section we review and discuss the example.

5.2 Formal Definitions

The following information will be used to represent any operation: operation type, sequence number, processor number and set of memory addresses where the operation takes effect on. Additionally, load operations have a loaded value associated

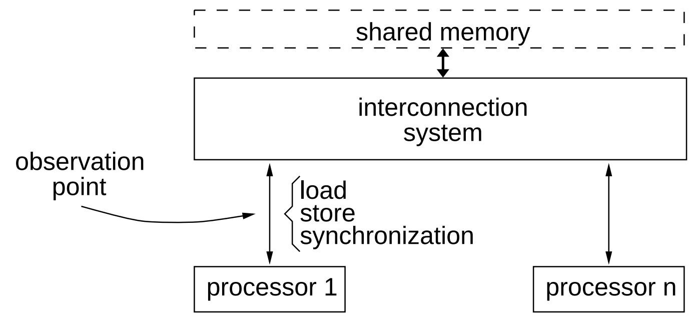

Figure 5.1: Generic structure of a shared memory system: n processors, a shared memory and inbetween an interconnection system. The interconnection system can reorder requests and can cause arbitrary delays when propagating requests from one processor to another. The shared memory itself is either physical or virtual. Program order and memory order relations over load, store and synchronization operations for a given processor are observed at the indicated observation point.
for a given processor are observed at the indicated observation point.

to them, store operations have a stored value, load-modify-store operations have a loaded and a stored value, and acquire, release and barrier-operations have an identification number associated to them.

A load operation reads from memory, a store operation writes to memory, and a load-modify-store operation modifies the value at a memory location without allowing intervening operations by other processors.

Definition 1 *operation type*

An operation has one of the types in $\mathit{Type} = \{\mathit{ns}, \mathit{nl}, \mathit{nf}, \mathit{ss}, \mathit{sl}, \mathit{sf}, \mathit{acq}, \mathit{rel}, \mathit{bar}\}$. These types are respectively normal store, load, load-modify-store, synchronizing store, load, load-modify-store, acquire, release and barrier operations.

We will use the symbol $n \in \mathbf{N}$ for the *number of processors* of the memory system, $P = \{1, \ldots, n\}$ for the *set of processor numbers*, and $p \in P$ for a *processor number*. $i \in \mathbf{N}$ is the per processor *operation number*, instructions executed later having a higher operation number. Equal operation numbers for different operations on the same processor are not allowed. $j \in \mathbf{N}$ is an *identification number* for barriers and critical sections, indicating which operations belong together. An entity in the memory that can be addressed is called a *location*, and Mem is the set of all addressable locations. $m \in \mathit{Mem}$ is a *single memory location*, and $M \subset \mathit{Mem}$ is a *set of memory locations*, indicating which set of locations is involved in an operation. For loads, stores and load-modify-stores this is typically a singleton, and for synchronization instructions this set M is a non-empty subset of Mem. What is stored into a location or read from a location is called a *value*, and the set of all allowed values is called Val. For an overview, see table 5.1.

operation name	general form	restricted form	shorthand
normal store	(ns, i, p, M, v_s)	$(ns, i, p, \{m\}, v\_s)$	l_$i(m,v_s)$
normal load	(nl, i, p, M, v_l)	$(nl, i, p, \{m\}, v\_l)$	s_$i(m,v_l)$
normal load-modify-store	(nf, i, p, M, v_l, v_s)	$(nf, i, p, \{m\}, v\_l, v\_s)$	f_$i(m,v_l,v_s)$
synchronizing store	(ss, i, p, M, v_s)	$(ss, i, p, \{m\}, v\_s)$	sl_$i(m,v_s)$
synchronizing load	(sl, i, p, M, v_l)	$(sl, i, p, \{m\}, v\_l)$	ss_$i(m,v_l)$
synchronizing load-modify-store	(sf, i, p, M, v_l, v_s)	$(sf, i, p, \{m\}, v\_l, v\_s)$	sf_$i(m,v_l,v_s)$
acquire	(acq, i, p, M, j)	(acq, i, p, M, j)	a_$i(M,j)$
release	(rel, i, p, M, j)	(rel, i, p, M, j)	r_$i(M,j)$
barrier	(bar, i, p, M, j)	(bar, i, p, M, j)	b_$i(M,j)$

Table 5.1: Names of the operation types, the general form of an operation tuple, the corresponding restricted form, and the abbreviation of the restricted form. The general form operates on a set of memory locations M, while the restricted form uses only one location m for load- and store-operations. Processor numbers are not specified in the restricted form: in graphs the processor number will be clear from the context.

The functions $num()$, $proc()$, $mem()$, $val\_l()$ and $val\_s()$ operate on operation tuples, and respectively return the values i, p, M, v_l and v_s.

The sets of operations of types ns, nl, nf, ss, sl, sf, acq, rel and bar are called respectively S_n, L_n, F_n, S_s, L_s, F_s, Acq, Rel, and Bar. Derived sets of operation types are in table 5.2.

5.2.1 Program Order and Executions

We will use the notation $\xrightarrow{\text{po } p}$ for the program order relation of processor p, and $\xrightarrow{\text{po}}$ for the union of these relations. Since the individual operations already include a per-processor sequence-number $proc()$, the program order relations can be easily defined using the sequence number.

Definition 2 *program order* $\xrightarrow{po}$ *and per processor program order* $\xrightarrow{po\ p}$

The program order relation $\xrightarrow{\text{po}}$ is the relation between operations $op\_1, op\_2 \in Op$ and is defined by

$$op\_1 \xrightarrow{\text{po}} op\_2 \iff (proc(op\_1) = proc(op\_2)) \wedge (num(op\_1) \leq num(op\_2)).$$

On processor p, operations are executed in the order $\xrightarrow{\text{po } p}$, defined by

$$op\_1 \xrightarrow{\text{po } p} op\_2 \iff (proc(op\_1) = proc(op\_2) = p) \wedge (num(op\_1) \leq num(op\_2)).$$

operation set contents	formal definition
store operations	$S = S\_n \cup F\_n \cup S\_s \cup F\_s$
load operations	$L = L\_n \cup F\_n \cup L\_s \cup F\_s$
low-level synchronization operations	$Sync\_L = S\_s \cup L\_s \cup F\_s$
high-level synchronization operations	$Sync\_H = Acq \cup Rel \cup Bar$
synchronization operations	$Sync = Sync\_L \cup Sync\_H$
operations executed on processor p	$Op\_p = \{op \in Op \mid proc(op) = p\}$
operations accessing at least location set M	$Op\_M = \{op \in Op \mid M \subset mem(op)\}$
operations accessing at least location set M on processor p	$Op\_M, p = Op\_M \cap Op\_p$
operations accessing at least location m	$Op\_m = Op\_\{m\}$
operations accessing at least location m on processor p	$Op\_m, p = Op\_m \cap Op\_p$

Table 5.2: Sets of operations for specific operation types, processor number, or accessed memory locations.

Parallel Program		Executed operations	
thread 1	*thread 2*	*processor 1*	*processor 2*
x := 1	y := 1	$s\_1(x, 1)$	$s\_3(y, 1)$
... := y	... := x	$l\_2(y, 0)$	$l\_4(x, 0)$

Figure 5.2: A short parallel program and the operations for one possible execution of that program. It is assumed that the memory has been initialized to zero. In this example the shared memory did not process the load and store requests in program order.

For an example of a parallel program, its program order and its memory order relations, see figures 5.2 and 5.3.

From the definition of the global and per processor program order relations, we can derive the following properties:

- the per-processor program order $\xrightarrow{\text{po}^{p}}$ is a total order for the operations of that processor, Op_p.
- the relation $\xrightarrow{\text{po}}$ is a partial order in the set Op.
- any order $\xrightarrow{\text{po}^{p}}$ is contained in the order relation $\xrightarrow{\text{po}}$, or $\forall p\colon P \cdot \xrightarrow{\text{po}^{p}} \subset \xrightarrow{\text{po}}$.

The set of all operations and the order in which they are executed contain all information, about how a program has been executed, observable by the threads. We call

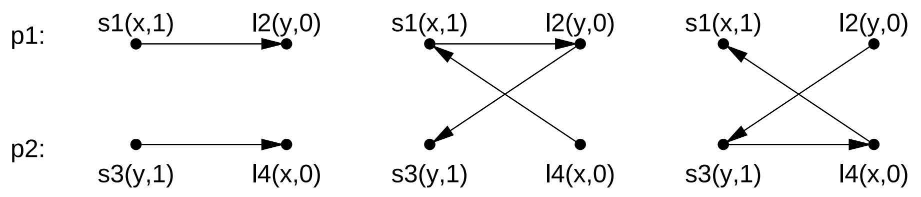

Figure 5.3: Sample global program order relation $\xrightarrow{\text{po}}$, and memory order relations $\xrightarrow{\text{mo}\,1}$ and $\xrightarrow{\text{mo}\,2}$ for processors p_1 and p_2 respectively. The relations in the graphs are the transitive reduction of the $\xrightarrow{\text{po}}$, $\xrightarrow{\text{mo}\,1}$ and $\xrightarrow{\text{mo}\,2}$ relations.

the pair of the set of operations and the program order relation of these operations an execution of the parallel program.

Definition 3 *execution E*

An execution $E = (Op, \xrightarrow{\text{po}})$ of a program consists of the set Op of operations and the order of execution $\xrightarrow{\text{po}}$ implied by the program.

5.2.2 Uniprocessor Correctness

Every processor of a multiprocessor system must obey uniprocessor correctness: when a sequential program is executed on a single processor of a multiprocessor system, the result must be the same as if the program was executed on a uniprocessor. This is achieved when all *data dependent* operations of a thread are present in the memory order relation in the same order as in program order. In its weakest form, two operations are data dependent if they access the same memory location, and either they are a store and a load operation, or they are two store operations writing different values. Executing data-dependent operations in a modified order changes either the result of one of the operations and/or the value written to memory. We will use a stronger condition than preserving data-dependences: we require that the order of any two operations referencing the same memory location is preserved.

Condition 1 *uniprocessor data dependences*

Any two operations on processor p to the same location that are ordered by program order, are ordered by the memory order relation $\xrightarrow{\text{mo}\,p}$ of that processor in the same way.

$$\forall m\colon Mem \cdot \forall op\_1, op\_2\colon Op\_m \cdot \forall p\colon P \cdot op\_1 \xrightarrow{\text{po}\,p} op\_2 \Longrightarrow op\_1 \xrightarrow{\text{mo}\,p} op\_2$$

5.2.3 Result of a Load

Since the relation $\xrightarrow{\text{mo}\,p}$ is the order in which a processor observes the memory operations of itself and other processors, the relation $\xrightarrow{\text{mo}\,p}$ determines the result of a load

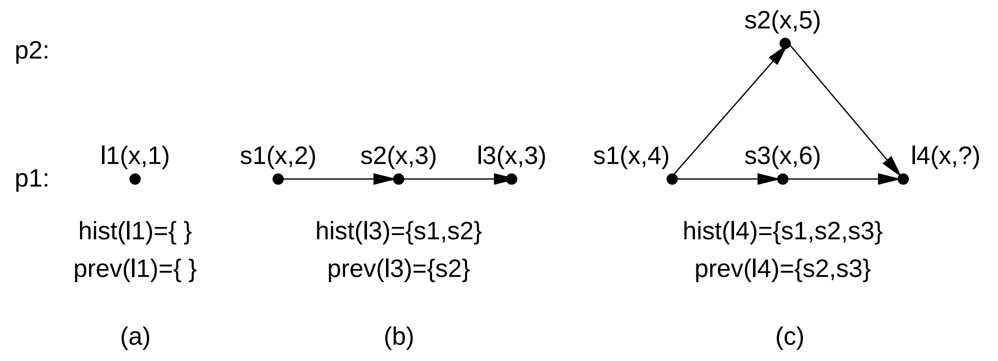

Figure 5.4: Three examples of execution sequences and their memory order relation $\xrightarrow{\text{mo } 1}$, illustrating the value condition. The first two sequences have been executed on one processor, the third on two processors. From the $\xrightarrow{\text{mo } 1}$ relation the sets *hist*(l) and *prev*(l) for the load l are derived, which in turn determine the result value of the load. The graphs represent the transitive reduction of the memory order relations $\xrightarrow{\text{mo } 1}$.

operation. For a load operation l, we define the set *hist*(l) as the set of all store operations preceding l, and the set *prev*(l) as the set of store operations immediately preceding l. When there is exactly one store operation s preceding l, the value stored by s is also the result returned by l. If there are zero, two, or more store operations preceding l, and they store zero, two, or more different values, then the result of l is undefined. For an example, see figure 5.4.

Condition 2 *value condition*

$$\begin{aligned}
p &\triangleq proc(l)\\
hist(l) &\triangleq \{s \in S\_mem(l) \setminus \{l\} \mid s \xrightarrow{\text{mo } p} l\}\\
prev(l) &\triangleq \{s \in hist(l) \mid \forall s'\colon hist(l) \setminus \{s\} \cdot \neg(s \xrightarrow{\text{mo } p} s')\}\\
&\forall s\colon prev(l) \cdot val\_l(l) = val\_s(s)
\end{aligned}$$

The relations $\xrightarrow{\text{mo } p}$ define the view of an individual processor of the order of memory operations. We also need the combined view of all processors of the order of the operations concerning a memory location m. We will call this relation the *access order for location m*, $\xrightarrow{\text{ao}}\_m$, and its union over all memory locations the *access order relation*, $\xrightarrow{\text{ao}}$.

Definition 4 *access order*
The access order $\xrightarrow{\text{ao}}\_m$ at the memory location m is the order in which operations are executed on the location m. The relation $\xrightarrow{\text{ao}}$, the access order relation, is the union of the $\xrightarrow{\text{ao}}\_m$ relations.

$$\begin{aligned}\xrightarrow{\text{ao}}\_m &\triangleq \bigcup p\colon P \cdot \xrightarrow{\text{mo } p} \cap\, Op\_m^2\\ \xrightarrow{\text{ao}} &\triangleq \left(\bigcup m\colon Mem \cdot \xrightarrow{\text{ao}}\_m\right)^{\star}\end{aligned}$$

Before defining the *shared memory condition*, we define *consistency of relations* and *sequential consistency of relations*.

Definition 5 *consistency of relations*
The relations $R\_1, \ldots, R\_n$ are consistent over the set A if and only if the relations $R\_1, \ldots, R\_n$ are identical when restricted to A:

$$R\_1 \cap A^2 = \ldots = R\_n \cap A^2.$$

Definition 6 *sequential consistency of relations*
The relations $R\_1, \ldots, R\_n$ are sequentially consistent over the set A if and only if they are consistent and every relation $R\_1, \ldots, R\_n$ is a total order over the set A.

While the view of operations of different processors on different memory locations can vary from processor to processor, all processors must have the same view of the order of operations on one single memory location. Additionally, combining these views must be meaningful. We call these two conditions together the *shared memory condition.*

Condition 3 *shared memory condition*

$$\begin{aligned}&\forall m\colon Mem \cdot \{\xrightarrow{\text{mo } 1}, \ldots, \xrightarrow{\text{mo } n}\} \text{ is consistent in } Op\_m\\ \wedge\ &\xrightarrow{\text{ao}} \text{ is a partial order.}\end{aligned}$$

From the shared memory condition, it follows that the individual $\xrightarrow{\text{ao}}\_m$ relations are partial order relations.

5.2.4 Synchronization

In this formalization of operations and memory models, there are three types of low-level synchronization operations (synchronizing load, synchronizing store and synchronizing load-modify-store) and three types of high-level synchronizing operations (barrier, acquire and release). These operations unify the synchronization primitives used in existing memory models.

Low-level synchronization operations behave the same way as their counterparts without synchronization, except that the order of low-level synchronization operations is identical in every memory order relation.

A *barrier* is a group of barrier operations b with the same identification number $id(b)$. These individual operations execute on a set of processors $P' = \{proc(b) \mid b \in B\}$. The barrier synchronizes for the processors in P' the access to the set of memory locations $M = mem(b)$: all memory references to any of the locations M that are ordered before the barrier as observed by any of the participating processors P', are before the barrier in the memory order relations of any processor. The same holds for memory accesses ordered after the barrier. We define the relation N as the equivalence relation that holds between barrier operations of the same barrier. Typically a barrier operation is implemented by waiting until all other processors participating in the barrier arrive at their corresponding barrier operation.

A critical section has an identification number j and protects the locations of the set M by sequentializing other critical sections to any of these locations M. A critical section on processor p starts with an *acquire* operation $(acq, ?, p, M, j)$ and ends with a *release* operation $(rel, ?, p, M, j)$. Although critical sections that have at least one location in common cannot overlap, access to any of the protected variables during the execution is still possible by another processor if it does not protect these accesses with a critical section. Operations executed inside the critical section are for every memory order relation ordered between the acquire and the release.

Before we start the formalization of barrier and critical section synchronization, we introduce the relations N and N'. These relations partition the barrier operations and the acquire/release pairs. The partitioning is based on the identification number $id()$. The relation N is extended to a reflexive relation over the set Op.

Definition 7 *equivalence relations* N, N'

$$\begin{aligned} N &\triangleq \{(op\_1, op\_2) \in Bar^2 \mid id(op\_1) = id(op\_2)\} \cup \{(op, op) \mid op \in Op\} \\ N' &\triangleq \{(a, r) \in (Acq \times Rel) \mid id(a) = id(r)\} \end{aligned}$$

Now we can express that barriers are totally ordered per memory location, and that all high- and low-level synchronization operations synchronizing a common memory location have the same ordering in all memory order relations. Also, we require that acquire-release pairs are totally ordered per memory location and neither overlap nor nest.

Condition 4 *order of synchronization operations*

$$\begin{aligned} & \forall m{:}\, Mem \cdot (\xrightarrow{\text{mo}\,1}/N, \xrightarrow{\text{mo}\,2}/N, \ldots, \xrightarrow{\text{mo}\,n}/N) \\ & \quad \text{are seq. consistent relations in } Sync\_m/N \\ \wedge\ & \forall m{:}\, Mem \cdot (\xrightarrow{\text{mo}\,1}/N', \xrightarrow{\text{mo}\,2}/N', \ldots, \xrightarrow{\text{mo}\,n}/N') \\ & \quad \text{are seq. consistent relations in } Sync\_m/N' \end{aligned}$$

Acquire and Release Every release requires exactly one matching acquire, but an acquire is allowed to have no matching release. In that case, the release is considered to be past the end of the execution of the thread on the processor of the acquire.

Condition 5 *correct use of acquire and release operations*

$$\begin{aligned} & \forall a\colon Acq \cdot \forall r\_1, r\_2\colon Rel \cdot aN'r\_1 \wedge aN'r\_2 \Longrightarrow r\_1 = r\_2 \\ \wedge \; & \forall r\colon Rel \cdot \exists!\, a\colon Acq \cdot aN'r \end{aligned}$$

Further, matching acquire and release operations have to reference the same set of memory locations and the acquire must be ordered before the release in the program order relation.

Condition 6 *same memory locations for matching acquire and release operations*

$$\forall a\colon Acq \cdot \forall r\colon Rel \cdot aN'r \Longrightarrow mem(a) = mem(r)$$

Condition 7 *execution order of matching acquire and release operations*

$$\forall a\colon Acq \cdot \forall r\colon Rel \cdot aN'r \Longrightarrow a \xrightarrow{\text{po}} r.$$

Also, any operation ordered by program order between the acquire and the release is for every processor also ordered by memory order between the acquire and the release.

Condition 8 *order of accesses to protected locations*

$$\begin{aligned} & \forall a\colon Acq \cdot \forall r\colon Rel \cdot aN'r \Longrightarrow \forall op\colon Op\_mem(a) \cdot \\ & (a \xrightarrow{\text{po}} op \xrightarrow{\text{po}} r \Longrightarrow \forall p\colon P \cdot a \xrightarrow{\text{mo } p} op \xrightarrow{\text{mo } p} r) \\ \wedge \; & \forall a\colon Acq \cdot (\neg \exists r\colon Rel \cdot aN'r) \Longrightarrow \forall op\colon Op\_mem(a) \cdot \\ & (a \xrightarrow{\text{po}} op \Longrightarrow \forall p\colon P \cdot a \xrightarrow{\text{mo } p} op) \end{aligned}$$

Barriers There are two conditions specific for barriers: any operation of a barrier has to reference the same set of memory locations, and all instructions executed before a barrier are observed before that barrier by any processor, while all instructions executed after a barrier are observed after that barrier.

Condition 9 *same set of locations per barrier*

$$\forall b\_1, b\_2\colon Bar \cdot b\_1 N b\_2 \Longrightarrow mem(b\_1) = mem(b\_2)$$

Condition 10 *correct barrier and operation ordering*

$$\begin{aligned} & \forall b\_1, b\_2 \cdot Bar. \forall p\colon P \cdot \forall op\colon Op\_mem(b\_1) \setminus \{b\_1\} \cdot \\ & \quad b\_1 N b\_2 \Longrightarrow (op \xrightarrow{\text{po}} b\_1 \Longrightarrow op \xrightarrow{\text{mo } p} b\_2) \\ & \qquad\qquad\qquad \wedge (b\_1 \xrightarrow{\text{po}} op \Longrightarrow b\_2 \xrightarrow{\text{mo } p} op) \end{aligned}$$

5.2.5 Memory Model

The following definition of a shared memory model contains the properties common to any model of a shared memory system.

Definition 8 *memory model*

A memory model *Mod* is a set of executions E, where every execution satisfies the general conditions 1, 2, 3, the synchronization conditions 4 to 10, and the model-specific conditions.

5.3 The VDM Specification of the Definitions

In the previous section a shared memory synchronization model is formalized in an informal way. In this section the VDM specification for all the definitions used for this formalization will be composed.
For that purpose the VDM specification support tool of Mural is used.

Mural is an interactive theorem prover and VDM specification support tool. The standard system is equipped with the theory of the VDM primitives. Because this theory is not expressive enough for our application, we added a theory for relations[1] to the parent theories of the specification.

5.3.1 Operations

The first object we have to specify is an operation. For that purpose some new types have to be declared.
A first such type, is the type to which all the terms expressing the kind of an operation can assigned to. We call this type *Type*:

Type = *not yet defined.*

We say that *Type* is not yet defined because there does not exist a type in the known theories, i.e. the theory of the VDM primitives and the relational theory, which we can use to specify the new type *Type*.
Because all terms that express some kind of an operation are assigned to the type *Type*, we know from section 5.2:

[1]This formal theory is described in appendix A

ns: *Type*

nl: *Type*

nf: *Type*

ss: *Type*

sl: *Type*

sf: *Type*

acq: *Type*

rel: *Type*

bar: *Type*

and these are the only terms of the type *Type*.
However, it is not necessary to include this information in the VDM specification. It is sufficient to wait until the VDM specification has been translated into its corresponding formal theory; and then a basic rule containing this information can be added to the set of basic rules of that theory.

In order to specify all possible operations of a particular execution of a parallel program two more types are needed. The first one is:

location = not yet defined.

An element of the type *location* is a memory location that can be addressed by an operation. All the allowed memory locations of a particular memory model are grouped in a set called *Mem*. This set is a constant for that memory model. Therefore its specification will be a constant of the form:

Mem: *location*-**set**.

What is stored into a memory location or read from a memory location is an element of the type *value*. This brings us to the second new type:

value = not yet defined.

The set of all the values that are allowed for a particular memory model is also a constant. This set is called *Val* and is specified by:

Val: *value*-**set**.

Making use of the previous defined types and constants all the possible operations can be specified.
Because the operations consist of different fields, as we can see in table 5.1, the best suited way to create these specifications is by using a composite type.

The normal store, load and load-modify-store operations

Translating the information of table 5.1, about the normal store operation to a VDM specification results in:

$$\begin{array}{rrcl} \textit{nstore} :: & \textit{type} & : & \textit{Type} \\ & i & : & \mathbf{N} \\ & p & : & \mathbf{N}_1 \\ & M & : & \textit{location}\text{-}\mathbf{set} \\ & v\_s & : & \textit{value} \end{array}$$

$$\begin{array}{l} \mathbf{inv}\ (\textit{type}\colon \textit{Type}, M\colon \textit{location}\text{-}\mathbf{set}, v\_s\colon \textit{value}) \triangleq \\ \quad \textit{type} = \textit{ns} \wedge ((M \subset \textit{Mem}) \wedge (v\_s \in \textit{Val})). \end{array}$$

The five fields of the record do not express all the information known about a normal store operation. Therefore an invariant containing the remaining information has been added. This invariant expresses that the type of a normal store operation always has to equal *ns*, and that the set of memory locations M has to be a subset of the allowed memory locations of the model and analogously the stored value v_s must be an element of the allowed values for this memory model.
The specification of the normal load operation is:

$$\begin{array}{rrcl} \textit{nload} :: & \textit{type} & : & \textit{Type} \\ & i & : & \mathbf{N} \\ & p & : & \mathbf{N}_1 \\ & M & : & \textit{location}\text{-}\mathbf{set} \\ & v\_l & : & \textit{value} \end{array}$$

$$\begin{array}{l} \mathbf{inv}\ (\textit{type}\colon \textit{Type}, M\colon \textit{location}\text{-}\mathbf{set}, v\_l\colon \textit{value}) \triangleq \\ \quad \textit{type} = \textit{nl} \wedge ((M \subset \textit{Mem}) \wedge (v\_l \in \textit{Val})). \end{array}$$

To construct the specification of a normal load-modify-store operation a record containing six fields instead of five fields is needed, because the normal load-modify-store operation first loads a value from a memory location and then stores another value to the same memory location. And of course both values have to be a member of the set of the allowed values of the memory model:

$$\begin{array}{rrcl} \textit{nmodify} :: & \textit{type} & : & \textit{Type} \\ & i & : & \mathbf{N} \\ & p & : & \mathbf{N}_1 \\ & M & : & \textit{location}\text{-}\mathbf{set} \\ & v\_l & : & \textit{value} \\ & v\_s & : & \textit{value} \end{array}$$

$$\begin{array}{l} \mathbf{inv}\ (\textit{type}\colon \textit{Type}, M\colon \textit{location}\text{-}\mathbf{set}, v\_l\colon \textit{value}, v\_s\colon \textit{value}) \triangleq \\ \quad \textit{type} = \textit{nf} \wedge ((M \subset \textit{Mem}) \wedge ((v\_l \in \textit{Val}) \wedge (v\_s \in \textit{Val}))). \end{array}$$

The synchronizing store, load and load-modify-store operations
From table 5.1, it follows that the only difference between the synchronizing store (resp. load, load-modify-store) operation and the normal store (resp. load and load-modify-store) operation is the type field:

$$\begin{aligned} \mathit{sstore} :: \mathit{type} &: \mathit{Type} \\ i &: \mathbf{N} \\ p &: \mathbf{N}_1 \\ M &: \mathit{location}\text{-}\mathsf{set} \\ v\_s &: \mathit{value} \end{aligned}$$

$$\begin{aligned} &\mathbf{inv}\ (\mathit{type}\colon \mathit{Type}, M\colon \mathit{location}\text{-}\mathsf{set}, v\_s\colon \mathit{value}) \underline{\triangle} \\ &\quad \mathit{type} = \mathit{ss} \wedge ((M \subset \mathit{Mem}) \wedge (v\_s \in \mathit{Val})) \end{aligned}$$

$$\begin{aligned} \mathit{sload} :: \mathit{type} &: \mathit{Type} \\ i &: \mathbf{N} \\ p &: \mathbf{N}_1 \\ M &: \mathit{location}\text{-}\mathsf{set} \\ v\_l &: \mathit{value} \end{aligned}$$

$$\begin{aligned} &\mathbf{inv}\ (\mathit{type}\colon \mathit{Type}, M\colon \mathit{location}\text{-}\mathsf{set}, v\_l\colon \mathit{value}) \underline{\triangle} \\ &\quad \mathit{type} = \mathit{sl} \wedge ((M \subset \mathit{Mem}) \wedge (v\_l \in \mathit{Val})) \end{aligned}$$

$$\begin{aligned} \mathit{smodify} :: \mathit{type} &: \mathit{Type} \\ i &: \mathbf{N} \\ p &: \mathbf{N}_1 \\ M &: \mathit{location}\text{-}\mathsf{set} \\ v\_l &: \mathit{value} \\ v\_s &: \mathit{value} \end{aligned}$$

$$\begin{aligned} &\mathbf{inv}\ (\mathit{type}\colon \mathit{Type}, M\colon \mathit{location}\text{-}\mathsf{set}, v\_l\colon \mathit{value}, v\_s\colon \mathit{value}) \underline{\triangle} \\ &\quad \mathit{type} = \mathit{sf} \wedge ((M \subset \mathit{Mem}) \wedge ((v\_l \in \mathit{Val}) \wedge (v\_s \in \mathit{Val}))). \end{aligned}$$

The acquire, release and barrier operations

There are three more operations left to specify. These are the acquire, release and barrier operations. They are called *high-level synchronizing* operations. Again, their specification will consist of a record, containing five fields, and an invariant explaining what properties the fields have to fulfill:

$$\begin{aligned} \mathit{acquire} :: \mathit{type} &: \mathit{Type} \\ i &: \mathbf{N} \\ p &: \mathbf{N}_1 \\ M &: \mathit{location}\text{-}\mathsf{set} \\ j &: \mathbf{N} \end{aligned}$$

$$\begin{aligned} &\mathbf{inv}\ (M\colon \mathit{location}\text{-}\mathsf{set}, \mathit{type}\colon \mathit{Type}) \underline{\triangle} \\ &\quad \mathit{type} = \mathit{acq} \wedge (M \subset \mathit{Mem}) \end{aligned}$$

$$\begin{aligned} \mathit{release} :: \mathit{type} &: \mathit{Type} \\ i &: \mathbf{N} \\ p &: \mathbf{N}_1 \\ M &: \mathit{location}\text{-}\mathsf{set} \\ j &: \mathbf{N} \end{aligned}$$

$$\begin{aligned} &\mathbf{inv}\ (\mathit{type}\colon \mathit{Type}, M\colon \mathit{location}\text{-}\mathsf{set}) \underline{\triangle} \\ &\quad \mathit{type} = \mathit{rel} \wedge (M \subset \mathit{Mem}) \end{aligned}$$

$$\begin{array}{rrcl} barrier :: & type & : & Type \\ & i & : & \mathbf{N} \\ & p & : & \mathbf{N}_1 \\ & M & : & location\text{-}\mathbf{set} \\ & j & : & \mathbf{N} \end{array}$$

$\mathbf{inv}\ (type\colon Type, M\colon location\text{-}\mathbf{set}) \triangleq$
$\quad type = bar \wedge (M \subset Mem).$

We still need to specify many functions before we can formalize our memory model. To make the specification of these functions a little easier some more types are defined. These new types are the union of previously declared types.
A first new type is the type *OPs* which collects all the store operations:

$OPs = nstore \mid sstore.$

Similarly we specify a type *OPl* for all load operations and a type *OPf* for all load-modify-store operations:

$OPl = nload \mid sload$

$OPf = nmodify \mid smodify.$

The type *OPsync* brings together all the elements of the type *acquire, release* and *barrier*:

$OPsync = acquire \mid release \mid barrier.$

Finally we introduce a union type that collects all kind of operations. This type is called *OP*. Making use of the previous definitions the specification of the type *OP* becomes:

$OP = OPs \mid OPl \mid OPf \mid OPsync.$

For the same reason some new types were defined, we also declare some useful functions.

5.3.2 Useful Functions

A first selected function is the function *type*. This function maps an element of the type *OP* to an element of the type *Type*; the image of a given operation under the function *type* is the type of that operation.
To which kind an operation belongs to, can be found in the first field of its corresponding record. The function associated with the first field of an operation record is the function *s-type*:

$type : OP \rightarrow Type$

$type(op) \quad \triangleq \quad s\text{-}type(op).$

The difference between the function *type* and the function *s-type* is: *type* is defined for all elements of the type *OP* and *s-type* is defined for all kinds of operations (i.e. normal load, normal store, ...) separately.

Some other useful functions are the functions $num()$, $proc()$, $mem()$, $val\_l()$ and $val\_s()$ mentioned in section 5.2. The VDM notation for these functions is similar to the previous specification:

$num : OP \rightarrow Type$

$num(op) \;\; \triangleq \;\; s\text{-}i(op).$

$proc : OP \rightarrow \mathbf{N}_1$

$proc(op) \;\; \triangleq \;\; s\text{-}p(op).$

$mem : OP \rightarrow location\text{-}\mathbf{set}$

$mem(op) \;\; \triangleq \;\; s\text{-}M(op).$

$val\text{-}l : OPl \mid OPf \rightarrow value$

$val\text{-}l(op) \;\; \triangleq \;\; s\text{-}v\_l(op).$

$val\text{-}s : OPs \mid OPf \rightarrow value$

$val\text{-}s(op) \;\; \triangleq \;\; s\text{-}v\_s(op).$

The last of these functions is *id* which maps a synchronizing operation to its identification number:

$id : OPsync \rightarrow \mathbf{N}$

$id(op) \;\; \triangleq \;\; s\text{-}j(op).$

By making use of the specifications of all these functions and types, we can specify one particular execution of a parallel program satisfying a given memory model.

5.3.3 The Program Order and Executions

Because the program order relation $\xrightarrow{pop}$ depends of an execution E, the notion execution has to be specified before the program order. Once the object execution is specified the program order can be derived from this specification.

A Program Execution

A program execution describes one particular execution of a parallel program for a given memory model. The definition of an execution E given in section 5.2.1 is a tuple satisfying some restrictions. Therefore a record and an invariant will be used for the VDM specification.
At first glance it seems that there are only two fields needed in the record: a first component being the set Op of all operations that have been executed during a particular execution E and a second component being a set po of pairs of operations. This set of pairs will define the program order of the execution when its elements satisfy the following three conditions:

- not only the first but also the last component of all the pairs must be an element of the previously defined set Op;
- both components of a pair must have been executed on the same processor;
- the operation number of the first component of a pair must be less than or equal to the operation number of the second component of that pair.

However, these two components together with their restrictions are not sufficient to specify an execution. We need a third component, namely a natural number n, different from zero, declaring the number of processors used for that particular execution of the parallel program. Knowing the number of processors used, we can formulate one more condition the execution has to satisfy: i.e. every operation of Op has to be executed on a processor used for the execution.
Bringing all this information together in a record with an invariant, results in:

$$\begin{array}{lrl} \mathit{Execution} :: & \mathit{Op} & : \mathit{OP}\text{-}\mathbf{set} \\ & \mathit{po} & : \mathit{OP} \times \mathit{OP}\text{-}\mathbf{set} \\ & n & : \mathbf{N}_1 \end{array}$$

$$\begin{array}{l} \mathbf{inv}\ (\mathit{Op}: \mathit{OP}\text{-}\mathbf{set}, \mathit{po}: \mathit{OP} \times \mathit{OP}\text{-}\mathbf{set}, n: \mathbf{N}_1) \triangleq \\ \quad \forall a: \mathit{OP} \cdot a \in \mathit{Op} \ \Rightarrow \ (\mathit{proc}(a) \leq n) \wedge \\ \quad \forall k: \mathit{OP} \times \mathit{OP} \cdot \\ \quad k \in \mathit{po} \ \Leftrightarrow \ (k \in \mathit{Prod}(\mathit{Op}, \mathit{Op})) \wedge \\ \quad (\mathit{proc}(\mathit{fst}(k)) = \mathit{proc}(\mathit{snd}(k)) \wedge (\mathit{num}(\mathit{fst}(k)) \leq \mathit{num}(\mathit{snd}(k)))). \end{array}$$

The Program Order and Per Processor Program Order

The program order po is one of the three defining components of an execution E. Therefore we do not need to specify the program order explicitly: for a given execution E the program order is specified by $s\text{-}po(E)$.

The program order is defined for all the operations of an execution. The per processor order only considers those operations that are executed on one particular processor of an execution.
Similarly to the program order, the per processor order is also specified as a set of pairs with each pair of the set satisfying four conditions:

- the first three conditions are the same as the conditions for the elements of the program order;
- the fourth condition expresses that all the operations of all the pairs must be executed on the same processor p.

To specify this set of pairs we use a function *Po-p*. This function depends on two variables: a particular execution E and a processor p used for the execution E:

$Po\text{-}p\ (p{:}\, \mathbf{N}_1, E{:}\, Execution)\ R{:}\, OP \times OP\text{-}\mathbf{set}$
$\mathbf{pre}\ p \leq s\text{-}n(E)$
$\mathbf{post}\ \forall k{:}\, OP \times OP \cdot k \in R \ \Leftrightarrow$
$\quad (k \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(fst(k)) = proc(snd(k))$
$\quad \wedge proc(snd(k)) = p) \wedge (num(fst(k)) \leq num(snd(k))))$.

Knowing the VDM specification of an execution, all the instructions S_n, L_n, F_n, S_s, L_s, F_s, *Acq*, *Rel*, and *Bar*, which were mentioned in section 5.2, and some more can also be specified.

Sets of Operations

A first set to be specified is the set $S\text{-}n(E)$ of all normal store operations executed during a particular execution E of a parallel program. Because all normal store operations of an execution are also operations of that execution, it is obvious that the set of all normal store operations executed in E is a subset of the set $s\text{-}Op(E)$. And vice versa when a normal store operation is an element of $s\text{-}Op(E)$ then it is a normal store operation executed by E. This means that:

$Sn\ (E{:}\, Execution)\ v{:}\, nstore\text{-}\mathbf{set}$
$\mathbf{post}\ \forall s{:}\, nstore \cdot s \in v \ \Leftrightarrow\ s \in s\text{-}Op(E)$.

The set $Ln(E)$ (resp. $Fn(E)$) of all normal load (resp. load-modify-store) operations of E, and the set $Ss(E)$ (resp. $Ls(E)$, $Fs(E)$) of all synchronizing store (resp. load, load-modify-store) operations are specified analogously:

$Ln\ (E{:}\, Execution)\ v{:}\, nload\text{-}\mathbf{set}$
$\mathbf{post}\ \forall l{:}\, nload \cdot l \in v \ \Leftrightarrow\ l \in s\text{-}Op(E)$

$Fn\ (E{:}\, Execution)\ v{:}\, nmodify\text{-}\mathbf{set}$
$\mathbf{post}\ \forall f{:}\, nmodify \cdot f \in v \ \Leftrightarrow\ f \in s\text{-}Op(E)$

$Ss\ (E{:}\, Execution)\ v{:}\, sstore\text{-}\mathbf{set}$
$\mathbf{post}\ \forall s{:}\, sstore \cdot s \in v \ \Leftrightarrow\ s \in s\text{-}Op(E)$

$Ls\ (E\colon Execution)\ v\colon sload\text{-}\mathbf{set}$
$\mathbf{post}\ \forall l\colon sload \cdot l \in v \ \Leftrightarrow\ l \in s\text{-}Op(E)$

$Fs\ (E\colon Execution)\ v\colon smodify\text{-}\mathbf{set}$
$\mathbf{post}\ \forall f\colon smodify \cdot f \in v \ \Leftrightarrow\ f \in s\text{-}Op(E).$

The same can be done for the set containing all the acquire *Acq*(E) (resp. release *Rel*(E), barrier *Bar*(E)) operations of an execution E:

$Acq\ (E\colon Execution)\ v\colon acquire\text{-}\mathbf{set}$
$\mathbf{post}\ \forall a\colon acquire \cdot a \in v \ \Leftrightarrow\ a \in s\text{-}Op(E)$

$Rel\ (E\colon Execution)\ v\colon release\text{-}\mathbf{set}$
$\mathbf{post}\ \forall r\colon release \cdot r \in v \ \Leftrightarrow\ r \in s\text{-}Op(E)$

$Bar\ (E\colon Execution)\ v\colon barrier\text{-}\mathbf{set}$
$\mathbf{post}\ \forall b\colon barrier \cdot b \in v \ \Leftrightarrow\ b \in s\text{-}Op(E).$

In furtherance of the next specifications, it is useful to specify the union of some previously specified sets: for example the set $S(E)$ of all store operations of an execution E and the set $L(E)$ of all load operations of E:

$S : Execution \to OP\text{-}\mathbf{set}$
$S(E) \ \triangleq\ (Sn(E) \cup Fn(E)) \cup (Ss(E) \cup Fs(E))$

$L : Execution \to OP\text{-}\mathbf{set}$
$L(E) \ \triangleq\ (Ln(E) \cup Fn(E)) \cup (Ls(E) \cup Fs(E)).$

Other sets that might be useful in the future are the set *SyncL*(E) of all low-level synchronizing operations and the set *SyncH*(E) of all high-level synchronizing operations:

$SyncL : Execution \to OP\text{-}\mathbf{set}$
$SyncL(E) \ \triangleq\ (Ss(E) \cup Ls(E)) \cup Fs(E)$

$SyncH : Execution \to OP\text{-}\mathbf{set}$
$SyncH(E) \ \triangleq\ (Acq(E) \cup Rel(E)) \cup Bar(E).$

The union of the two previous sets is $Sync(E)$, the set of all synchronizing operations of a particular execution E:

$Sync : Execution \to OP\text{-}\mathbf{set}$

$Sync(E) \quad \triangleq \quad SyncL(E) \cup SyncH(E).$

All the previously declared sets group operations of the same kind. Now, instead of grouping operations of a same '*Type*', some other properties are used to classify the operations of an execution E.

For instance, the set $Op\text{-}p(E)$ brings together all the operations of an execution E executed on a same processor p; where p is a processor used for the execution E:

$Op\text{-}p\ (p{:}\, \mathbf{N}_1, E{:}\, Execution)\ v{:}\, OP\text{-}\mathbf{set}$
pre $p \leq s\text{-}n(E)$
post $\forall op{:}\, OP \cdot op \in v \;\Leftrightarrow\; proc(op) = p \wedge (op \in s\text{-}Op(E)).$

The pre-condition of this function expresses that p needs to be a processor used for the given execution E.
$Op\text{-}M(E)$ is the set of all the operations of an execution E accessing at least a given set of memory locations M, with M a subset of all the allowed memory locations Mem:

$Op\text{-}M\ (M{:}\, location\text{-}\mathbf{set}, E{:}\, Execution)\ v{:}\, OP\text{-}\mathbf{set}$
pre $M \subseteq Mem$
post $\forall op{:}\, OP \cdot op \in v \;\Leftrightarrow\; (M \subseteq mem(op)) \wedge (op \in s\text{-}Op(E)).$

The intersection of the two previous sets is called $Op\text{-}Mp(M, p, E)$:

$Op\text{-}Mp : location \times \mathbf{N}_1 \times Execution \to OP\text{-}\mathbf{set}$

$Op\text{-}Mp(M, p, E) \quad \triangleq \quad Op\text{-}M(M, E) \cap Op\text{-}p(p, E)$

pre $(M \subseteq Mem) \wedge (p \leq s\text{-}n(E)).$

The function $Op\text{-}m$ returns the set of all operations of an execution E accessing at least a single given memory location m. Using the function $Op\text{-}M$ its specification is straightforward:

$Op\text{-}m : location \times Execution \to OP\text{-}\mathbf{set}$

$Op\text{-}m(m, E) \quad \triangleq \quad Op\text{-}M(\{m\}, E)$

pre $m \in Mem.$

The specification of the set $Op\text{-}mp(m, p, E)$ of all operations accessing at least a single memory location m on processor p is:

$Op\text{-}mp : location \times \mathbf{N}_1 \times Execution \rightarrow OP\text{-}\mathbf{set}$

$Op\text{-}mp(m, p, E) \quad \triangleq \quad Op\text{-}Mp(\{m\}, p, E)$

pre $(m \in Mem) \wedge (p \leq s\text{-}n(E))$.

For later purpose some of these sets are defined for load and synchronizing operations only:

$L\text{-}p\ (p\colon \mathbf{N}_1, E\colon Execution)\ v\colon OP\text{-}\mathbf{set}$
pre $p \leq s\text{-}n(E)$
post $\forall op\colon OP \cdot op \in v \Leftrightarrow proc(op) = p \wedge (op \in L(E))$

$L\text{-}m\ (m\colon location, E\colon Execution)\ v\colon OP\text{-}\mathbf{set}$
pre $m \in Mem$
post $\forall op\colon OP \cdot op \in v \Leftrightarrow (m \in mem(op)) \wedge (op \in L(E))$

$L\text{-}mp : location \times \mathbf{N}_1 \times Execution \rightarrow OP\text{-}\mathbf{set}$

$L\text{-}mp(m, p, E) \quad \triangleq \quad L\text{-}m(m, E) \cap L\text{-}p(p, E)$

pre $(m \in Mem) \wedge (p \leq s\text{-}n(E))$

$Sync\text{-}m\ (m\colon location, E\colon Execution)\ v\colon OP\text{-}\mathbf{set}$
pre $m \in Mem$
post $\forall op\colon OP \cdot op \in v \Leftrightarrow (m \in mem(op)) \wedge (op \in Sync(E))$.

Although there is not much mentioned about the memory order in the formalization, a specification of the memory order is indispensable to express the conditions a memory model has to fulfill.

5.3.4 Memory Order

Since the memory order relation is specified as a set of restrictions on a partial order relation, the only explicit information we have about the memory order $MO\text{-}p(p, E)$ is that it is a partial order:

$MO\text{-}p\ (p\colon \mathbf{N}_1, E\colon Execution)\ R\colon OP \times OP\text{-}\mathbf{set}$
pre $p \leq s\text{-}n(E)$
post $R \subseteq Prod(s\text{-}Op(E), s\text{-}Op(E)) \wedge PartOrder(R, s\text{-}Op(E))$.

More properties of this order are revealed in the following conditions.

5.3.5 Uniprocessor Correctness

The uniprocessor correctness is formalized through a property of the memory model. This property is not necessary to specify the memory model, it only expresses a property of the model. However to be sure the model satisfies this condition a basic rule, associated with the condition, will be added to the set of inference rules of the formal theory corresponding with the specification of the memory model.

The same remark can be made for all the conditions mentioned in the section 5.2.

5.3.6 The Result of a Load

Although there is no need for a specification of the value condition, some definitions appearing in this condition have to be declared:

$hist\ (l\colon OPl \mid OPf, E\colon Execution)\ v\colon OPs \mid OPf$-**set**
pre $l \in L(E)$
post $\forall s\colon OPs \mid OPf \cdot s \in v \;\Leftrightarrow$
$\quad(s \in S(E)) \wedge ((mem(l) \subseteq mem(s))\wedge$
$\quad(s \neq l \wedge (pair(s,l) \in MO\text{-}p(proc(l),E)))).$

$prev\ (l\colon OPl \mid OPf, E\colon Execution)\ v\colon OPs \mid OPf$-**set**
pre $l \in L(E)$
post $\forall s\colon OPs \mid OPf \cdot s \in v$
$\quad\Leftrightarrow\ (s \in hist(l,E)) \wedge \forall s'\colon OPs \mid OPf \cdot s' \in (hist(l,E) - \{s\})$
$\quad\Rightarrow\ pair(s,s') \notin MO\text{-}p(proc(l),E).$

Also the shared memory condition introduces a new definition: the access order.
To specify the definition of the access order, the access order at every memory location, *Ao-m*, has to be declared first:

$Ao\text{-}m\ (m\colon location, E\colon Execution)\ R\colon OP \times OP$-**set**
post $\forall k\colon OP \times OP \cdot k \in R \;\Leftrightarrow\; \exists p\colon \mathbf{N}_1 \cdot$
$\quad k \in MO\text{-}p(p,E) \wedge (k \in Prod(Op\text{-}M(\{m\},E), Op\text{-}M(\{m\},E))).$

The previous result helps to specify the access order:

$Ao\ (E\colon Execution)\ R\colon OP \times OP$-**set**
post $\forall k\colon OP \times OP \cdot k \in R \;\Leftrightarrow\; \exists m\colon location \cdot k \in Ao\text{-}m(m,E).$

5.3.7 Synchronization Operations

To describe most conditions the memory model has to comply with, synchronization operations and some relations grouping these operations are needed. The synchronization operations have been specified before, now we still have to consider the relations.
The first relation concerns barrier operations, the second relation groups acquire and release operations which belong together:

$$\begin{array}{l} N\ (E{:}\ \mathit{Execution})\ N{:}\ OP \times OP\text{-}\mathbf{set} \\ \mathbf{post}\ \forall k{:}\ OP \times OP \cdot k \in N \ \Leftrightarrow \\ \qquad (k \in \mathit{Prod}(\mathit{Bar}(E), \mathit{Bar}(E))) \wedge \mathit{id}(\mathit{fst}(k)) = \mathit{id}(\mathit{snd}(k)) \\ \qquad \vee\ (k \in \mathit{Prod}(s\text{-}Op(E), s\text{-}Op(E))) \wedge \mathit{fst}(k) = \mathit{snd}(k); \end{array}$$

$$\begin{array}{l} N'\ (E{:}\ \mathit{Execution})\ N'{:}\ OP \times OP\text{-}\mathbf{set} \\ \mathbf{post}\ \forall k{:}\ OP \times OP \cdot k \in N' \\ \qquad \Leftrightarrow\ (k \in \mathit{Prod}(\mathit{Acq}(E), \mathit{Rel}(E))) \wedge \mathit{id}(\mathit{fst}(k)) = \mathit{id}(\mathit{snd}(k)). \end{array}$$

And finally our goal is reached, a memory model can be specified !

5.3.8 Memory Model

A memory model is a set of executions, therefore the type *Model*, every memory model can be assigned to is:

$\mathit{Model} = \mathit{Execution}\text{-}\mathbf{set}$.

Of course all the executions which are an element of a memory model need to satisfy a few conditions. To make sure that these conditions are satisfied the set of basic rules of the formal theory corresponding to the specification of the memory model has to be extended.

5.4 A Formal Theory for Shared Memory Synchronization

5.4.1 The Formal Language

In the previous section a VDM specification for the formalization of a memory model has been presented. To determine the formal language of the formal theory corresponding to this specification, Mural comes to assist. More precisely, the translation from the specification to its corresponding formal theory happens automatically in Mural.
The automatic translation of our specification generates the demanded formal language together with a set of inference rules. However this set of inference rules is

not sufficient to establish the formal theory of the considered memory model, as will be explained hereafter.

5.4.2 The Set of Inference Rules

The set of inference rules produced by Mural contains all the basic rules and some derived rules, known as proof obligations, corresponding with the translated specification. However, the specification presented in the previous section does not contain all the information available on the memory model since the ten conditions, the model has to satisfy, were not added to the specification. As mentioned before, these conditions have to be annexed to the set of basic rules.
Therefore we have to rewrite them in a more formal way:

Condition 1 *uniprocessor data dependences*:

$$\boxed{\text{condition 1}}\ \frac{\begin{array}{c} E{:}\, Execution,\, m{:}\, location,\, m \in Mem,\, op1{:}\, OP,\, op2{:}\, OP, \\ op1 \in Op\text{-}m(m,E),\, op2 \in Op\text{-}m(m,E),\, p{:}\,\mathbf{N}_1,\, p \le s\text{-}n(E) \end{array}}{((op1, op2) \in Po\text{-}p(p,E)) \;\Rightarrow\; ((op1, op2) \in MO\text{-}p(p,E))}$$

Condition 2 *value condition*:

$$\boxed{\text{condition 2}}\ \frac{\begin{array}{c} l{:}\,(OPl \mid OPf),\, s{:}\,(OPs \mid OPf), \\ E{:}\, Execution,\, l \in L(E),\, s \in prev(l,E) \end{array}}{val\text{-}l(l) = val\text{-}s(s)}$$

Condition 3 *shared memory condition.*
This condition describes two properties of the memory model. The first property expresses that the memory order relations on all the several processors must be consistent for all memory locations:

$$\boxed{\text{condition 3A}}\ \frac{\begin{array}{c} E{:}\, Execution,\, m{:}\, location,\, m \in Mem,\, R{:}\,(((OP \times OP)\text{-}\mathbf{set})\text{-}\mathbf{set}), \\ R = \{Ri{:}\,(OP \times OP)\text{-}\mathbf{set} \mid \\ \exists p{:}\,\mathbf{N}_1 \cdot (p \le s\text{-}n(E)) \wedge (Ri = MO\text{-}p(p,E))\}) \end{array}}{consistent\text{-}Rel(R, Op\text{-}m(m,E))}$$

The second property demands that the access order is a partial order relation:

$$\boxed{\text{condition 3B}}\ \frac{E{:}\, Execution}{PartOrder(Ao(E), s\text{-}Op(E))}$$

Condition 4 *order of synchronization operations.*
The fact that barrier, acquire, release and low-level synchronization operations are totally ordered per memory location is expressed in a first rule:

$$\boxed{\text{condition 4A}}\;\dfrac{\begin{array}{c} m\colon \mathit{location}, m \in \mathit{Mem}, E\colon \mathit{Execution},\\ R\colon ((((OP\text{-}\mathbf{set}) \times (OP\text{-}\mathbf{set}))\text{-}\mathbf{set})\text{-}\mathbf{set}),\\ R = (\{Ri\colon ((OP\text{-}\mathbf{set}) \times (OP\text{-}\mathbf{set}))\text{-}\mathbf{set} \mid \exists p\colon \mathbf{N}_1 \cdot\\ (p \le s\text{-}n(E)) \wedge (Ri = \mathit{QuotRel}(MO\text{-}p(p,E), N(E), s\text{-}Op(E)))\}) \end{array}}{\mathit{seq}\text{-}\mathit{const}\text{-}\mathit{Rel}(R, \mathit{partition}(\mathit{Sync}\text{-}m(m,E), N(E)))}$$

A second rule involves only acquire and release operations:

$$\boxed{\text{condition 4B}}\;\dfrac{\begin{array}{c} m\colon \mathit{location}, m \in \mathit{Mem}, E\colon \mathit{Execution},\\ R\colon ((((OP\text{-}\mathbf{set}) \times (OP\text{-}\mathbf{set}))\text{-}\mathbf{set})\text{-}\mathbf{set}),\\ R = (\{Ri\colon ((OP\text{-}\mathbf{set}) \times (OP\text{-}\mathbf{set}))\text{-}\mathbf{set} \mid \exists p\colon \mathbf{N}_1 \cdot\\ (p \le s\text{-}n(E)) \wedge (Ri = \mathit{QuotRel}(MO\text{-}p(p,E), N'(E), s\text{-}Op(E)))\}) \end{array}}{\mathit{seq}\text{-}\mathit{const}\text{-}\mathit{Rel}(R, \mathit{partition}(\mathit{Sync}\text{-}m(m,E), N'(E)))}$$

Condition 5 *correct use of acquire and release operations.*

$$\boxed{\text{condition 5A}}\;\dfrac{a\colon \mathit{acquire}, E\colon \mathit{Execution}, r1\colon \mathit{release}, r2\colon \mathit{release}}{(((a, r1) \in N'(E)) \wedge ((a, r2) \in N'(E))) \;\Rightarrow\; (r1 = r2)}$$

$$\boxed{\text{condition 5B}}\;\dfrac{r\colon \mathit{release}, E\colon \mathit{Execution}, r \in \mathit{Rel}(E)}{\exists!\, a\colon \mathit{acquire} \cdot (a, r) \in N'(E)}$$

Condition 6 *same memory locations for matching acquire and release operations.*

$$\boxed{\text{condition 6}}\;\dfrac{a\colon \mathit{acquire}, r\colon \mathit{release}, E\colon \mathit{Execution}}{((a, r) \in N'(E)) \;\Rightarrow\; (\mathit{mem}(a) = \mathit{mem}(r))}$$

Condition 7 *execution order of matching acquire and release operations.*

$$\boxed{\text{condition 7}}\;\dfrac{a\colon \mathit{acquire}, r\colon \mathit{release}, E\colon \mathit{Execution}}{((a, r) \in N'(E)) \;\Rightarrow\; ((a, r) \in s\text{-}po(E))}$$

Condition 8 *order of accesses to protected locations*
The first part deals with the situation in which an acquire operation has a matching release operation:

$$\boxed{\text{condition 8A}}\;\dfrac{a\colon \mathit{acquire}, r\colon \mathit{release}, E\colon \mathit{Execution}}{\begin{array}{c} ((a, r) \in N'(E)) \;\Rightarrow\\ (\forall op\colon OP \cdot (op \in Op\text{-}m(\mathit{mem}(a), E)) \;\Rightarrow\\ ((((a, op) \in s\text{-}Op(E)) \wedge ((op, r) \in s\text{-}Op(E))) \;\Rightarrow\\ (\forall p\colon \mathbf{N}_1 \cdot (p \le s\text{-}n(E)) \;\Rightarrow\\ (a, op) \in MO\text{-}p(p, E)) \wedge ((op, r) \in MO\text{-}p(p, E))))) \end{array}}$$

The second part concerns the situation for which an acquire has no matching release:

condition 8B
$$\frac{a\colon acquire,\ E\colon Execution}{\begin{array}{c}(\neg\,(\exists r\colon release \cdot (a,r) \in N'(E)))\ \Rightarrow\\ (\forall op\colon OP \cdot (op \in Op\text{-}m(mem(a),E))\ \Rightarrow\\ (((a,op) \in s\text{-}Op(E))\ \Rightarrow\\ (\forall p\colon \mathbf{N}_1 \cdot (p \le s\text{-}n(E))\ \Rightarrow\ ((a,op) \in MO\text{-}p(p,E)))))\end{array}}$$

Condition 9 *same set of locations per barrier.*

condition 9
$$\frac{b1\colon barrier,\ b2\colon barrier,\ E\colon Execution}{((b1,b2) \in N(E))\ \Rightarrow\ (mem(b1) = mem(b2))}$$

Condition 10 *correct barrier and operation ordering*

condition 10
$$\frac{\begin{array}{c}b1\colon barrier,\ b2\colon barrier,\ E\colon Execution,\ p\colon \mathbf{N}_1,\ p \le s\text{-}n(E),\\ op\colon OP,\ op \in Op\text{-}m(mem(b1),E),\ \neg\,(b1 = op)\end{array}}{\begin{array}{c}((b1,b2) \in N(E))\ \Rightarrow\\ ((((op,b1) \in s\text{-}Op(E))\ \Rightarrow\ ((op,b2) \in MO\text{-}p(p,E)))\\ \wedge\\ (((b1,op) \in s\text{-}Op(E))\ \Rightarrow\ ((b2,op) \in MO\text{-}p(p,E))))\end{array}}$$

Everything, that was formalized previously, is captured in the formal theory except for one fact.

Type Definition

The first type specified in this text was the type *Type*. In the section 5.2 some information about this type was given. However not all this information was added to the specification because it seemed better to express it in a basic rule:

Type definition
$$\frac{t\colon Type}{\begin{array}{c}((((t = ns) \vee (t = nl)) \vee (t = nf)) \vee\\ (((t = ss) \vee (t = sl)) \vee (t = sf))) \vee\\ (((t = acq) \vee (t = rel)) \vee (t = bar))\end{array}}$$

All these basic rules together with the basic rules and formal language generated by Mural determine the formal theory of the formalized memory model.

5.4.3 A Proof

To check the usefulness of the previous formal theory, a basic property of this memory model is studied:

property po-p 1
$$\frac{p\colon \mathbf{N}_1,\ p \le s\text{-}n(E),\ E\colon Execution}{Po\text{-}p(p,E) \subseteq s\text{-}po(E)}$$

This rule was also mentioned in section 5.2.1. It expresses that the per processor order is contained in the program order, which can be accepted intuitively.

from		
h1	$p\colon \mathbf{N}_1$	
h2	$p \le s\text{-}n(E)$	
h3	$E\colon Execution$	
1	$s\text{-}po(E)\colon ((OP \times OP)\text{-}\mathbf{set})$	s-po(Execution)-formation(h3)
2	$pre\text{-}Po\text{-}p(p, E)$	folding (h2)
3	$\exists R\colon (OP \times OP)\text{-}\mathbf{set} \cdot post\text{-}Po\text{-}p(p, E, R)$	Po-p implementability(h1, h3, 2)
4	$Po\text{-}p(p, E)\colon ((OP \times OP)\text{-}\mathbf{set})$	Po-p formation(h1, h3, 2, 3)
	from	
5.h1	$a\colon (OP \times OP)$	
5.h2	$a \in Po\text{-}p(p, E)$	
	infer $a \in s\text{-}po(E)$	lemma po-p 1(h1, h2, 5.h1, 5.h2, h3)
infer	$Po\text{-}p(p, E) \subseteq s\text{-}po(E)$	$\subseteq$-I(4, 1, 5)

Figure 5.5: Proof of 'property po-p 1'

Deriving the proof will show whether our intuition about the triviality of the property is valid or not. To construct the corresponding proof[2] of the rule the interactive theorem prover of Mural is used. We obtain the proof in Figure 5.5. In this proof we introduced a lemma that will be proven separately.

Lemma

The lemma introduced in the proof is as follows:

$$\boxed{\text{lemma po-p 1}}\ \frac{p\colon \mathbf{N}_1,\ p \le s\text{-}n(E),\ a\colon (OP \times OP),\ a \in Po\text{-}p(p, E),\ E\colon Execution}{a \in s\text{-}po(E)}$$

Proving this lemma in its turn, results in the proof in Figures 5.6 and 5.7. The fact that this proof is rather long makes us revise our first impression about the triviality of the proved property.

[2]The inference rules, from the formal theory of the memory model, used in the proof are printed in appendix B

from

h1	$p: \mathbf{N}_1$	
h2	$p \leq s\text{-}n(E)$	
h3	$a: (OP \times OP)$	
h4	$a \in Po\text{-}p(p, E)$	
h5	$E: Execution$	
1	$pre\text{-}Po\text{-}p(p, E)$	folding (h2)
2	$\exists R: (OP \times OP)\text{-}\mathbf{set} \cdot post\text{-}Po\text{-}p(p, E, R)$	Po-p implementability(h1, h5, 1)
3	$post\text{-}Po\text{-}p(p, E, Po\text{-}p(p, E))$	Po-p specification(h1, h5, 1, 2)
4	$\forall k: OP \times OP \cdot (k \in Po\text{-}p(p, E)) \;\Leftrightarrow$ $((k \in Prod(s\text{-}Op(E), s\text{-}Op(E)))\wedge$ $(((proc(\mathbf{fst}\ k) = proc(\mathbf{snd}\ k)) \wedge (proc(\mathbf{snd}\ k) = p))\wedge$ $(num(\mathbf{fst}\ k) \leq num(\mathbf{snd}\ k))))$	unfolding (3)
5	$(a \in Po\text{-}p(p, E)) \;\Leftrightarrow$ $((a \in Prod(s\text{-}Op(E), s\text{-}Op(E)))\wedge$ $(((proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a)) \wedge (proc(\mathbf{snd}\ a) = p))\wedge$ $(num(\mathbf{fst}\ a) \leq num(\mathbf{snd}\ a))))$	$\forall$-E(4, h3)
6	$(a \in Prod(s\text{-}Op(E), s\text{-}Op(E)))\wedge$ $(((proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a)) \wedge (proc(\mathbf{snd}\ a) = p))\wedge$ $(num(\mathbf{fst}\ a) \leq num(\mathbf{snd}\ a)))$	$\Leftrightarrow$ -E-left(5, h4)
7	$a \in Prod(s\text{-}Op(E), s\text{-}Op(E))$	$\wedge$-E-right(6)
8	$((proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a)) \wedge (proc(\mathbf{snd}\ a) = p))\wedge$ $(num(\mathbf{fst}\ a) \leq num(\mathbf{snd}\ a))$	$\wedge$-E-left(6)
9	$num(\mathbf{fst}\ a) \leq num(\mathbf{snd}\ a)$	$\wedge$-E-left(8)
10	$(proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a)) \wedge (proc(\mathbf{snd}\ a) = p)$	$\wedge$-E-right(8)
11	$proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a)$	$\wedge$-E-right(10)
12	$mk\text{-}Execution(s\text{-}Op(E), s\text{-}po(E), s\text{-}n(E)) = E$	Execution-introduction(h5)
13	$mk\text{-}Execution(s\text{-}Op(E), s\text{-}po(E), s\text{-}n(E)): Execution$	=-type-inherit-left(h5, 12)
14	$inv\text{-}Execution(s\text{-}Op(E), s\text{-}po(E), s\text{-}n(E))$	inv-Execution-deduction(13)
15	$s\text{-}Op(E): (OP\text{-}\mathbf{set})$	s-Op(Execution)-formation(h5)
16	$s\text{-}po(E): ((OP \times OP)\text{-}\mathbf{set})$	s-po(Execution)-formation(h5)
17	$s\text{-}n(E): \mathbf{N}_1$	s-n(Execution)-formation(h5)

Figure 5.6: First part of proof of 'lemma po-p 1'

18 $((\forall a: OP \cdot (a \in s\text{-}Op(E)) \Rightarrow (proc(a) \leq s\text{-}n(E)))\wedge$
$(\forall k: OP \times OP \cdot (k \in s\text{-}po(E)) \Leftrightarrow$
$((k \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(\mathbf{fst}\ k) = proc(\mathbf{snd}\ k))\wedge$
$(num(\mathbf{fst}\ k) \leq num(\mathbf{snd}\ k)))))): \mathbf{B}$
inv-Execution wff(15, 16, 17)

19 $inv\text{-}Execution(s\text{-}Op(E), s\text{-}po(E), s\text{-}n(E)) =$
$((\forall a: OP \cdot (a \in s\text{-}Op(E)) \Rightarrow (proc(a) \leq s\text{-}n(E)))\wedge$
$(\forall k: OP \times OP \cdot (k \in s\text{-}po(E)) \Leftrightarrow$
$((k \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(\mathbf{fst}\ k) = proc(\mathbf{snd}\ k))\wedge$
$(num(\mathbf{fst}\ k) \leq num(\mathbf{snd}\ k))))))$
inv-Execution definition(15, 16, 17, 18)

20 $(\forall a: OP \cdot (a \in s\text{-}Op(E)) \Rightarrow (proc(a) \leq s\text{-}n(E)))\wedge$
$(\forall k: OP \times OP \cdot (k \in s\text{-}po(E)) \Leftrightarrow$
$((k \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(\mathbf{fst}\ k) = proc(\mathbf{snd}\ k))\wedge$
$(num(\mathbf{fst}\ k) \leq num(\mathbf{snd}\ k)))))$ =-subs-right(b)(18, 19, 14)

21 $\forall k: OP \times OP \cdot (k \in s\text{-}po(E)) \Leftrightarrow$
$((k \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(\mathbf{fst}\ k) = proc(\mathbf{snd}\ k))\wedge$
$(num(\mathbf{fst}\ k) \leq num(\mathbf{snd}\ k))))$ $\wedge$-E-left(20)

22 $(a \in s\text{-}po(E)) \Leftrightarrow$
$((a \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a))\wedge$
$(num(\mathbf{fst}\ a) \leq num(\mathbf{snd}\ a))))$ $\forall$-E(21, h3)

23 $(a \in Prod(s\text{-}Op(E), s\text{-}Op(E))) \wedge ((proc(\mathbf{fst}\ a) = proc(\mathbf{snd}\ a))\wedge$
$(num(\mathbf{fst}\ a) \leq num(\mathbf{snd}\ a)))$ $\wedge$-I-three(7, 11, 9)

infer $a \in s\text{-}po(E)$ $\Leftrightarrow$ -E-right(22, 23)

Figure 5.7: Second part of proof of 'lemma po-p 1'

5.5 Discussion

The goal of this chapter was to prove some properties of a shared memory synchronization model in a formal way. To obtain this goal we started from the informal description of the memory model. This formalization contains all the information one has to know to compose the corresponding VDM specification. On the other hand writing down the VDM specification may help to detect some inaccuracies of the model. Having determined the VDM specification, the translation from the specification to its corresponding formal theory can be made. At this point the assistance of Mural is called in.

As mentioned before, Mural translates a VDM specification automatically to its corresponding formal theory: i.e. the system generates the formal language and the set of inference rules of the theory. The set of inference rules contains the basic rules, which define the relation between the formal language and the deduction calculus, as well as some derived rules. The derived rules are the proof obligations following from the function and constant definitions of the VDM specification. These proof obligations can be used to prove other properties of the theory before their proof has been derived. To show the correctness of the property discussed in section 5.4.3 we have made use of some proof obligations. In order to develop the proof the theorem prover available in Mural was used. This theorem prover is not at all an automatic theorem prover, it is more an interactive proof tool. This means it can give suggestions for the verification of a certain line in a proof, but most of the thinking comes from the user. However using a theorem prover is still advantageous because of the certainty one gets about the consistency of a developed proof in the considered theory.

The property we have proved, seemed to be trivial at first sight, but writing down its formal proof showed the opposite. Note that proving such theorems can be quite educative, since deriving such a proof can help the specifier to see which are the obstacles in the model's formalization to prove the property in an easier way. This information can be used to rewrite the original VDM specification as a more straightforward specification. In doing so a VDM development has been created for which a retrieve function must be defined to be sure that the new specification is a correct reification of the first. Of course, the more experience someone has, the fewer refinements he will have to write.

As a global conclusion we can state that VDM is a useful specification language. Starting from an informal text the translation to the corresponding VDM specification is straightforward. Look for example at the specification of all the different kinds of operations: to compose such a specification we only had to change the informal n-tuple, by which the operation is formalized, into a record consisting of n fields and an invariant. The invariant is required to express all the remaining information about the operation. The usefulness of a specification support tool and/or theorem prover for these kind of assignments is indisputable, even if they are only used to guarantee the consistency of the newly specified objects or derived lemmas.

5.6 Related Work

We presented a unified formalism for specifying memory models, able to represent existing memory models accurately. Since the semantics of the memory models remain unchanged when formalizing them, any program that has a defined result in the original memory model has the same result using the equivalent formalized model. Often simplifying assumptions can be made about the programs that will be executed. An assumption that holds for many parallel programs is that the parallel program is data-race free. Under this assumption, several memory models execute parallel programs in the same way. Adve and Hill present a formalization of a memory model that unifies the properties of four existing memory models for data-race free programs [1].

Our approach to memory models focuses on the result of memory operations. When implementing a shared memory, the read and write operations have to be specified in more detail than we do. The approach of Gibbons and Merritt [3] is to identify writes with a write request sent by the processor to the shared memory, and to replace read operations with a sequence of a read request sent to the memory and a read response received from the memory. A write request includes a location and a value, a read request a location, and a read response only a value. In the approach of Gibbons and Merritt, a blocking memory is a memory that refuses further requests from any processor after having received a read request and before having answered this request. In other words, the memory blocks while processing a shared-memory read.

5.7 Appendix A. A Formal Theory for Relations

5.7.1 Signature

CONSTANTS

irreflexive $\mapsto (1,0)$

QuotRel $\mapsto (3,0)$

total $\mapsto (1,0)$

PartOrder $\mapsto$ $antisymmetric(Rel\_in([\![e1]\!],[\![e2]\!])) \wedge transitive(Rel\_in([\![e1]\!],[\![e2]\!]))$

S-TotOrder $\mapsto$ $TotOrder([\![e1]\!],[\![e2]\!]) \wedge irreflexive(Rel\_in([\![e1]\!],[\![e2]\!]))$

EquivRel $\mapsto$ $reflexive(Rel\_in([\![e1]\!],[\![e2]\!])) \wedge (symmetric(Rel\_in([\![e1]\!],[\![e2]\!]))$
$\wedge\, transitive(Rel\_in([\![e1]\!],[\![e2]\!])))$

Rel_in $\mapsto (2,0)$

TotOrder $\mapsto$ $PartOrder([\![e1]\!],[\![e2]\!]) \wedge total(Rel\_in([\![e1]\!],[\![e2]\!]))$

transitive $\mapsto (1,0)$

reflexive $\mapsto (1,0)$

partition $\mapsto (2,0)$

equiv-class $\mapsto (3,0)$

seq-const-Rel $\mapsto (2,0)$

S-PartOrder $\mapsto$ $PartOrder([\![e1]\!],[\![e2]\!]) \wedge irreflexive(Rel\_in([\![e1]\!],[\![e2]\!]))$

antisymmetric $\mapsto$ $(1, 0)$

Prod $\mapsto$ $(2, 0)$

symmetric $\mapsto$ $(1, 0)$

5.7.2 Axioms

$$\boxed{\text{antisymmetric-def}}\ \frac{R\colon ((X \times X)\text{-}\mathbf{set})}{\begin{array}{c} \mathit{antisymmetric}(R) = (\forall a1\colon X \cdot \forall a2\colon X \cdot \\ ((((a1, a2)) \in R) \wedge (((a2, a1)) \in R)) \;\Rightarrow\; (a1 = a2)) \end{array}}$$

$$\boxed{\text{equiv-class-def}}\ \frac{a\colon X, Q\colon ((X \times X)\text{-}\mathbf{set}), A\colon (X\text{-}\mathbf{set}), \mathit{EquivRel}(Q, A), a \in A}{\mathit{equiv\text{-}class}(a, Q, A) = (\{a'\colon X \mid ((a, a')) \in \mathit{Rel\_in}(Q, A)\})}$$

$$\boxed{\text{irreflexive-def}}\ \frac{A\colon (X\text{-}\mathbf{set}), R\colon ((X \times X)\text{-}\mathbf{set})}{\begin{array}{c} \mathit{irreflexive}(\mathit{Rel\_in}(R, A)) = \\ (\forall a\colon X \cdot (a \in A) \;\Rightarrow\; (\neg (((a, a)) \in R))) \end{array}}$$

$$\boxed{\text{partition-def}}\ \frac{Q\colon ((X \times X)\text{-}\mathbf{set}), A\colon (X\text{-}\mathbf{set}) \mathit{EquivRel}(Q, A)}{\begin{array}{c} \mathit{partition}(A, Q) = (\{S[a]\colon X\text{-}\mathbf{set} \mid \exists a\colon X \cdot \\ (a \in A) \wedge (S[a] = \mathit{equiv\text{-}class}(a, Q, A))\}) \end{array}}$$

$$\boxed{\text{Prod-def}}\ \frac{A\colon (X\text{-}\mathbf{set}), B\colon (Y\text{-}\mathbf{set})}{\mathit{Prod}(A, B) = (\{k\colon X \times Y \mid (\mathbf{fst}\ k \in A) \wedge (\mathbf{snd}\ k \in B)\})}$$

$$\boxed{\text{Prod-form}}\ \frac{A\colon (X\text{-}\mathbf{set}), B\colon (Y\text{-}\mathbf{set})}{\mathit{Prod}(A, B)\colon ((X \times Y)\text{-}\mathbf{set})}$$

$$\boxed{\text{QuotRel-def}}\ \frac{R\colon ((X \times X)\text{-}\mathbf{set}), Q\colon ((X \times X)\text{-}\mathbf{set}), A\colon (X\text{-}\mathbf{set}), \mathit{EquivRel}(Q, A)}{\begin{array}{c} \mathit{QuotRel}(R, Q, A) = \\ (\{S\colon (X\text{-}\mathbf{set}) \times (X\text{-}\mathbf{set}) \mid \exists a1\colon X \cdot \exists a2\colon X \cdot \\ ((a1 \in A) \wedge (a2 \in A)) \\ \wedge \\ (((\mathbf{fst}\ S = \mathit{equiv\text{-}class}(a1, Q, A)) \wedge (\mathbf{snd}\ S = \mathit{equiv\text{-}class}(a2, Q, A))) \\ \wedge \\ (((a1, a2)) \in R))\}) \end{array}}$$

$$\boxed{\text{reflexive-def}}\ \frac{A\colon (X\text{-}\mathbf{set}), R\colon ((X \times X)\text{-}\mathbf{set})}{\mathit{reflexive}(\mathit{Rel\_in}(R, A)) = (\forall a\colon X \cdot (a \in A) \;\Rightarrow\; (((a, a)) \in R))}$$

$$\boxed{\text{Rel\_in-def}}\ \frac{A\colon (X\text{-}\mathbf{set}), R\colon ((X \times X)\text{-}\mathbf{set})}{\mathit{Rel\_in}(R, A) = (\{k\colon X \times X \mid k \in (R \cap \mathit{Prod}(A, A))\})}$$

$$\boxed{\text{seq-const-Rel-def}}\ \frac{R: (((X \times X)\textbf{-set})\textbf{-set}), A: (X\textbf{-set})}{\begin{array}{c} \mathit{seq\text{-}const\text{-}Rel}(R, A) = \\ (\exists S: (X \times X)\textbf{-set} \cdot \\ \mathit{TotOrder}(S, A) \\ \wedge \\ (\forall Ri: (X \times X)\textbf{-set} \cdot (Ri \in R) \ \Rightarrow\ (S = (Ri \cap \mathit{Prod}(A, A))))) \end{array}}$$

$$\boxed{\text{symmetric-def}}\ \frac{R: ((X \times X)\textbf{-set})}{\begin{array}{c} \mathit{symmetric}(R) = \\ (\forall a1: X \cdot \forall a2: X \cdot (((a1, a2)) \in R) \ \Rightarrow\ (((a2, a1)) \in R)) \end{array}}$$

$$\boxed{\text{total-def}}\ \frac{A: (X\textbf{-set}), R: ((X \times X)\textbf{-set})}{\begin{array}{c} \mathit{total}(\mathit{Rel\_in}(R, A)) = \\ (\forall a1: X \cdot \forall a2: X \cdot ((a1 \in A) \wedge (a2 \in A)) \\ \Rightarrow \\ (((((a1, a2)) \in R) \vee (((a2, a1)) \in R)) \vee (a1 = a2))) \end{array}}$$

$$\boxed{\text{transitive-def}}\ \frac{R: ((X \times X)\textbf{-set})}{\begin{array}{c} \mathit{transitive}(R) = \\ (\forall a1: X \cdot \forall a2: X \cdot \forall a3: X \cdot \\ ((((a1, a2)) \in R) \wedge (((a2, a3)) \in R)) \ \Rightarrow\ (((a1, a3)) \in R)) \end{array}}$$

5.8 Appendix B. Some Rules Used in the Proof

5.8.1 Axioms

$$\boxed{\text{Execution-introduction}}\ \frac{t: \mathit{Execution}}{\mathit{mk\text{-}Execution}(\mathit{s\text{-}Op}(t), \mathit{s\text{-}po}(t), \mathit{s\text{-}n}(t)) = t}$$

$$\boxed{\text{inv-Execution definition}}\ \frac{\begin{array}{c} \mathit{Op}: (\mathit{OP}\textbf{-set}), \mathit{po}: ((\mathit{OP} \times \mathit{OP})\textbf{-set}), n: \mathbf{N}_1, \\ ((\forall a: \mathit{OP} \cdot (a \in \mathit{Op}) \ \Rightarrow\ (\mathit{proc}(a) \leq n)) \\ \wedge \\ (\forall k: \mathit{OP} \times \mathit{OP} \cdot (k \in \mathit{po}) \ \Leftrightarrow \\ ((k \in \mathit{Prod}(\mathit{Op}, \mathit{Op})) \wedge ((\mathit{proc}(\textbf{fst } k) = \mathit{proc}(\textbf{snd } k)) \\ \wedge(\mathit{num}(\textbf{fst } k) \leq \mathit{num}(\textbf{snd } k)))))): \mathbf{B} \end{array}}{\begin{array}{c} \mathit{inv\text{-}Execution}(\mathit{Op}, \mathit{po}, n) = \\ ((\forall a: \mathit{OP} \cdot (a \in \mathit{Op}) \ \Rightarrow\ (\mathit{proc}(a) \leq n)) \\ \wedge \\ (\forall k: \mathit{OP} \times \mathit{OP} \cdot (k \in \mathit{po}) \ \Leftrightarrow \\ ((k \in \mathit{Prod}(\mathit{Op}, \mathit{Op})) \wedge ((\mathit{proc}(\textbf{fst } k) = \mathit{proc}(\textbf{snd } k)) \\ \wedge(\mathit{num}(\textbf{fst } k) \leq \mathit{num}(\textbf{snd } k)))))) \end{array}}$$

$$\boxed{\text{inv-Execution-deduction}}\ \frac{\mathit{mk\text{-}Execution}(e1, e2, e3): \mathit{Execution}}{\mathit{inv\text{-}Execution}(e1, e2, e3)}$$

$$\boxed{\text{Po-p formation}}\ \frac{p\colon \mathbf{N}_1,\ E\colon \mathit{Execution},\ \mathit{pre\text{-}Po\text{-}p}(p, E),\\ \exists R\colon (\mathit{OP} \times \mathit{OP})\text{-}\mathbf{set} \cdot \mathit{post\text{-}Po\text{-}p}(p, E, R)}{\mathit{Po\text{-}p}(p, E)\colon ((\mathit{OP} \times \mathit{OP})\text{-}\mathbf{set})}$$

$$\boxed{\text{Po-p specification}}\ \frac{p\colon \mathbf{N}_1,\ E\colon \mathit{Execution},\ \mathit{pre\text{-}Po\text{-}p}(p, E),\\ \exists R\colon (\mathit{OP} \times \mathit{OP})\text{-}\mathbf{set} \cdot \mathit{post\text{-}Po\text{-}p}(p, E, R)}{\mathit{post\text{-}Po\text{-}p}(p, E, \mathit{Po\text{-}p}(p, E))}$$

$$\boxed{\text{s-n(Execution)-defn}}\ \frac{\mathit{mk\text{-}Execution}(e1, e2, e3)\colon \mathit{Execution}}{\mathit{s\text{-}n}(\mathit{mk\text{-}Execution}(e1, e2, e3)) = e3}$$

$$\boxed{\text{s-Op(Execution)-defn}}\ \frac{\mathit{mk\text{-}Execution}(e1, e2, e3)\colon \mathit{Execution}}{\mathit{s\text{-}Op}(\mathit{mk\text{-}Execution}(e1, e2, e3)) = e1}$$

$$\boxed{\text{s-po(Execution)-formation}}\ \frac{t\colon \mathit{Execution}}{\mathit{s\text{-}po}(t)\colon ((\mathit{OP} \times \mathit{OP})\text{-}\mathbf{set})}$$

5.8.2 Proof Obligations

$$\boxed{\text{inv-Execution wff}}\ \frac{\mathit{Op}\colon (\mathit{OP}\text{-}\mathbf{set}),\ \mathit{po}\colon ((\mathit{OP} \times \mathit{OP})\text{-}\mathbf{set}),\ n\colon \mathbf{N}_1}{\begin{gathered}((\forall a\colon \mathit{OP} \cdot (a \in \mathit{Op}) \ \Rightarrow \ (\mathit{proc}(a) \leq n))\\ \wedge\\ (\forall k\colon \mathit{OP} \times \mathit{OP} \cdot (k \in \mathit{po}) \ \Leftrightarrow\\ ((k \in \mathit{Prod}(\mathit{Op}, \mathit{Op})) \wedge ((\mathit{proc}(\mathbf{fst}\ k) = \mathit{proc}(\mathbf{snd}\ k)) \wedge\\ (\mathit{num}(\mathbf{fst}\ k) \leq \mathit{num}(\mathbf{snd}\ k)))))) \colon \mathbf{B}\end{gathered}}$$

$$\boxed{\text{Po-p implementability}}\ \frac{p\colon \mathbf{N}_1,\ E\colon \mathit{Execution},\ \mathit{pre\text{-}Po\text{-}p}(p, E)}{\exists R\colon (\mathit{OP} \times \mathit{OP})\text{-}\mathbf{set} \cdot \mathit{post\text{-}Po\text{-}p}(p, E, R)}$$

5.9 Bibliography

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: a practitioners guide*, Springer-Verlag London Limited 1994.

[3] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1992.

[4] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. Mural*: a formal development support system*, Springer-Verlag London Limited 1991.

Index