第十一章：指定和验证 Memory Consistency models 和 Cache Coherence

到目前为止，我们希望让您相信 consistency models 和 cache coherence protocols 是复杂而微妙的。在本章中，我们讨论了严格指定 (specify) consistency models 和 coherence protocols 的方法，并探索它们允许的行为。我们还讨论了指定其实现和验证其正确性的方法。

我们首先简要概述指定并发系统的两种方法，操作法 (operational method) 和公理法 (axiomatic method)，并重点关注如何使用这些方法指定 consistency models 和 coherence protocols（第 11.1 节）。对于给定的形式化的 consistency 规范，我们将讨论如何自动探索 (explore) 其行为（第 11.2 节）。最后，我们将快速浏览如何验证 (validating) consistency 的实现，其中，同时涵盖了形式化方法和基于测试的方法（第 11.3 节）。

11.1 Specification

一份规范，通过精确定义系统的合法行为的集合，作为系统用户与系统实现者之间的契约。一份规范需要回答以下问题：系统允许的行为是什么？相反，一个实现需要回答以下问题：系统如何实施规定的行为？

我们所说的行为 (behavior) 是什么意思呢？是说，系统通过一组动作 (actions) 与其用户（或环境）交互：(1) *输入动作 (input actions)*：从用户到系统的动作；(2) *内部动作 (internal actions)*：系统内部发生的动作；(3) *输出动作 (output actions)*：从系统到用户的动作。其中，只有输入和输出动作对用户是可见的，而内部动作不是。因此，一系列（可观察的）输入和输出动作定义了系统的行为。我们感兴趣的是两类行为属性：*安全属性 (safety property)* 和 *活跃属性 (liveness property)*。

> 译者注：我们将 “behavior” 翻译为“行为”，将 “action” 翻译为“动作”。

安全属性断言坏事不应该发生，即，它指定了哪些可观察动作的序列是合法的。对于并发系统，由于不确定性，通常有多个合法序列。然而，安全属性不足以完全指定一个系统。考虑一个系统，它接受了一个输入动作并简单地停机 (halts) 了；尽管它没有表现出任何不良情况，但这样的系统显然没有用。这就是为什么规范还应当包括活跃属性，它断言好事最终一定会发生。

在我们进一步解释如何形式化指定行为之前，让我们首先考虑一下 consistency models 和 coherence protocols 的可观察动作是什么。

Consistency models: Observable actions and behavior

一个 memory consistency model 是软件（用户）和硬件（实现者）之间的契约。给定包含 per-thread 的 loads 和 stores 序列的多线程程序，memory model 指定每个 load 必须返回的值。因此，相关的输入动作是 loads 和 stores（以及它们的相关参数，包括处理器的 core ID、地址、以及在一次 store 中所要存储的值）。输出动作是响应 loads 的返回值。这些可观察的动作（stores、loads 和返回值）的序列，代表了 memory consistency model 的行为。

Coherence protocols: Observable actions and behavior

回想一下，处理器核心流水线与 coherence protocol 交互，以共同实现所需的 memory consistency model。因此，coherence protocol 的“用户”是与其交互的流水线，具体通过以下两个输入动作：(1) 程序中每个 load 的 read-request；(2) 程序中每个 store 的 write-request。Coherence protocol 的输出动作是：(1) read-return，为每个 read-request 返回一个值；(2) write-return，简单地对 write-request 进行确认 (acknowledge)。（流水线必须知道一个 write 何时完成。）这些可观察的动作的序列，代表了 coherence protocol 的外部的可观察行为。

重要的是，要注意 coherence protocol 与 consistency model 中，可观察的内容之间的区别。在 coherence protocol 中，一个 read-request 或 write-request 返回的瞬间是可观察的。然而，对于 consistency model，一个 load 或 store 返回的瞬间是不可观察的，只有 loads 返回的值是可观察的。

接下来，我们讨论指定系统行为的两种主要方法：*操作法 (operational method)* 和*公理法 (axiomatic method)*。在前者中，系统使用一个抽象参考实现 (abstract reference implementation) 来描述，而在后者中，数学公理 (mathematical axioms) 用于描述系统的行为。

11.1.1 Operational Specification

一份操作性的规范 (operational specification) 使用参考实现 (reference implementation) 来描述系统的行为，通常以状态机的形式表示。参考实现展示的行为，即输入/输出动作的序列，*定义 (define)* 了系统所有合法行为的集合。操作模型 (operational models) 通常使用内部状态和内部动作来约束系统的行为，从而确保安全性 (safety)。操作模型的活跃性 (liveness) 取决于状态变化最终必须发生的事实。通常，活跃性是在外部（在状态机规范之外）表达的，它会使用以*时间逻辑 (temporal logic)* [31] 编写的数学公理。

> 译者注：我们将 “temporal logic” 翻译为“时间逻辑”，以区别于通常被翻译为“时序逻辑”的 “sequential logic”。

Specifying consistency models and coherence protocols operationally

可以使用抽象实现 (abstract implementations) 来操作性地指定 consistency models。与它们的现实对应物一样，抽象实现通常同时具有处理器核心流水线组件 (processor core pipeline component) 和内存系统组件 (memory system component)。回想一下，内存系统组件与 coherence protocol 具有相同的接口（即，相同的可观察动作的集合，其中，可观察的动作是指：read-request、read-return、write-request、write-return），内存系统组件指定了 coherence protocol。

在下文中，我们将描述 sequential consistency (SC) 的两种操作性模型。两种模型具有相同的流水线组件，但它们的内存系统组件不同。前者指定了一个 consistency-agnostic coherence protocol，而后者指定了一个 consistency-directed coherence protocol。这将使我们能够操作性地描述两者之间的差异。

SC Operational Spec1: In-order pipeline + atomic memory

举个例子，一份操作性的 SC 规范 (operational SC specification)，类似于第 3.6 章 中描述的朴素的 SC 实现 (the switch)。除了可观察的动作（stores、loads 和返回值）之外，该模型还使用内部状态（内存）来约束 loads 可以读取的值。

操作性规范的工作方式如下。（假设每个核心都有一个程序计数器，指向下一条要取得的指令。）

- Step 1: *Fetch*. 其中一个处理器核心被非确定性地选中；该核心取得它的下一条指令，并将其插入局部指令队列。
- Step 2: *Issue from the pipeline*. 再一次地，一个核心被非确定性地选中；该核心从其指令队列中解码下一条指令。如果是内存指令，则流水线会发出 read-request (for a load) 或 write-request (for a store)，并阻塞。
- Step 3: *Atomic memory system*. 收到 read-request 后，内存系统会读取内存中的相应位置，并以返回值响应；收到 write-request 后，内存系统会将其写入内存，并以 ack 响应。
- Step 4: *Return to the pipeline*. 流水线在收到响应后解除阻塞。其中，store 会以 ack 响应，load 会以返回值响应。

本规范产生的行为，是遵循程序顺序的输入动作 (load/store at Step 1)、和输出动作 (value returned for a load at Step 4) 构成的序列。很容易看出它的行为满足 SC。例如，在表 11.1 的消息传递示例中，以下序列是可观察的：

«S1, S2, L1:r1=SET, L2:r2=NEW»

同样地，L1 重复一次或多次，直到它看到 SET 的值，这种序列也是可能的。但是，以下 SC 违例是不可观察的：

«S1, S2, L1:r1=SET, L2:r2=0»

Table 11.1: Message passing example

Core C1	Core C2	Comments
S1: St data = NEW; S2: St flag = SET;	L1: Ld r1 = flag; B1: if(r1 ≠ SET) goto L1; L2: Ld r2 = data;	// Initially data and flag are 0. L1, B1 may repeat many times.

SC Operational Spec2: In-order pipeline + buffered memory

现在，让我们来考虑另一个 SC 操作性模型 (SC operational model)。其中，原来的原子内存，现在被添加了一个全局 FIFO 存储队列 (global FIFO store queue) 作为前端 (frontend)，以形成一个缓冲内存系统 (buffered memory system)。存储队列中的每个条目都包含地址、要存储的值、以及发射 write request 的 core ID。缓冲内存系统使用两种类型的 acks 来响应 write request：第一种 ack 表示请求已插入写入队列，另一种 late-ack 表示请求已真正写入内存。更具体地来说，缓冲内存系统的工作方式如下。（Step 1 和 4 与之前的规范相同，因此省略。）

- Step 2': *Issue from the pipeline*. 一个核心被非确定性地选中；该核心从其指令队列中解码下一条指令。如果是 store 指令，则流水线会发出 write-request，并阻塞；如果是 load 指令，则流水线会等待，直到最新的 write 被 late-acked，然后才发出 read-request，并阻塞。
- Step 3': *Buffered memory system*. 收到 write-request 后，缓冲内存系统将其插入全局存储队列并使用 ack 响应流水线。（当 write 最终被写入内存时，late-ack 被发送到发起 write 的处理器核心。）在收到 read-request 后，从内存中读取值并返回到流水线。

尽管有缓冲，但该模型产生的行为满足 SC。回到表 11.1 所示的例子：

«S1, S2, L1:r1=SET, L2:r2=0»

上面的 SC 违例是不可观察的，因为全局存储队列是一个 FIFO，因此将 stores 提交到内存并不会违反 per-core 的程序顺序。

Consistency-agnostic vs. Consistency-directed coherence

回想一下，在第 2 章中，我们将 coherence 接口分为两类：consistency-agnostic 和 consistency-directed。原子内存系统 (Step 3 from Spec1) 指定了一种 consistency-agnostic 协议，因为 writes 是同步发生的。而缓冲内存系统 (Step 3') 是 consistency-directed 协议规范的一个示例，因为 writes 是异步的。（比后者的更激进的规范也是可能的 [3]。）尽管这两种规范都与顺序流水线相结合，以强制实施 SC，但这两种协议表现出了不同的行为。考虑表 11.2 中所示的外部可观察的动作的序列。

«write-request(X,1), write-return(X,1),read-request(X), read-return(X,0)»

Table 11.2: Linearizability vs. SC: whereas SC allows for the read to return a 0, linearizability does not.

Time	Core C1	Core C2
t_0	Write-request (X,1)	
t_1	Write-return (X,1)→ok	
t_2		Read-request (X)
t_3		Read-return (X)→0

在原子内存系统中，不能观察到上述序列，尽管该序列不违反 SC。这是因为，原子内存规范确保 write 在其调用 (invocation, t0) 和响应 (response, t1) 之间，已经被真正地写入了内存，从而确保 read 将返回 1 而不是 0。但是，缓冲内存规范允许此行为。因为，当 read 发生时，对 X 的 write 可以缓冲在全局队列中，从而允许 read 看到 0。

原子内存系统 (or consistency-agnostic coherence) 可以使用比 SC 更强的正确性条件进行建模，称为*可线性化性 (linearizability)* [16]。除了 SC 规则之外，可线性化性要求 write 必须在其调用和响应之间实时生效。可线性化性是一个*可组合 (composable)* 的属性：一个对象是可线性化的，当且仅当其组件是可线性化的。回到我们的场景，一个内存系统是可线性化的，当且仅当它的每个位置都是单独可线性化的。因此，可以在每个内存位置的基础上，指定 consistency-agnostic coherence，即每个内存位置的可线性化性。这也解释了为什么可以使用诸如分布式目录，强制实施各个单独位置的 coherence。

相反，SC 不是一个可组合的属性。即使所有内存位置单独地满足 SC（并且流水线不对内存操作进行重新排序），也不意味着整个系统就是 SC。回想一下，在第 2 章（侧边栏）中，我们研究了 coherence 的 consistency-like 的定义，其中，coherence 被定义为每个内存位置基础上的 SC。然而，这种 coherence 的定义并没有完全指定 coherence 协议。（因为可线性化性强于 SC，所以 per-location SC 不够强、以指定 consistency-agnostic coherence。因为 SC 不是可组合的，所以 per-location SC 也不够强、以指定 consistency-directed coherence。）然而，per-memory location SC 是一种对于 coherence 有用的安全性检查。回想一下，consistency-agnostic protocol 满足 per-memory location 的可线性化性。因此，任何 consistency-agnostic protocol，必然会满足 per-memory location SC。即使对于 consistency-directed coherence 协议，per-location SC 也是一种有用的安全性检查，因为大多数 consistency models，在 per-location 的基础上，明确要求了 SC（并将其称为 “coherence” 公理！[4]）。

总之，consistency-agnostic coherence 暴露了一个原子（可线性化的）内存接口，而 consistency-directed coherence 暴露了 target consistency model 的一个接口。

Specifying implementations operationally

到目前为止，我们看到了如何操作性地指定 consistency models 和 coherence protocols。操作法的好处之一是，可以通过使用特定于实现的内部状态和动作，改进基本规范，来自然地对较低级别的实现（例如，详细的 coherence protocols）进行建模。

例如，让我们将缓存添加到上述 SC 操作性规范 (Spec1) 中，每个处理器现在都拥有一个局部缓存及其关联的 coherence controller。来自流水线的每个读/写请求，现在首先发送到局部缓存控制器，以检查 hit 还是 miss。如果 miss，则缓存控制器会通过 GetS 或 GetM 请求联系目录/内存控制器（正如我们在前面章节中看到的关于 coherence protocols 的内容）。确切的状态转换自然地取决于所采用的 coherence protocol 指定的细节，但这里的要点是，前面以表格格式讨论的 coherence protocols，本质上还是通过改进内存系统 (Step 3 or Step 3') 获得的操作性模型。同样，可以改进流水线规范，以对额外的流水线阶段进行建模，从而忠实地对流水线的实现进行建模。

Tool support

操作性模型可以直接用任何用于表达状态机的语言来表达，例如 Murphi (http://mclab.di.uniroma1.it/site/index.php/software/18-cmurphi) 或 TLA (https://lamport.azurewebsites.net/tla/tla.html)。特别地，书中的每个 coherence protocol 表都可以很容易地表示为状态机。

11.1.2 Axiomatic Specification

公理性的规范 (axiomatic specification) 是一种更抽象的、指定并发系统行为的方法，它利用了由可观察的动作（以及这些动作之间的关系）组成的数学公理，来约束允许行为的集合。

Specifying consistency models axiomatically

我们在前面的章节中用来描述 SC、TSO 和 XC 内存模型的形式化方法 (formalism) 对应于公理法 (axiomatic method)。回想一下，形式化方法由以下组件组成。

- *Observable actions*: 这些是指 loads、stores 和 loads 的返回值。
- *Relations*: 程序顺序 (program order) 关系被定义为每个核心的全序 (total order)，代表了每个核心中 loads 和 stores 出现的顺序。全局内存顺序 (global memory order) 被定义为所有核心的内存操作的全序。
- *Axioms*: 有两类公理，*安全性公理 (safety axiom)* 和 *活跃性公理 (liveness axiom)*，分别指定安全属性和活跃属性。对于 SC，存在三个安全性公理：(1) preserved program order axiom: the global memory order respects program order of each core; (2) load value axiom: the load gets the value written by the most recent store before it in global memory order; (3) the atomicity axiom: the load and store of an RMW instruction occur consecutively in global memory order. 除了这些安全性公理之外，还有一个活跃性公理，它指出 no memory operations must be preceded by an infinite sequence of other memory operations; 不正式地来说，这意味着内存操作最终必须执行，不能无限期地延迟。

> 译者注：为防止原意失真，本文不对公理进行翻译。

对于任意可观察动作的序列，如果我们能够构造一个遵循上述公理的全局内存顺序，那么它就被称为该内存模型的一个合法行为。

Specifying coherence protocols axiomatically

我们之前看到了如何操作性地指定 coherence protocols。在这里，我们将看到如何在更抽象的层次上、公理性地指定它们，这对于验证 (verification) 很有用。具体来说，我们专注于指定一份满足可线性化性的 consistency-agnostic coherence 协议。（Consistency-directed coherence 协议的指定方式与 consistency models 的指定方式类似，同样可以被公理化地指定。）回想一下，可线性化性比 SC 更强，例如，它不允许表 11.2 中所示的动作顺序。因此，本规范在 SC 之上添加了一个附加公理来约束此类行为。

- *Observable actions*: 因为我们是在指定一个 coherence protocol，所以相关的动作是 read-request、read-return、write-request、以及 write-return 事件（处理器流水线看到的 coherence protocol 接口）。
- *Internal action*: 除了可观察的动作之外，我们还添加了两个内部动作，read-perform 和 write-perform，代表 read 或 write 生效的瞬间。
- *Relations*: 与 SC 类似，全局内存顺序被定义为所有核心的 read-perform 和 write-perform 事件的全序。
- *Axioms*: 除了与 SC 相关联的三个安全性公理之外，还有第四个公理，表明 read 或 write 必须在其调用和响应之间执行；更正式地来说，a write-perform (read-perform) action must appear in between the write-request (read-request) and write-return (read-return) actions in the global memory order. 最后，就像在 SC 中一样，有一个活跃性公理，表明任何 read 或 write 请求最终都必须返回。

Specifying implementations axiomatically

我们之前看到了，如何通过扩展基本的操作性规范的内部状态和动作，来自然地、操作性地表达实现（例如，coherence protocol 的实现）。以类似的方式，也可以通过扩展基本的公理性规范，来公理化地表达实现。在本节中，我们将重点关注如何按照 Lustig et al. [20] 的方式来公理性地指定处理器核心流水线。

回想一下公理性规范的抽象本质，其中 load/store 被建模为单个瞬时动作。不过，我们现在的目标有所不同。我们不是为了指定正确性，相反，我们的目标是忠实地模拟一个真实的处理器核心，其中每个 load 或 store 都经过多个流水级。因此，单个 load 或 store 动作现在被扩展为多个内部的流水线子操作 (sub-actions)，每个子操作代表一个流水级。

例如，让我们考虑一下经典的五级流水线。在这里，一个 load 被分成五个子动作：fetch、decode、execute、memory 和 writeback。一个 store 被分成七个子动作：fetch、decode、execute、memory、writeback、exit-store-buffer 和 memory-write。（“Memory” 是指流水线中的内存阶段，而 “memory-write” 是指将值实际写入内存的子动作。）

回想一下，公理性的 consistency 规范的一个关键组件是 preserved program order (ppo) axiom，它要求程序顺序必须保留在全局内存顺序中。类似于 ppo，一个想法是使用 *pipeline ordering axioms* 来指定微架构上的 happens-before 关系，即，指定流水线是否保留不同指令的子动作之间的顺序。同样地，理解 ppo 和 pipeline ordering axioms 之间的细微差别很重要。前者的目标是指定正确性，而后者的目标是忠实地建模流水线。事实上，流水线的实现是否遵守 ppo 公理（对于预期的 consistency model）是一个重要的验证 (validation) 问题，我们将在下一节中解决。

对于五级流水线，pipeline ordering axioms 如下。

- *Fetch honors program order*. If instruction i_1 occurs before i_2 in program order (po), then the fetch of i_1 happens before i_2.
  - That is: $i_1 \overset{po}{\longrightarrow} i_2 \implies i_1.\text{fetch} \longrightarrow i_2.\text{fetch}$
- *Decode is in order*. If fetch of i_1 happens before i_2, then decode of i_1 happens before i_2.
  - That is: $i_1.\text{fetch} \longrightarrow i_2.\text{fetch} \implies i_1.\text{decode} \longrightarrow i_2.\text{decode}$
- In a similar vein, execute, memory and writeback stages preserve ordering (elided).
- *FIFO store buffer*. If writeback of store i_1 happens before that of store i_2, then i_1 exits the store buffer before i_2.
  - That is: $i_1.\text{writeback} \longrightarrow i_2.\text{writeback} \implies i_1.\text{exits-store-buffer} \longrightarrow i_2.\text{exits-store-buffer}$
- *Ordered writes*. If i_1 exits the store buffer before i_2, then i_1 writes to memory before i_2.
  - That is: $i_1.\text{exits-store-buffer} \longrightarrow i_2.\text{exits-store-buffer} \implies i_1.\text{memory-write} \longrightarrow i_2.\text{memory-write}$

最后，load value axiom 指定了一些约束，这些约束限制了 load 可以从同一地址的 store 中读取哪些值。

- *Load value axiom*. If a store i_1 writes to memory before load i_2's memory stage (and there are no other memorywrites in the middle from other stores), then i_2 reads the value written by i_1.

遵守上述公理的任何流水线动作序列（以及 load 的返回值）都是合法行为。为了更好地理解这一点，请考虑表 11.3 中显示的消息传递示例。当 L1 看到新值 SET 时，L2 能看到旧值 0 吗？如果能够构建满足上述公理的流水线动作的全局序列，则五级流水线将允许此行为。如果不能，即，如果约束形成了一个环，则不允许此行为。让我们尝试构建一个全局序列。

- S1.memory-write $\longrightarrow$ S2.memory-write (the pipeline ordering axioms ensure that S1 writes to memory before S2).
- S2.memory-write $\longrightarrow$ L1.memory (S2 has to be written to memory before L1's memory stage—only then, will L1 be able to read from S2).
- L1.memory $\longrightarrow$ L2.memory (the pipeline ordering axioms ensure that L1 performs its memory stage before L2).

- L2:memory ⟶ S1:memory-write (if L2 must read the old value, and not the one produced by S1, its memory stage must have happened before S1's memory write).

Table 11.3: Message passing example

Core C1	Core C2	Comments
S1: St data = NEW; S2: St flag = SET;	L1: Ld r1 = flag; L2: Ld r2 = data;	L1 reads SET Can L2 read initial value 0?

正如我们所看到的，上面列出的约束是不可能满足的，因为它会形成一个环。因此，对于这个例子，五级流水线不允许这种行为。

在上面的讨论中，我们没有明确地对 caches 或 cache coherence 建模。Manerkar et al. [24] 展示了如何扩展流水线的公理性规范，来建模 coherence protocol/流水线交互。

总之，我们看到了如何公理性地指定流水线的实现。建模实现的主要好处是可以验证它们。在第 11.3.1 节中，我们将看到如何根据实现的公理性模型的 consistency 规范，来验证它。

Tool support

Alloy (http://alloy.mit.edu) 或 *Cat* 等规范语言 (specification languages)，在与 *Herd* 工具 (http://diy.inria.fr/herd/) 相关联后，可用于表达 consistency models 的公理性规范。 μ-spec 领域特定语言 (domain specific language) 允许公理性地指定流水线和 coherence protocol 的实现 (http://check.cs.princeton.edu/#tools)。

11.2 Exploring The Behavior of Memory Consistency Models

因为 consistency models 定义了共享内存的行为，所以对于程序员和 middleware 的编写者（例如，编译器编写者和内核开发人员）而言，了解内存模型允许哪些行为以及不允许哪些行为是至关重要的。通常，memory consistency models 由处理器供应商以白话文的形式指定，这通常是模棱两可的 [1]。形式化地指定的 memory consistency model 不仅需要是明确的，而且还需要可以自动探索行为。但在深入研究如何进行这种探索之前，让我们简要讨论一下*石蕊测试 (litmus tests)*，它们是揭示内存模型属性的简单程序。

11.2.1 Litmus Tests

Memory consistency model 可能与使用库进行同步的高层程序员不太相关。然而，它对于开发同步结构的底层程序员很重要，无论是实现语言级同步的编译器编写者、还是开发内核级同步的内核编写者、抑或是开发无锁数据结构的应用程序程序员。这样的底层程序员会想要知道内存模型表现出的行为，以便他们可以为他们手工编写的同步场景获得所期望的行为。Litmus tests 是抽象同步场景的简单多线程代码序列；因此，内存模型在这些测试中表现出的行为对底层程序员具有指导意义。例如，它们可能会揭示是否需要特定的 FENCE 才能获得所期望的行为。这些 litmus tests 也会用于处理器供应商的手册中，以解释 memory consistency model 的属性。

Examples.

回想一下，我们在前面的章节中使用过这样的 litmus tests，来解释内存模型的行为。例如，表 3.3 被称为 Dekker's litmus test，因为它抽象了 Dekker 的互斥算法中的同步场景；表 3.1 通常被称为 message passing test。考虑表 11.4 中显示的另一个例子，它被称为 "coherence" litmus test。如果一个执行 (execution) 导致 r1 读取 NEW 值、但 r2 读取旧值 0，则它违反了数据值不变量 (data-value invariant)，从而违反了 coherence litmus test。

Table 11.4: Can r1 be NEW while r2 is 0?

Core C1	Core C2	Comments
S1: x = NEW;	L1: r1 = x;	// Initially x = 0
	L2: r2 = x	

11.2.2 Exploration

给定一个内存模型的形式化规范，以及一个 litmus test，我们能否自动探索对于这个 litmus test，该内存模型所有可能的行为？幸运的是，有工具可以做到这一点。直觉上，一旦内存模型被形式化，无论是公理性地、还是操作性的，就可以详尽地枚举 litmus test 的每一个可能的计划表 (schedule)，并且对于每个这样的执行，使用数学公理（在公理性的情况下）或执行状态机（在操作性的情况下）来确定执行的输出（loads 的返回值）。

Tool support

回想一下，*herd* 工具提供了一种语言来以数学的方式表达 consistency models；该工具还有一个内置模拟器，用于探索 litmus tests 的行为。*ppcmem* 工具 (https://www.cl.cam.ac.uk/~pes20/ppcmem/help.html) 可以做类似的事情，但专用于 POWER 和 ARM 的内存模型，而 *CppMem* 工具 (http://svrpes20-cppmem.cl.cam.ac.uk/cppmem/) 面向 C/C++ 语言级内存模型。就操作性模型而言，*RMEM* 工具 (https://www.cl.cam.ac.uk/~sf502/regressions/rmem/) 具有针对 ARM（多个变体）、POWER（多个变体）、x86-TSO 和 RISC-V 的内置操作性模型，还允许探索它们的行为。

如何生成 litmus tests？它们当然可以手动生成，以测试任何有趣的内存模型特性或同步场景。此外，也有工具支持。Litmus tests 可以通过测试生成器随机生成 [14, 15]。在给定测试形状（程序顺序关系和共享内存相关性）的情况下，diy 工具 (http://diy.inria.fr/doc/gen.html) 可以帮助生成 litmus test。最后，litmusttestgen (https://github.com/nvlabs/litmustestgen) 是一种更加自动化的方法，可以为公理性地表达的内存模型规范生成一组全面的 litmus tests。

虽然 litmus tests 是传达关于内存模型的直觉的好方法，但它们通常不能作为内存模型的完整规范，因为它们留下了一些未指定的潜在行为（那些未包含在测试中的行为）。但如果有足够数量的测试，以及内存模型 (with missing pieces) 的句法模板 (syntactic template)，那么 MemSynth (http://memsynth.uwplse.org/) 展示了如何合成满足 litmus tests 的完整内存模型。

11.3 Validating Implementations

Memory consistency 的强制实施，会跨越多个子系统，包括处理器流水线、缓存、以及内存子系统。所有这些子系统必须相互合作，以强制实施所承诺的 memory consistency model。验证它们是否正确地这么做了，是本节的主题。

我们的最终目标是，详尽地验证完整的实现是否满足所承诺的内存模型。但是，现代多处理器的绝对复杂性使这成为一个非常困难的问题。因此，存在各种验证 (validation) 技术，从形式化验证 (verification) 到非形式化测试 (test)。这本身就是一个巨大的话题，可能值得单独写一本书。在这里，我们浅尝了其中的一些技术，并提供了进一步研究的指导。

11.3.1 Formal Methods

在这个类别中，一个实现的形式化模型根据规范进行验证。我们根据规范和实现是以公理性的方式、还是以操作性的方式表达，来对这些方法进行分类。然后，我们通过考虑属于该类别的一两个技术示例，来解释每个类别。最后，我们以关于操作性方法与公理性方法的一些总结性思考，来结束本节。

Operational Implementation against Axiomatic Specification

Manual proof

以操作性模型的形式给定一个实现，并且以公理性模型的形式给定其规范，则我们可以通过手动证明的方式，表明该实现满足规范中的每个公理。

例如，考虑一个系统，其中处理器流水线按程序顺序呈现 loads 和 stores，并且 cache coherence protocol 可以确保 SWMR、以及数据值不变量 (data-value invariants)。Meixner 和 Sorin [26] 展示了，只要这样的系统满足每个 SC 公理，那么它就能强制实施 SC。

有一个用于提出证明的强大模板 [30]，它遵循了 Lamport 的方法，完全地排序 (totally ordering) 了分布式系统的事件 [18]。这个想法是，为每个独立的动作（例如，在流水线中的 store 指令的一次 fetch）分配假设的时间戳，并导出存在因果关系的事件的时间戳（例如，由于 store 导致了 GetM，那么 GetM 请求获得了更高的时间戳）。在这样做的过程中，可以导出所有内存操作的全局顺序。然后检查每个 load 的值，以确定返回值是否与 load value axiom 一致 (consistent)。

Model checking

Coherence protocols 的操作性规范，通常使用模型检查器，针对公理不变量 (axiomatic invariants) 进行验证。模型检查器，例如 Murphi [12]，允许类似于前面章节中以表格格式描述的 coherence protocols，以领域特定语言表示。该语言还允许表达 SWMR 等公理。然后，模型检查器探索协议的整个状态空间，以确保不变量成立、或报告反例（违反不变量的状态序列）。然而，由于状态空间爆炸，这种显式状态的模型检查只能针对有限的实现模型（例如，两个或三个地址、值、以及处理器）进行。

有许多方法可以对抗状态空间爆炸。符号模型检查器 (symbolic model checkers) [7]，使用逻辑来操纵状态的集合（而不是单个状态），从而实现模型的缩放。有界模型检查器 (bounded model checkers) [6]，通过仅在状态序列中寻找有限长度的反例，来对详尽性做出妥协。最后，完整证明的一种方法是，对有限系统进行详尽的模型检查，然后使用*参数化 (parameterization)* [11、17] 来概括任意数量的处理器、地址等。

以下是如何使用 Murphi 对一个非平凡的 coherence protocol 进行模型检查的具体示例：(https://github.com/icsa-caps/c3d-protocol)。

Operational Implementation against Operational Specification

假设有两个状态机：Q（实现）和 S（规范）具有相同的可观察动作的集合。我们如何证明 Q 忠实地实现了 S？为了证明这一点，我们需要证明 Q 的所有行为（可观察动作的序列）都包含在 S 中。

Refinement

一种允许对 Q 和 S（而不是序列）中的状态对进行推理的程式化的证明技术称为 refinement [2]（也称为 simulation [27]）。关键思想是确定一个抽象函数 F（也称为 refinement mapping 或 simulation relation），它将实现中的每个可达状态映射到规范中的状态，就像这样：

- initial states ($q_0 \in Q$, $s_0 \in S$) are part of the mapping. That is, $F(q_0) = s_0$;
- the abstraction function preserves state transitions. That is, for every state transition in the implementation between two states q_i and q_j with some observable action a, i.e., $q_i \xrightarrow{a} q_j$, the abstraction function will lead to two states in the specification with the same observable action, i.e., $F(q_i) \xrightarrow{a} F(q_j)$.

例如，考虑一个包含缓存和全局内存的 cache coherent 系统，该系统使用基于原子总线的 MSI coherence protocol 来保持它是 coherent（实现）。回想一下，这个实现的整体状态包括所有缓存的状态（每个位置的值和 MSI 状态），以及全局内存的状态。我们可以展示，cache coherent 内存使用 refinement 来实现原子内存（规范）。为此，我们确定了一个抽象函数，将 cache coherent 内存的状

态与原子内存的状态相关联。具体来说，一个给定位置的原子内存块的值由下式给出：(1) 如果该块处于 M 状态，则为该位置的缓存块的值；(2)否则，它与每个处于 S 状态的块、以及全局内存中的值相同。然后，证明 (proof) 将查看协议中所有可能的状态转换，以检查抽象函数是否成立。例如，考虑缓存块 X 的 S $\xrightarrow{Write(X,1)}$ M 转换，其初始值为 0。对于进入 M 状态的缓存块，抽象函数成立，因为它将缓存最新值 1。对于处于状态 S 的其他缓存块，抽象函数成立的前提是，MSI 协议正确地强制实施 SWMR、并无效化所有的这些块。

因此，一个 refinement proof 不仅需要识别抽象函数，还需要对已实现系统中的状态转换进行后续推理；该过程可能会揭示其他需要证明的不变量（例如，上面揭示的 SMWR）。Kami 框架 [10] 使用 refinement proof 来表明，一个处理器的操作性规范、与 cache coherent 内存系统相结合的话，可以满足 SC。

Model checking

表明一个实现满足规范的一种方法是，用相同的输入动作序列并排运行两个状态机，并观察它们是否产生相同的输出动作。虽然这个简单的策略是合理的，但它可能并不总是成功。（当该方法说状态机是等价的时，它们肯定是等价的；但当该方法说它们不等价时，它们可能仍然是等价的。）这是因为并发系统固有的不确定性。有时，规范和实现可能表现出相同的行为，但具有不同的时间表。话虽如此，这仍然是一个非常有用的策略，尤其是当实现和规范在概念上相似时。Banks et al. [5] 采用这种策略来证明 cache coherence protocol 满足所承诺的 consistency model。

Axiomatic Implementation against Axiomatic Specification

我们看到了如何公理性地指定实现（例如核心流水线的实现）。如何形式化地验证这样一个公理性模型是否满足 consistency model 的公理性规范（注1）呢？

> 原书作者注1：本节假设根据公理性规范进行验证。用一种类似的方法，根据操作性规范来验证一个实现的公理性模型、是可能的，尽管我们知道在这个类别中没有这样的提议。

工作 [20, 21, 24] 中的 Check line 为此目的利用了详尽探索的想法。给定一个 litmus test，以及内存模型的公理性规范，我们在第 11.2 节中看到了如何在 litmus test 上详尽地探索内存模型的所有行为。对于实现的公理性规范，也可以进行这种详尽的探索。然后，可以将 litmus test 中的行为与抽象公理性模型中的行为进行比较。如果实现显示了更多的行为，则表示实现中存在错误。另一方面，如果实现显示的行为少于规范，则表明该实现是保守的。然后，对作为套件一部分的其他 litmus tests 重复相同的过程。

上述方法的一个局限性是，验证仅对已探索的 litmus tests 是详尽无遗的。完整的验证需要探索所有可能的程序。PipeProof [22] 通过对程序的符号抽象进行归纳式证明来实现这一点，该程序允许对具有任意数量指令的程序以及所有可能的地址和值进行建模。

Summary: Operational vs. Axiomatic

一般来说，公理性规范是声明性的；他们描述了哪些行为是允许的，但没有完整描述这个系统是如何实现的。通常，它们更抽象，更容易用数学表达；因此，可以更快地探索它们的行为。另一方面，操作性模型更接近于实现，对架构师来说更直观；因此，可以说，操作性地对详细的实现进行建模是更容易的。回想一下，我们能够在前面的章节中以表格格式自然地描述相当复杂的 coherence protocols。总而言之，是走公理性的路线、还是走操作性的路线，这不是一个能够简单回答的问题。

最后，重要的是，需要了解混合公理性/操作性模型也是可能的。例如，让我们考虑一个使用 coherence protocol 保持 cache coherent 的多处理器。除了操作性地描述 coherence protocol，还可以使用 SWMR 和数据值不变量来公理性地抽象这个协议。当想要将复杂的验证问题分解为更简单的问题时，这种方法可能很有吸引力。例如，当想要验证流水线和内存系统的组合是否正确地强制实施了 consistency model 时，独立验证 coherence protocol、然后通过 SMWR 来公理性地抽象它，可能是有意义的。

11.3.2 TESTING

形式化地证明实现满足规范，提供了很高的信心，即实现确实是正确的。然而，形式化验证并不是万灵药。原因有三。首先，形式化验证难以扩展。模型检查等自动技术可能无法处理复杂实现的庞大状态空间；另一方面，能够处理如此大的状态空间的技术并不是完全自动化的，对架构师来说并不容易。其次，正确地、形式化地指定一个实现并不简单，而且可能容易出错。第三，即使模型是准确的，它也可能是不完整的，因为模型可能没有考虑到实现的某些部分。这就是为什么彻底测试最终的实现是非常重要的。我们将测试分为两类：*离线测试 (offline testing)* 和 *在线测试 (online testing)*。对于前者（也称为静态测试），在部署之前测试真实或模拟的多处理器是否存在 consistency 违例。对于后者，违例会在部署后的运行时检测到。

Offline Testing

这个想法是，在真实（或模拟）机器上运行测试程序，观察它们的行为，并验证它们。该方法提出了两个问题：如何生成测试程序？ 他们的行为如何得到验证？

TSOtool [15] 是关于 memory consistency 测试的早期工作之一，它使用伪随机生成器来生成简短的测试程序。给定一个以前没见过的随机生成的程序，如何验证其观察的行为（reads 的返回值）？这是 TSOtool 解决的关键挑战。直觉上，这需要一个 *consistency checker*，它从观察中动态地重建全局顺序，并检查 consistency model 是否允许该顺序。虽然从 reads 的返回值重建全局顺序是一个 NP 完全问题，但 TSOtool 提出了一种以准确性换取性能的多项式算法（它可能会错过一些违例行为）。MTraceCheck [19] 进一步改进了这个检查器。

除了使用随机生成的程序，还可以使用 litmus tests（第 11.2.1 节）。使用 litmus tests 的好处是，一个测试套件已经可以用于大多数内存模型；此外，对于每个 litmus test，预期的行为都是已知的，这就不需要复杂的检查器来重建全局内存顺序。Litmus 工具 diy (http://diy.inria.fr/doc/litmus.html) 是使用此方法测试真实处理器的框架。

虽然 litmus testing 和随机测试 (random testing) 在后硅 (post-silicon) 环境中有效，但在慢速模拟器环境中可能效果不佳。McVersi [14] 提出了一种基于遗传编程 (genetic-programming-based) 的方法，该方法利用模拟器环境的白盒特性来提出可能更快地揭示 consistency 违例的测试程序。

Online Testing

离线测试当然是有益的，但由于测试的基本限制，它可能无法发现一些错误。此外，即使离线测试没有遗漏任何错误，硬件瞬态错误和制造错误也可能在非人工环境中导致 consistency 违例。对于在线测试（也称为动态测试），它将硬件支持添加到多处理器以检测执行期间的此类违例。一种概念上直接的方法是，像在 TSOtool 中那样，在硬件中实现 consistency checker。然而，就内存（元数据）和通信开销而言，这种方案的简单实现将是不切实际的。Chen et al. [9] 提出了一种显著降低这些成本的方案，利用了并非所有内存操作都需要跟踪的事实（例如，不需要跟踪对局部变量的 loads 和 stores）。Meixner 和 Sorin [25] 提出了一种替代策略，将验证任务减少为验证两个不变量的任务：(a) 检查 cache coherence protocol 是否正确地强制实施了 SWMR；(b) 检查流水线是否与 coherence protocol 正确交互。值得注意的是，他们表明可以在硬件中有效地检查这两个不变量。

11.4 HISTORY AND FURTHER READING

TODO

11.5 REFERENCES

[1] Linux Kernel mailing list: Spin unlock optimization. https://lists.gt.net/engine?post=105365;list=linux

[2] M. Abadi and L. Lamport. The existence of refinement mappings. Theoretical Computer Science, 82(2):253–284, 1991. DOI: 10.1109/lics.1988.5115

[3] Y. Afek, G. M. Brown, and M. Merritt. Lazy caching. ACM Transactions on Programming Languages and Systems, 15(1):182–205, 1993. DOI: 10.1145/151646.151651

[4] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Transactions on Programming Languages and Systems, 36(2):7:1–7:74, 2014. DOI: 10.1145/2627752

[5] C. J. Banks, M. Elver, R. Hoffmann, S. Sarkar, P. Jackson, and V. Nagarajan. Verification of a lazy cache coherence protocol against a weak memory model. In Formal Methods in Computer Aided Design (FMCAD), pages 60–67, Vienna, Austria, October 2–6, 2017. DOI: 10.23919/fmcad.2017.8102242

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference (TACAS), Held as Part of the Proc. of the European Joint Conferences on the Theory and Practice of Software (ETAPS), pages 193–207, Amsterdam, The Netherlands, March 22–28, 1999. DOI: 10.21236/ada360973

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In Proc. of the 27th ACM/IEEE Design Automation Conference, pages 46–51, Orlando, FL, June 24–28, 1990. DOI: 10.1109/dac.1990.114827

[8] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. In Proc. Computer Aided Verification, 14th International Conference (CAV), pages 123–136, Copenhagen, Denmark, July 27–31, 2002. DOI: 10.1007/3-540-45657-0_10