Towards a Formalization of the HSA Memory Model in the cat Language

Jade Alglave and Luc Maranget
© 2018 HSA Foundation. All rights reserved.

The contents of this document are provided in connection with the HSA Foundation specifications. This specification is protected by copyright laws and contains material proprietary to the HSA Foundation. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of HSA Foundation. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

HSA Foundation grants express permission to any current Founder, Promoter, Supporter Contributor, Academic or Associate member of HSA Foundation to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the HSA Foundation web-site should be included whenever possible with specification distributions.

HSA Foundation makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. HSA Foundation makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the HSA Foundation, or any of its Founders, Promoters, Supporters, Academic, Contributors, and Associates members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.
Acknowledgments

This document is the result of the contributions of many people. Here is a partial list of the contributors, including the companies that they represented at the time of their contribution.

**INRIA**
- Luc Maranget

**Microsoft Research Cambridge and University College London**
- Jade Alglave
## Contents

Acknowledgments ............................................................................................................. 3

About this document: Towards a Formalization of the HSA Memory Model in the cat Language ........................................................................................................... 7

Audience ............................................................................................................................... 7
Online companion materials ................................................................................................. 7
HSA Information Sources ........................................................................................................ 7
Disclaimer ............................................................................................................................... 8

1. Preamble on axiomatic models ......................................................................................... 9

1.1 Multithreaded programs ................................................................................................. 9
1.2 Control-flow semantics .................................................................................................. 11
1.3 Data-flow semantics ....................................................................................................... 11
1.4 Candidate executions ..................................................................................................... 13
  1.4.1 Events ....................................................................................................................... 13
  1.4.2 Program order .......................................................................................................... 14
  1.4.3 Read-from ................................................................................................................ 14
  1.4.4 Initial and final writes ............................................................................................... 14
  1.4.5 Scope relation .......................................................................................................... 15
1.5 Consistency specification ............................................................................................... 15

2. A glimpse of cat ................................................................................................................. 16

2.1 Flagging and forbidding the non-SC execution of MP ................................................... 17
  2.1.1 Execution characteristics ......................................................................................... 17
  2.1.2 The cat language ..................................................................................................... 18
  2.1.3 Flagging the non-SC execution of MP ................................................................. 19
  2.1.4 Forbidding the non-SC execution of MP ............................................................. 19
  2.1.5 Notion summary .................................................................................................... 20
  2.1.6 The herd7 tool ........................................................................................................ 20
2.2 Sequential consistency, per location or not .................................................................. 21
  2.2.1 Under SC ................................................................................................................ 21
  2.2.2 SC per location ....................................................................................................... 22
2.3 Using annotations ........................................................................................................... 24
  2.3.1 Annotating the MP example ................................................................................... 24
  2.3.2 Ruling out the incriminated execution ................................................................... 25
2.4 Scoped models ............................................................................................................... 26
  2.4.1 Scope relations and scope instances ..................................................................... 26
  2.4.2 Scope annotations and active instances ............................................................... 28
  2.4.3 Ruling out the incriminated execution ................................................................... 29
  2.4.4 Mixing memory order annotations, scopes, and scope annotations ....................... 29
2.5 Building the coherence order ......................................................................................... 30
  2.5.1 Example 2+2w ....................................................................................................... 30
  2.5.2 The relation co0 ..................................................................................................... 30
  2.5.3 Linearisations ......................................................................................................... 32
  2.5.4 Writes to the same location .................................................................................... 33
  2.5.5 The set of all possible coherence orders coL for all locations L ................................. 33
  2.5.6 Cross product ......................................................................................................... 33
  2.5.7 All total orders over writes to the same location .................................................... 34
  2.5.8 Benefiting from this construction of the coherence order co .................................. 34
3. A cat specification of the HSA memory model ............................................. 35
   3.1 Features and structure of the HSA model .................................................. 35
      3.1.1 Features of the HSA model .............................................................. 35
      3.1.2 The structure of the HSA model ....................................................... 35
      3.1.3 The relations that are built .............................................................. 35
      3.1.4 Organization of the remaining sections .............................................. 36
   3.2 Declaring tags, scopes, and instructions for HSA ....................................... 36
      3.2.1 Scopes ............................................................................................. 36
      3.2.2 Accesses .......................................................................................... 36
      3.2.3 Memory order annotations ............................................................... 37
      3.2.4 Scope annotations ............................................................................ 37
      3.2.5 All together ...................................................................................... 37
   3.3 Two running examples ............................................................................. 38
      3.3.1 Example 1: isa2 .............................................................................. 38
      3.3.2 Example 2: sb ............................................................................... 40
   3.4 Utilities over scopes .............................................................................. 41
      3.4.1 The set active-events ...................................................................... 42
      3.4.2 The relation active-instance ............................................................ 42
   3.5 Coherence coh ...................................................................................... 43
      3.5.1 Definition ....................................................................................... 43
      3.5.2 Examples on isa2 and sb .................................................................. 44
      3.5.3 Consistency of coh and po ............................................................... 45
   3.6 Heterogeneous happens-before hhb ........................................................... 45
      3.6.1 Scoped synchronization order ......................................................... 46
      3.6.2 Heterogeneous happens-before ....................................................... 47
   3.7 SC orders .............................................................................................. 48
      3.7.1 Example on sb ............................................................................... 49
   3.8 Data races .............................................................................................. 49
      3.8.1 Conflicts ....................................................................................... 50
      3.8.2 Races ............................................................................................. 51

4. References .................................................................................................. 53

A. Three cat7 library functions ....................................................................... 54
   A.1 Definition of fold ................................................................................... 54
   A.2 Definition of map ................................................................................. 54
   A.3 Definition of cross ............................................................................... 54

B. Bell and cat files for the HSA model ........................................................... 55
   B.1 Bell file ................................................................................................ 55
   B.2 Cat file ................................................................................................ 55
      B.2.1 Utilities: hsa-lib.cat ..................................................................... 55
      B.2.2 Handling the scope hierarchy: scopes.cat .................................... 56
      B.2.3 Coherence: coh.cat ....................................................................... 56
      B.2.4 Heterogeneous happens-before: hhb.cat ...................................... 56
      B.2.5 SC orders: sc.cat .......................................................................... 57
      B.2.6 Races: hsa-race.cat ................................................................. 57
      B.2.7 All together .................................................................................. 57

Figures

Figure 1–1 A message passing idiom in Lisa ...................................................... 9
About this document: Towards a Formalization of the HSA Memory Model in the cat Language

This document describes the HSA memory model formalization in the cat language.

The cat language is domain-specific and allows users to define axiomatic models by stating constraints over the candidate executions of a concurrent program. For a given program, candidate execution gathers a choice of control flow and data flow.

This document is organized as follows:

- Section 1 Preamble on axiomatic models (on page 9): Contains a brief overview of the notions of axiomatic models and candidate executions.
- Section 2 A glimpse of cat (on page 16): Describes the consistency model written in cat, which states constraints over these candidates and rule out some of them. This section also explains how a specification written in cat can rule out executions, and the syntax of cat.
- Section 3 A cat specification of the HSA memory model (on page 35): Presents several versions of a cat specification of the HSA memory model.

Formal syntax and semantics of the cat language are listed in Syntax and Semantics of the cat Language.

Audience

This document is written for system and component architects interested in supporting the HSA infrastructure (hardware and software) within platform designs.

Online companion materials

The herd7 tool is the simulator used, which takes as input a cat specification and a litmus test, and determines if the candidate executions of this test are allowed according to the cat specification. The semantics of cat have been implemented in the herd7 tool. The sources and documentation for the tool are available online at diy.inria.fr/tst7/doc/herd.html. You are encouraged to try out the cat files and tests shown in this document on the web interface of herd7, located at virginia.cs.ucl.ac.uk/herd. The HSA models are available as cat files at virginia.cs.ucl.ac.uk/herd?record=hsa.

HSA Information Sources

- HSA Platform System Architecture Specification Version 1.2 describes the HSA system architecture.

---

Disclaimer

Most of this document has been taken from the authors’ paper¹, currently under submission.

---

¹Jade Alglave, Patrick Cousot, and Luc Maranget. La langue au chat: cat, a language to describe consistency properties. Under submission.
1. Preamble on axiomatic models

Axiomatic models filter candidate executions of a multithread program. Such models are usually defined in three stages:

- First, an instruction semantic maps each instruction of the program to mathematical objects. This allows for definition of the control-flow semantics of a multithreaded program.
- Second, a set of candidate executions is built from the control-flow semantics. Each candidate execution represents a specific data-flow of the program, i.e., the communications that might happen between the different threads of the program.
- Third, a consistency specification decides the valid and invalid candidate executions.

1.1 Multithreaded programs

Multithreaded programs give one sequence of instructions per thread. Instructions can come from an assembly language instruction set, such as Power ISA, or be pseudo-code instructions as shown in Figure 1–1 (below).

![Figure 1–1 A message passing idiom in LISA](image)

This program is written in the homemade LISA (Litmus Instruction Set Architecture) language. The syntax of LISA is not described in this document, but a few key elements of it are summarized next.

Semantics of instructions. This document abstracts away from instruction semantics. It is assumed that an engine, or semantics, exists that can build its candidate executions from a given program. This is assumed because such instruction semantics have been implemented in the herd7 simulator for various front-ends, including LISA (the pseudo-code used in this document), IBM Power, and ARM assembly.

Intuitively, the program shown in Figure 1–1 (above) (a message-passing idiom) is made of two threads \( P_0 \) and \( P_1 \) running in parallel; they communicate via shared variables \( x \) and \( y \) that are initialized to the value 0. The thread \( P_0 \) writes the value 1 into \( x \) and the value 1 into \( y \). The thread \( P_1 \) reads \( y \) and places its value into register \( r_1 \), and reads \( x \) and places its value into register \( r_2 \).

If \( x \) is thought of as data being updated by \( P_0 \) and \( y \) as a flag being set up by \( P_0 \), it is apparent that \( P_0 \) and \( P_1 \) are in a producer-consumer relationship; it could be desirable that the consumer \( P_1 \) could access the updated data after getting the flag from the producer \( P_0 \).

Syntactically, this program shows:

- its name, \( MP \) (for “message-passing”), prefixed by the language in which the program is written, LISA:

  LISA MP

  `{ x = 0; y = 0; }
  P0 | P1
  w[] x 1 | r[] r1 y ;
  w[] y 1 | r[] r2 x ;
  exists(1:r1=1 /\ 1:r2=0)

- its prelude, between curly brackets:

  `{ x = 0; y = 0; }

- its body, made of two threads \( P_0 \) and \( P_1 \) in parallel:
1. Preamble on axiomatic models  1.1 Multithreaded programs

\[
\begin{align*}
P0 & | P1 ; \\
W[0] & x 1 | r[0] r1 y ; \\
W[0] & y 1 | r[0] r2 x ; \\
\end{align*}
\]

- its postlude (a question about the final state of the registers and shared variables after the two threads are done):
  \[
  \exists ( r1 = 1 \land r2 = 0 )
  \]

In the prelude, it is announced that the two threads \( P0 \) and \( P1 \) are sharing variables \( x \) and \( y \), and that these two variables are initialized to the value 0.

In the body, notice that the thread \( P0 \) holds two write instructions (as shown by the syntax \( W[0] \)), these instructions being in sequence.

- The first instruction, below the process identifier \( P0 \), writes the immediate value 1 into the shared variable \( x \); the LISA syntax is \( W[] \ x \ 1 \).(The purpose of the square brackets \( [ ] \) is described in 1.4.1 Events (on page 13) and 2.3 Using annotations (on page 24).

- The second instruction on \( P0 \) writes 1 into the shared variable \( y \); the LISA syntax is \( W[] \ y \ 1 \).

The thread \( P1 \) holds two read instructions as shown by the syntax \( r[0] \) in sequence.

- The first instruction, just below the identifier \( P1 \), reads the shared variable \( y \) and places the result into register \( r1 \), private to \( P1 \); the LISA syntax is \( r[] \ r1 \ y \).

- The second instruction on \( P1 \) reads the shared variable \( x \) and places the result into register \( r2 \); the LISA syntax is \( r[] \ r2 \ x \).

In the postlude (the last line of the program, starting with \( \exists \)), a question is asked about the values in registers \( r1 \) and \( r2 \) at the end of the execution of the two threads: Is there a program execution that, in the end, \( r1 \) holds the value 1, and \( r2 \) holds the value 0? If such an execution exists, the message-passing protocol has failed. In this case, the consumer \( P1 \) could access stale data (the initial value of \( x \)), whereas it got the flag (the value 1 in \( y \)) from the producer \( P0 \).

There is no right answer to a postlude question. Answering it is a matter of:

1. Building the candidate executions of the program.
2. Using a weakly consistent specification at hand to filter out the candidate executions that are forbidden by said specification.
3. Checking the remaining candidate executions allowed by the specification to see if some can lead to the values specified in the postlude.

Building the candidate executions of the example program means trying to understand how the program can execute. The semantics of LISA are not described in this document, but a glimpse is shown in 1.2 Control-flow semantics (on the facing page), 1.3 Data-flow semantics (on the facing page), and 1.4 Candidate executions (on page 13). Section 2 A glimpse of cat (on page 16) provides some initial examples of cat specifications.

Executions are phrased in terms of:

- Events that represent, for example, register or memory accesses
- Relations between these events, for example, communications between two threads

Candidate executions are built in stages:
The control-flow semantics (see 1.2 Control-flow semantics (below)) build events (which model accesses to registers or memory) and the program order (the order in which instructions have been written in the original program).

The data-flow semantics (see 1.3 Data-flow semantics (below)) build the communications between threads, which determines where a read of a given shared variable takes its value.

The candidate executions (see 1.4 Candidate executions (on page 13)) gather control- and data-flow.

1.2 Control-flow semantics

The instruction semantics translate instructions into events, which represent memory or register accesses (reads and writes from and to memory or registers), branching decisions, or fences.

Figure 1–2 (below) shows possible control-flow semantics to the program in Figure 1–1 (on page 9). To do this:

- Each store instruction (e.g., \( w[] \times 1 \) on \( P0 \)) corresponds to a write event specifying an address and a value (e.g., \( W[] \times 1 \)).
- Each load instruction (e.g., \( r[] \times 1 \) on \( P1 \)) corresponds to a read event that specifies an address and an undetermined value (e.g., \( R[] \times ? \)). In the example, the addresses of the events are determined by the program text and the values of the writes.

```
Figure 1–2 Control-flow semantics for the message-passing pattern of Figure 1–1

\[
\begin{array}{ccc}
  \text{a: } & W[] \times 1 & \text{c: } R[] \times ? \\
  \text{po} & \downarrow & \text{po} \\
  \text{b: } & W[] \times 1 & \text{d: } R[] \times ? \\
\end{array}
\]
```

For reads, the values are determined in the next stage (see 1.3 Data-flow semantics (below)). Implicit write events \( W[] \times 0 \) and \( W[] \times 0 \) also exist, representing the initial state of \( x \) and \( y \), which are not depicted here.

The instruction semantics also define relations over these events, representing, for example, the program order within a thread, or address, data or control dependencies from one memory access to the other via computations over register values.

Figure 1–2 (above) also shows the program order relation, written \( \text{po} \), which lifts the order in which instructions have been written to the level of events. For example, the two stores on \( P0 \) in Figure 1–1 (on page 9) have been written in program order; thus their corresponding events \( W[] \times 1 \) and \( W[] \times 1 \) are related by \( \text{po} \) in Figure 1–2 (above).

To summarize: Given a program such as the one shown in Figure 1–1 (on page 9), several event graphs exist, such as the one in Figure 1–2 (above). Each graph gives a set of events representing accesses to memory and registers, and the program order between these events, including branching decisions and the dependencies.

1.3 Data-flow semantics

Data flow defines which communications, or interferences, might happen between the different threads of the program. To do so, the read-from relation \( \text{rf} \) over memory events must be defined.
For any given read, the read-from relation \( r_f \) describes from which write this read has taken its value. A read-from arrow with no source, as shown in the top left of Figure 1–3 (below), corresponds to reading from the initial state.

In Figure 1–3 (below), notice the candidate execution at the top right corner. The read \( c \) from \( y \) takes its value from the initial state, hence reads the value 0. The read \( d \) from \( x \) takes its value from the update \( a \) of \( x \) by \( P_0 \), hence reads the value 1.

The initial writes are not shown in Figure 1–3 (above), but the bottom left drawing of Figure 1–4 (below) shows a more complete picture with the initial writes. Note that the candidate execution of Figure 1–4 (below) is in violation of Lamport’s Sequential Consistency (SC). The initial writes are the writes coming from the prelude of a test, and they are gathered in the set \( IW \). Figure 1–3 (above) shows \( IW = \{ ix, iy \} \).

At this point, the following items have been illustrated:

- A given program (Figure 1–1 (on page 9))
- An event graph as given by the control-flow semantics (Figure 1–2 (on the previous page))

---

Several read-from relations describing possible communications across threads (Figure 1–3 (on the previous page))

Note that for a given control-flow semantics, there could be several suitable data-flow semantics. For example, if there were several writes to \( x \) with value 1 in the program, there would be two possible writes to give the value 1 to the read of \( x \) on \( P_1 \).

Each such object is called a candidate execution. As shown in Figure 1–3 (on the previous page), there can be more than one candidate execution for a given program. To learn more about candidate executions, see 1.4 Candidate executions (below).

1.4 Candidate executions

Candidate executions are tuples:

\[
X \in \text{Candidate} \quad \triangleq \quad \text{Evts} \times \text{Program-order} \times \text{Read-from} \times \text{Writes} \times \text{Writes} \times \text{Scope-rel}
\]

which gather a number of objects, some of which have already been seen:

- the events (reads and writes relative to memory)
- the program order \( \text{po} \) on each thread
- the read-from relation \( \text{rf} \), modelling who reads from where
- the initial writes \( \text{IW} \)
- the final writes
- a scope relation, indicating how threads are distributed along a given concurrency hierarchy as needed to model, for example, GPU models (see e.g., Alglave et al\(^1\) and 3 A cat specification of the HSA memory model (on page 35).

1.4.1 Events

Events \( \text{evts} \) gather register, write and read accesses, branch, and fence events. More precisely, an event specifies:

- its location, whether a private register \( \tau \) or a shared variable \( x \).
- its kind (\( W \) for writes, \( R \) for reads, \( B \) for branches, \( F \) for fences). \( W, R, B, F \) is written for the set of all writes, reads, branches, and fences respectively.
- its process identifier (\( \text{pid} \)); the thread it comes from.

Events can be annotated as described in 2.3 Using annotations (on page 24). This is the purpose of the square brackets in an event (for example, \( w[\] \times 1 \) is a write to address \( x \) with value 1 with no annotation, whereas \( w[\text{rel}] \times 1 \) has an annotation \( \text{rel} \)).

---

\(^1\)Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In ASPLOS, 2015.
Such annotations can then be given semantics within a consistency specification. For example, 3.2 Declaring tags, scopes, and instructions for HSA (on page 36) and 2.3 Using annotations (on page 24) show the release-acquire paradigm as used, for example, in C++\(^1\) or HSA\(^2\) as a set of constraints on “release” and “acquire” annotations.

Auxiliaries to extract components of an event \(e\) include:

\[
\begin{align*}
\text{loc-of}(e) & \triangleq \text{location of } e \\
\text{kind-of}(e) & \triangleq \text{kind of } e \\
\text{pid-of}(e) & \triangleq \text{pid of } e \\
\text{annot-of}(e) & \triangleq \text{annotations of } e
\end{align*}
\]

1.4.2 Program order

Program order, abbreviated \(\text{po}\), lifts the order in which instructions have been written in the text of a program to the level of events. For each candidate execution, program order is a total order over events within the same thread, and it cannot relate events from different threads.

1.4.3 Read-from

Read-from, abbreviated \(\text{rf}\), relates a read of a certain shared variable \(x\) to a unique write of the same variable. The read-from relation indicates who reads from where.

1.4.4 Initial and final writes

Initial and final writes are gathered in the sets \(\text{IW}\) and \(\text{FW}\) respectively.

Initial writes \(\text{IW}\) are the writes in the prelude of the program. Final writes are gathered in the set \(\text{FW}\), which can be empty or contain one write per address (picked arbitrarily in the program body).

Note: This departs from a traditional view on ordering writes to the same location. There is no built-in coherence order within candidate executions. By using the sets of initial and final writes \(\text{IW}\) and \(\text{FW}\), two relations, \(\text{co-pre}\) and \(\text{co-post}\), are built so that:

- \(\text{co-pre}\) relates the initial write of \(x\), taken from \(\text{IW}\), to all the writes in the program body:
  \[
  \text{co-pre} \triangleq \{(w, w') \in W \times W \mid \text{loc-of}(w) = \text{loc-of}(w') \land w \in \text{IW} \land w' \notin \text{IW}\}
  \]

- \(\text{co-post}\) relates the writes of \(x\) in the program body to the final write of \(x\), if any:
  \[
  \text{co-post} \triangleq \{(w, w') \in W \times W \mid \text{loc-of}(w) = \text{loc-of}(w') \land w \notin \text{FW} \land w' \in \text{FW}\}
  \]

\(\text{co-pre}\) and \(\text{co-post}\) are gathered in a relation that is written \(\text{co0}\). The user is left in charge of building a coherence order at his discretion (see example in 2.5 Building the coherence order (on page 30)).

---


1.5 Consistency specification

The consistency specification determines if each candidate execution is valid.

Traditionally, such specifications list constraints phrased in terms of acyclicity, irreflexivity, or emptiness of various combinations of the relations over events given by the candidate execution. For example, the specification could forbid a candidate execution if the candidate contains a cycle amongst a certain relation declared acyclic in the specification.

---

1Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In ASPLOS, 2015.
2. A glimpse of cat

Consistency models can be viewed as ways of constraining the semantics of a concurrent program. A consistency model enunciates constraints over the writes that a given read can see.

A semantic framework, implemented in the herd7 tool, is provided to define consistency models. The domain-specific cat language is proposed for specifying consistency models as lists of constraints over candidate executions.

This section introduces concepts that will be useful in the formalization of HSA. (See Syntax and Semantics of the cat Language Version 1.2 for the complete syntax and semantics of cat and examples of how consistency models can be written in cat.)

The HSA model\(^1\) ensures the following properties (among others described in this section):

- It ensures a property called SC per location in\(^2\), which is a feature of several models such as x86\(^3\) and IBM Power\(^4\).
- It ensures that message-passing idioms (see Figure 1–1 (on page 9)) behave so that the consumer \(P_1\) cannot read stale data in \(x\) after seeing the flag \(y\) raised by the producer \(P_0\). In other words, it forbids the non-SC execution of MP (see Figure 1–4 (on page 12)). This feature is at the heart of several models, including C++\(^5\), Java\(^6\), IBM Power, and Nvidia PTX\(^7\).
- It provides means to restore SC at each scope level.

The HSA model has the following features:

1. Accesses can be annotated to form special pairs, typically used to:
   - Forbid the non-SC execution of the message-passing idiom by forming special inter-thread communication pairs
   - Restore a strong model such as SC

2. Threads are distributed along a concurrency hierarchy delimited by scopes.

To introduce these concepts gradually, several cat specifications are provided:

\(^{2}\)Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. TOPLAS, 36(2), 2014b.
\(^{7}\)Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In ASPLOS, 2015.
2.1 Flagging and forbidding the non-SC execution of MP (below) contains a cat specification that signals the non-SC execution of MP, and forbids it.

2.2.1 Under SC (on page 21) contains a cat specification of SC.

2.2.2 SC per location (on page 22) weakens SC to hold only per location.

2.3 Using annotations (on page 24) and 2.4 Scoped models (on page 26) describe the notions of annotations on accesses and scopes (necessary to describe HSA).

In Sections 1 Preamble on axiomatic models (on page 9) and 3 A cat specification of the HSA memory model (on page 35), a somewhat traditional view on coherence is taken, in that there is a total order over writes to the same address, which intuitively represents the order in which these writes hit the memory. co is written for this order, and 2.5 Building the coherence order (on page 30) details how it is built in cat. This traditional view on coherence is not a built-in of cat, and a cat specification can be written, for example, in which the coherence is not total.

2.1 Flagging and forbidding the non-SC execution of MP

Section 2.1.1 Execution characteristics (below) characterizes the execution that leads to the message-passing protocol failing. This execution is shown in Figure 1–4 (on page 12), where the read of y by P1 reads from the write of y by P0, whereas the read of x by P0 reads from the prelude.

2.1.1 Execution characteristics

The relations between all the events of this execution are as follows:

- The two writes by P0 are in program order: \((a, b) \in \text{po}\). In addition, the two reads by P1 are in program order: \((c, d) \in \text{po}\).
- The read of \(y\) by P1 takes its value from the write of \(y\) by P0; thus these two events are in \(\text{rf}: (b, c) \in \text{rf}\). Moreover, they come from different threads so they also belong to the \(\text{ext}\) relation, which gathers all pairs of events coming from different threads: \((b, c) \in \text{rf} \cap \text{ext}\).
- The read of \(x\) by P1 takes its value from the write \(\text{ix}\) of \(x\) from the prelude: \((\text{ix}, d) \in \text{rf}\). By convention, the write \(\text{ix}\) of \(x\) from the prelude hits the memory before the write of \(x\) by P0: \((\text{ix}, a) \in \text{co}\); thus the read \(d\) of \(x\) by P1 relates to the write \(a\) of \(x\) by P0 as follows: the read \(d\) takes its value from the initialization \(\text{ix}\), which is overwritten by the update \(a\). This relation is called \(\text{from-read}\), abbreviated \(\text{fr}: (d, a) \in \text{fr}\).

In general, a new relation is defined \(\text{fr}\), from a read \(r\) to writes (e.g., \(w_1, w_2\)) that are co-after the unique write \(w_0\) such that \((w_0, r) \in \text{rf}\). Intuitively, \(\text{fr}\) relates a read \(r\) to the writes (e.g., \(w_1, w_2\)) that overwrite the values that \(r\) reads from the write \(w_0\).

Figure 2-1 (on the next page) shows a graphical representation of \(\text{fr}\). Figure 2-2 (on the next page) shows a redraw of the non-SC execution of MP, with an apparent \(\text{fr}\) arrow, but without initial writes.
2. A glimpse of cat  2.1 Flagging and forbidding the non-SC execution of MP

2.1.2 The cat language

The cat language is a homemade, domain-specific language that lets users specify such relations between events and constraints over these relations. Certain syntactic constructs of cat describe the sample specifications:

- The following built-in sets: the set of all writes $\mathbb{W}$ (amongst which the initial and final writes $\mathbb{IW}$ and $\mathbb{FW}$), the set of all reads $\mathbb{R}$, the set of all memory events $\mathbb{M}$ (such that $\mathbb{M} = \mathbb{W} \cup \mathbb{R}$), and the set of all events, denoted by the underscore symbol "_".

- The following built-in relations: the empty relation $0$, the program order $po$, the read-from $rf$, the relation $ext$ gathering events from different threads, and the relation $loc$ gathering events accessing the same shared variable.

- The new relations are defined with let or let rec operators. Unions, intersections, and sequences of relations are built with $\cup$, $\cap$, and $;$ respectively. Transitive closure is built with $+$, and transitive and reflexive closure is built with $\ast$. The complement is built with $\sim$, and the subtraction is built with $\setminus$.

- Checks are implemented by the acyclic, irreflexive, and empty constructs. The negation of such statements can be checked as well. For example, $\neg$irreflexive($r$) checks if the relation $r$ is not irreflexive (i.e., $r$ is reflexive). If the property does not hold, the candidate execution is forbidden.

For example, the read-from relation between events of different threads can be defined as follows:

```cat
let rfe = rf & ext
```

As shown, a new identifier $rfe$ (for read-from external) is declared that denotes the intersection $\&$ of the read-from $rf$ and the $ext$ relation.

---

Figure 2–1 Illustration of $fr$

Figure 2–2 The non-SC execution of MP, with $fr$ apparent
As shown in Figure 2-1 ([on the previous page]), the from-read relation has been defined and illustrated. The from-read relation is defined as follows:

```
let fr = rf^-1;co
let fre = fr & ext
```

A new identifier fr, is declared, which is bound to the expression rf^-1;co. This expression reads "one step of rf backward, then one step of co forward." In other words, the symbol \& denotes the composition, or sequence, of relations: r1; r2 is defined as the set of pairs (x, y) such that there exists an intervening z, such that (x, z) \& r1 and (z, y) \& r2. The symbol ^-1 denotes the inverse of a relation. The external from-read relation fre is declared in a similar way to rfe above.

### 2.1.3 Flagging the non-SC execution of MP

Notice that in the candidate execution being characterized ([Figure 2-2 ([on the previous page])]), there is one step of program order po on P0 ((a, b) \& po), then one step of rfe between P0 and P1 via y ((b, c) \& rf), then one more step of po on P1 ((c, d) \& po). In cat, this can be written:

```
po; rfe; po
```

More generally, sequence of steps is apparent in either program order po or external read-from rfe. This relation is called happens-before, and is abbreviated as hb. For this sample specification, hb is defined as:

```
let hb = po | rfe +
```

In other words, hb is the transitive closure + of the union of program order po and external read-from rfe; there is (a, d) \& hb.

Thus the execution under examination is such that the external from-read relation fre between P1 and P0 via x goes against happens-before hb: there is (d, a) \& fr and (a, d) \& hb. In other words, this is an execution of MP where the sequence fre; hb is not irreflexive. This can be characterized in cat using the flag mechanism:

```
flag -irreflexive fre; hb as incriminated
```

Note that this cat statement will not forbid the non-SC execution of MP shown in Figure 2-2 ([on the previous page]). Flags are merely for signaling and recording certain shapes of executions, like the one characterized step-by-step in 2.1.1 Execution characteristics ([on page 19]). Thus the statement above will simply flag the non-SC execution of MP under the name incriminated.

### 2.1.4 Forbidding the non-SC execution of MP

The "goes against" concept (or its negation) will be used often. The HSA documentation calls this "consistent," so the same terminology will be used. Two relations a and b are consistent (or equivalently that b does not go against a) when their sequence is irreflexive:

```
procedure consistent(a, b) =
  irreflexive a; b
end
```

A procedure is consistent when it takes two arguments a and b and requires the irreflexivity of the sequence a; b. Calling this procedure on fre and hb:

```
call consistent(fre, hb)
```

will forbid the incriminated execution.
2.1.5 Notion summary

Figure 2–3 (below) and Figure 2–4 (below) summarize the notions that have been introduced in two cat files. Notice the difference between flagged and non-flagged checks:

- The keyword flag records that an execution has exhibited a certain shape (here that fre goes against hb). The execution is allowed by the specification, but remarkable in the way defined by the flagged check.
- A non-flagged check rules out an execution. The execution is forbidden by the specification.

For example, the cat specification in Figure 2–3 (below) will flag the non-SC execution of MP under the name incriminated, where the specification in Figure 2–4 (below) will forbid the non-SC execution of MP.

**Figure 2–3 Flagging the incriminated execution**

"Flagging the incriminated execution"

```plaintext
let rfe = rf & ext
let fr = rf^-1;co
let fre = fr & ext
let hb = (po | rfe)+
flag ~irreflexive fre;hb as incriminated
```

**Figure 2–4 Forbidding the incriminated execution**

"Forbidding the incriminated execution"

```plaintext
let rfe = rf & ext
let fr = rf^-1;co
let fre = fr & ext
let hb = (po | rfe)+
procedure consistent(a,b) =
  irreflexive a;b
end
call consistent(fre, hb)
```

2.1.6 The herd7 tool

The herd7 tool takes as input cat files like the ones shown in Figure 2–3 (above) and Figure 2–4 (above), and a test like the MP test shown in Figure 1–1 (on page 9). You are encouraged to try out the cat files and tests shown in this document on the web interface of herd7: virginia.cs.ucl.ac.uk/herd.

The litmus test can be written in a variety of languages, including LISA (the pseudo-code used here), IBM Power, and ARM assembly.

The herd7 tool then enumerates the candidate executions of the test and decides which are allowed by the cat specification given as argument.

A cat file is composed as follows:

- A title, as in “Flagging the incriminated execution” in Figure 2–3 (above)
- A list of statements, including:
2.2 Sequential consistency, per location or not

2.2.1 Under SC

Under SC, incriminated execution would be forbidden. As shown in e.g., Alglave et al\(^1\), SC is equivalent to a specification phrased in terms of events and relations where there is no cycle in the union of the program order \(po\) and the communication relations.

Communication relations are the union of the read-from \(rf\) (modelling who reads from where), the coherence \(co\) (the order in which writes to a given address hit the memory), and the from-read \(fr\) (relating a read to all writes to the same address that overwrite the value taken by the read). In \(cat\), defining the communications \(com\) is straightforward:

```informal
let com = rf | co | fr
```

A new identifier \(com\) is declared, which is made of the union \(|\) of \(rf\), \(co\) and \(fr\) (recall how \(fr\) is defined in Figure 2-1 (on page 18), and in \(cat\), as \(rf^{\top};co\)).

Now, SC can be simply phrased as follows:

```informal
procedure sc() =
  let sc-order = (po | com)+
  acyclic sc-order
end
```

Now a procedure \(sc\) is defined, in which a local identifier \(sc-order\) is declared that is bound to a relation made of the transitive closure \(\cdot\) of the union of the program order \(po\) and the communications \(com\). The acyclicity of this relation is then required.

To apply this procedure, it needs to be called:

```informal
call sc()
```

This call will forbid the incriminated execution and let all the other executions of Figure 1-3 (on page 12) pass; a specification of SC has been built in \(cat\) as summarized in Figure 2-5 (on the next page).

Now, the message-passing idiom might be used in a weaker model than SC, where the incriminated execution in 2.1.3 Flagging the non-SC execution of MP (on page 19) would not be forbidden natively. If this incriminated execution is undesirable, synchronization must be used to forbid it. Examples of synchronization are shown in 2.3.2 Ruling out the incriminated execution (on page 25), 2.4.3 Ruling out the incriminated execution (on page 29), and 3.6.2 Heterogeneous happens-before (on page 47).

---

2.2 SC per location

SC per location is a property in most models studied (see e.g., Alglave et al\(^1\)), although it is not enforced by default. The SC per location property gives some intuition with respect to the communications defined in 2.2.1 Under SC (on the previous page) and it appears in the HSA models described in 3 A cat specification of the HSA memory model (on page 35).

Intuitively, the property SC per location says that a communication relation (the read-from \(rf\), the coherence \(co\), or the from-read \(fr\)) cannot go against the program order. More precisely, SC per location requires that any sequence of communications cannot go against the program order.

The notion of sequence of communications is formalized as the transitive closure \(complus\) of the relation \(com\):

\[
let \textit{complus} = ( \textit{rf} | \textit{co} | \textit{fr})^+ 
\]

SC per location simply says the relations \(complus\) and \(po\) are consistent in the sense of the procedure \textit{consistent} defined in 2.1.4 Forbidding the non-SC execution of MP (on page 19). This property forbids exactly the five scenarios shown in Figure 2–6 (on the facing page) (see e.g., Alglave\(^2\) and Alglave et al\(^3\). This is because \(complus\) is equal to the union of these relations: \(rf, co, fr, co; rf,\) and \(fr; rf\) (see e.g., Alglave\(^4\) for the proof). Thus, each of these five relations goes against the program order \(po\).

---

Towards a Formalization of the HSA Memory Model in the cat Language, Version 1.2

Figure 2–6 The five idioms forbidden by SC per location

Figure 2–7 (below) formalizes SC per location as follows:

- The from-read \( fr \) is defined as done previously, out of which the relation \( \text{complus} \) is built.
- The relation \( \text{complus} \) must be consistent with the program order \( \text{po} \) (in the formal sense of the procedure \( \text{consistent} \) as defined in 2.1.4 Forbidding the non-SC execution of MP (on page 19).

Figure 2–7 (below) shows a whole cat file: the first line "SC per location" is its title, and the second and third lines define \( fr \) and \( \text{complus} \). The last line calls the procedure \( \text{consistent} \) on the arguments \( \text{complus} \) and \( \text{po} \) and gives this statement a name, \text{sc-per-loc}, thanks to the \text{as} construct. This name can be used for reference later, for example, in the \text{herd7} tool.

Figure 2–7 Enforcing SC per location

"SC per location"

```
let fr = rf^-1; co
let complus = (rf | co |fr)+

call consistent(complus, po) as sc-per-loc
```

Equivalently, SC per location can be phrased as the acyclicity of the union of:

- The communications \( \text{com} \), and
- The program order restricted to accesses relative to the same shared variable, a relation called \( \text{po-loc} \)

See e.g., Alglave\(^1\) for the equivalence proof.

---

The relation \texttt{loc} gathers pairs of accesses with the same shared variable, or location. Thus the relation \texttt{po-loc} can be formalized in a straightforward way in \texttt{cat} as the intersection of the program order \texttt{po} and the relation \texttt{loc}:

\begin{verbatim}
let po-loc = po \& loc
\end{verbatim}

\texttt{SC} per location can now be phrased as shown in Figure 2–8 (below). The name \texttt{SC} per location should be less mysterious now; notice how the program order \texttt{po} as used in \texttt{SC} has been replaced by the more constrained program order per location \texttt{po-loc}.

\begin{verbatim}
Figure 2–8 Enforcing SC per location in a different, equivalent way
"SC per location bis"
let fr = rf ^-1; co
let com = rf | co | fr
let po-loc = po \& loc
procedure sc-per-loc() =
    acyclic po-loc | com
end

call sc-per-loc()
\end{verbatim}

2.3 Using annotations

Using models such as C++\(^1\) or HSA\(^2\), this section describes a specification that enforces this property: the message-passing protocol should work exclusively when the flag is passed via special accesses (for example, a release-acquire pair as in C++).

In \texttt{cat}, users can define special types of accesses by using \texttt{tags}. The tags that are going to be used must first be declared:

\begin{verbatim}
enum memory-order = 'rlx \|\| 'acq || 'rel
\end{verbatim}

This defines an enumeration type \texttt{memory-order} that contains the tags ‘rlx (relaxed), ‘acq (acquire), and ‘rel (release). These tags can then be used to specify that certain instructions can bear eponymous annotations.

For example, the following declarations:

\begin{verbatim}
instructions W[{'rlx,'rel}]
instructions R[{'rlx,'acq}]
\end{verbatim}

specify that write instructions can only bear the annotations \texttt{rlx} or \texttt{rel} (be relaxed or release accesses), while read instructions can only bear the annotations \texttt{rlx} or \texttt{acq} (be relaxed or acquire accesses).

2.3.1 Annotating the MP example

The user can then annotate instructions in a \texttt{LISA} litmus test as shown at the left of Figure 2–9 (on the facing page).

---


This new annotated test will have almost the same candidate executions as the MP example, whose four executions are shown in Figure 1–3 (on page 12). The only difference is that the events will bear the annotations of the instructions they come from. For example, the right side of Figure 2–9 (below) gives the incriminated execution for MP-relacq (where relacq stands for “release-acquire,” echoing the annotations used on the accesses to the flag y).

![Figure 2-9 Example MP-relacq and its incriminated execution](image)

### 2.3.2 Ruling out the incriminated execution

The incriminated execution can then be ruled out exclusively when the communication via y is not made through special accesses, tagged rel for writes, and acq for reads, as shown in Figure 2–10 (below).

In Figure 2–10 (below), the definitions are first recalled of rfe, fr, and fre again. Then a set is defined called Release (resp. Acquire), which gathers all the events bearing the tag ’rel (resp. ’acq), thanks to the cat primitive tag2events. The relation rfe-relacq is then built, which is the intersection of the external read-from rfe and the pairs where the write bears the tag ’rel and the read bears the tag ’acq. From the relation rfe-relacq, the relation hb-relacq is derived, which is the transitive closure + of the union of the program order po and the rfe-relacq relation.

![Figure 2-10 Ruling out the incriminated execution on MP-relacq](image)

"Ruling out the incriminated execution on MP-relacq"

```haskell
let rfe = rf & ext
let fr = rf'~1;co
let fre = fr & ext

let Release = tag2events('rel)
let Acquire = tag2events('acq)

let rfe-relacq = rfe & (Release * Acquire)
let hb-relacq = (po | rfe-relacq)+

call consistent(fre,hb-relacq) as ComHBCons
```

Figure 2–11 (on the next page) shows the rfe-relacq and hb-relacq relations on the incriminated execution of the MP-relacq example (see Figure 2–9 (above)). Note that edges are omitted that result from transitivity.

Hence calling the procedure consistent on fre and hb-relacq will forbid executions like the incriminated one studied in Figure 2–9 (above) only when the communication via y is made by a special release-acquire pair.
2.4 Scoped models

Inspired by models such as Nvidia PTX\(^1\) and HSA\(^2\), this section describes scoped models. In such models, the programmer has access to how the threads are laid out over the concurrency hierarchy. In cat, scopes are special tags and the identifier `scopes` is reserved for them. Here a set of two scopes is defined: `’wi`, which stands for work-item (a thread), and `’system`, which stands for the whole system:

```plaintext
enum scopes = ’wi || ’system
```

Scoped models are usually hierarchical, which needs to be specified using the identifiers `narrower` and `wider`. A hierarchy is shown here where `system` is the widest scope and `wi` a narrower scope:

```plaintext
let narrower(lvl) = match lvl with
  || ’system -> ’wi
end
```

The function `wider` describes the inverse:

```plaintext
let wider(lvl) = match lvl with
  || ’wi -> ’system
end
```

2.4.1 Scope relations and scope instances

These scopes can then be used to augment LISA programs with a scope tree as shown in Figure 2–12 (below). A scope tree must be in accordance with the hierarchy as defined by the functions `narrower` and `wider`.

```plaintext
Figure 2–12 The example MP-scope
LISA MP-scope
{ x = 0; y = 0; } P0
P0 |P1
w[] x 1 | r[] r1 y ;
w[] y 1 | r[] r2 x ;
scopes: (system (wi P0) (wi P1))
extists(1:r1=1 \ 1:r2=0)
```

---

\(^1\)Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In ASPLOS, 2015.

The scope tree scopes: (system (wi P0) (wi P1)) in Figure 2-12 (on the previous page) specifies that the threads P0 and P1 reside in two different scope instances of level wi. In contrast, as specified by the scope tree, there is one scope instance of level system and both threads reside in this common instance.

Given a scope tag lvl, the cat primitive tag2scope returns the pairs of events that belong to the scope relation of level lvl with respect to a given scope tree. In other words, it returns the component sr-of(X) of a given candidate execution X.

Figure 2-13 (below) shows the relations tag2scope (’system) (system for short) and tag2scope (’wi) (wi for short) in the case of the MP-scoped example (identity pairs have been removed for clarity).

Figure 2-13 Scope relations and instances on MP-scoped

![Scope relations and instances on MP-scoped](image)

Note that the relations tag2scope(lvl) are equivalence relations. The scope instances at level lvl are the equivalence classes of the scope relation tag2scope(lvl). In other words, the notions of scope relation and scope instances differ as follows:

- Scope relation of level lvl: Each scope tag (’system and ’wi) has one unique eponymous scope relation.
- Scope instances of level lvl: The scope tag ’wi has two scope instances and the scope tag ’system has one instance.

The example contains:

- One scope relation of level wi, viz., the following set containing two pairs: {(a, b), (c, d)}.
- Two scope instances of level wi, viz., the two sets containing the elements of the two pairs from the above set: {a, b} on one hand, and {c, d} on the other. In other words, the write events coming from P0 are related in one scope instance of level wi, and the read events from P1 are related in another instance of level wi.
- One scope relation of level system, viz., the following set containing six pairs: {(a, b), (a, c), (a, d), (b, c), (b, d), (c, d)}.
- One single scope instance of level system, viz., the set containing all events: {a, b, c, d}. In other words, all events are related to each other in the scope instance of level system.
2.4.2 Scope annotations and active instances

Several scoped models, including the HSA model shown in A cat specification of the HSA memory model (on page 35), use annotations to indicate if an instruction should be active at a certain scope level. More precisely, instructions can bear scope annotations, eponymous of the scope tags that define the concurrency hierarchy. For example, in this declaration, writes and reads can bear any annotation from the set scopes previously defined as {’wi, ’system}:

instructions W[scopes]
instructions R[scopes]

Thus the scoped MP example can be augmented with scope annotations as shown in Figure 2–14 (below) (see the annotations wi and system between square brackets).

Figure 2–14 The example MP-scoped-mit-scope-tags
LISA MP-scoped-mit-scope-tags
{ x = 0; y = 0; }
P0 | P1 ;
w[wi] x 1 | r[system] r1 y ;
w[system] y 1 | r[wi] r2 x ;
scopes: (system (wi P0) (wi P1))
exists(1:r1=1 /\ 1:r2=0)

Then an instruction is active at scope level lvl if:

- it resides in a scope instance of level lvl (in the sense of tag2scope), and
- it bears the scope annotation lvl (in the sense of tag2events).

This leads to the notion of active scope instance, which is a scope instance of level lvl restricted to the events that are active at level lvl. In cat, this notion is defined as follows:

let active-instance(lvl) =
tag2scope(lvl) \& (tag2events(lvl) \* tag2events(lvl))

Figure 2–15 (below) shows the active instances at level wi and system for MP-scoped-mit-scope-tags. Exceptionally shown here is identity pairs. For instance, active-wi (defined as active-instance ’wi) reduces to the pairs (a, a) and (d, d).

Figure 2–15 Active instances at level wi and system for MP
2.4.3 Ruling out the incriminated execution

Now the incriminated execution can be ruled out only when the communication via the flag \( y \) is made through accesses that belong to the same active scope instance. To do so, the union of all scope instances over all scopes must be built.

To achieve this, a function \( \text{union-scopes} \) is built, which given a function \( f \), returns \( \{ f(t) \mid t \in \text{scopes} \} \). This function is built using a \( \text{fold} \) library function which, given a function \( f \), a set \( S = \{ e_1, e_2, \ldots, e_n \} \) and an element \( y \), returns \( \text{fold}(f(e_1, f(e_2, \ldots, f(e_n, y))) \) where \( i_1, i_2, \ldots, i_n \) is a permutation of \( 1, 2, \ldots, n \). This function can be implemented in \( \text{cat} \) as defined in A.1 Definition of fold (on page 54).

The \( \text{fold} \) can then be used to build \( \text{union-scopes} \):

```plaintext
let union-scopes f = fold (fun (s,y) -> f s | y) (scopes,())
```

The function \( \text{union-scopes} \) is then used to build the union of all scoped-rfe over all scopes. Figure 2-16 (below) shows a \( \text{cat} \) specification that forbids the incriminated execution of MP-scoped-mit-scope-tags.

![Figure 2-16 Ruling out the incriminated execution on MP-scoped-mit-scope-tags](image)

In Figure 2-16 (above), a relation \( \text{scoped-rfe} \) is defined for each scope level \( \text{lvl} \), which corresponds to the external read-from relation \( \text{rfe} \) (restricted to both extremities that belong to the same active scope instance of level \( \text{lvl} \)). From \( \text{scoped-rfe} \) the \( \text{scoped-hb} \) relation is derived at each scope level, simply the transitive closure of the union of the program order \( \text{po} \) and the scoped read-from \( \text{scoped-rfe} \).

Then for each scope level in the scope set \( \text{scopes} \) (’system and ’wi), \( \text{rfe} \) must be consistent with the scoped \( \text{hb} \) relation.

Thus the incriminated execution of the example MP-scoped-mit-scope-tags should be forbidden at scope level ’wi but not at scope level ’system. Notice how the accesses over \( y \), namely the write \( b \) and the read \( c \), belong to \( \text{tag2scope} \) (’system) but not to \( \text{tag2scope} \) (’wi). Therefore \( (b, c) \) are in \( \text{scoped-rfe} \) (’system), hence in \( \text{scoped-hb} \) (’system), but not in \( \text{scoped-rfe} \) (’wi).

2.4.4 Mixing memory order annotations, scopes, and scope annotations

A next natural step is to have a specification that features memory annotations, scopes, and scope annotations. This is what the HSA model does\(^1\). The formalization is shown in 3 A cat specification of the HSA memory model (on page 35).

---

2.5 Building the coherence order

In weak memory formalizations, the coherence order is often defined as a total order over all writes to the same location. This section describes how such a co order is built in cat (see Figure 2-17 (below)). To make the exposition less abstract, the following definitions will be illustrated on the example 2+2w shown in Figure 2-18 (on page 32).

Before describing this example, the building of co from a high-level point of view is explained. To build co in cat as a total order over writes the same location:

- From the initial and final writes IW and FW, the relation co0 is built (the union of co-pre and co-post).
- The set of all writes is divided into the subsets relative to the same location. In other words, equivalence classes of the relation same-loc-writes are built, which gathers pairs of writes to the same location L.
- All the possible linearisations coL of co0 over writes are written to the same location L, (total orders extending co0 over the equivalence classes of same-loc-writes).

```plaintext
Figure 2-17 Building the coherence order—cat file building-co.cat
"Building co"
let co-pre = loc & (IW * (W\IW))
let co-post = loc & ((W\IW) * FW)
let co0 = co-pre | co-post

let makeCo(s) = linearisations(s,co0)
let same-loc-writes = loc & (W*W)
let allCoL = map makeCo (classes (same-loc-writes))
let allCo = cross allCoL
with co from allCo
```

2.5.1 Example 2+2w (below) and 2.5.2 The relation co0 (below) explain the example 2+2w and the building of co.

2.5.1 Example 2+2w

Example 2+2w consists of threads P0 and P1, which share variables x and y both initialized to 0 (see the initialization writes ix and iy in the prelude).

The thread P0 writes 2 to x (see the write a on P0) and 1 to y (see the write b on P0). The thread P1 writes 2 to y (see the write c on P1) and 1 to x (see the write d on P1). In the postlude, it is asked if the final value in both x and y can be 2. Intuitively, this would mean that the last writes to hit the memory are a for x and c for y.

The initial writes are given by the text of the program: IW = \{ix, iy\}. For final writes, an arbitrary choice must be made, so FW = {} is chosen to explore all possible final states. Note that FW = \{a, b\} would satisfy the postlude of the program.

2.5.2 The relation co0

The relation co0 gathers the union of co-pre and co-post.

The relation co-pre relates, for a given address x, the initialization write of x to all the writes of x in the body of the program. In cat, this is:

```plaintext
let co-pre = loc & (IW * (W\IW))
```
The pairs of writes are built to the same location (these pairs belong to the relation \textit{loc}) such that the first is an initial write (belongs to \textit{IW}) and the second comes from the body of the program (belongs to the set \textit{W} \setminus \textit{IW}, which gathers all writes that are not initial writes).

\textbf{2+2w} has \textit{IW} \supseteq \{i\texttt{x}, i\texttt{y}\}, and therefore has \textbf{co-pre} \supseteq \{(i\texttt{x}, a), (i\texttt{x}, d), (i\texttt{y}, b), (i\texttt{y}, c)\}. A graphical representation of \textbf{co-pre} is shown at the top of Figure 2–18 (on the next page).

The relation \textbf{co-post} relates, for a given address \texttt{x}, the writes within the body of the program to the final writes. On the example, \textbf{co-post} \supseteq \{} in \textit{cat} is:

\begin{verbatim}
let co-post = loc \& ((W\setminus FW) * FW)
\end{verbatim}

The pairs of writes are built to the same location (\textit{loc}) such that the first is not a final write (belongs to \textit{W} \setminus \textit{FW}) and the second is a final write (belongs to \textit{FW}). \textbf{2+2w} has \textbf{co-post} \supseteq \{} because \textit{FW} was chosen to be empty.
2. A glimpse of cat  2.5 Building the coherence order

Figure 2–18 How co is built as a total order over writes to the same location, on 2+2w

```
LISA 2+2w
{ ix: W[] x 0;
  iy: W[] y 0; }
PO    | P1
a: W[] x 2 | c: W[] y 2;
 b: W[] y 1 | d: W[] x 1;
exists(x=2 \&\& y=2)

IW ⊆ {ix, iy}
W ⊆ {ix, iy, a, b, c, d}
FW ⊆ { }
loc ⊆ { (ix, ix), (ix, a), (ix, d), (a, ix), (a, a), (a, d),
       (d, ix), (d, a), (d, d), (iy, iy), (iy, b), (iy, c),
       (b, iy), (b, b), (c, iy), (c, b), (c, c) }
co-pre ⊆ { (ix, a), (ix, d), (iy, b), (iy, c) }
co-post ⊆ { }
co0 ⊆ co-pre \ co-post
same-loc-writes ⊆ loc & (W * W) = loc
classes (same-loc-writes) = { {ix, a, d}, {iy, b, c} }
linearisations({ix, a, d}, co0) = { [ix; a; d], [ix; d; a] }
       where [ix; a; d] ⊆ { (ix, a), (a, a), (a, d) }
linearisations({iy, b, c}, co0) = { [iy; b; c], [iy; c; b] }
allCoL ⊆ map makeCo (classes (same-loc-writes))
       = { [{ix; a; d}, [ix; d; a]], [{iy; b; c}, [iy; c; b]] }
alCo ⊆ cross allCoL
       = { [{ix; a; d}, [iy; b; c]], [{ix; a; d}, [iy; c; b]],
           [{ix; d; a}, [iy; b; c]], [{ix; d; a}, [iy; c; b]] }
```

2.5.3 Linearisations

Recall that a goal is to build co as a total order over writes to the same location. In cat, all total orders on a certain set of events can be generated with the primitive linearisations(S,R) that takes two arguments: a set of events S and a relation R. The primitive builds R5, the restriction of R to S. If R5 is acyclic, the call to linearisations will return the set of all total orders that extend R5. If, for example, R5 has a cycle, the primitive returns the empty set.
Hence, assuming $S_l$ to be the set of all write events to location $L$, the set of all possible $\texttt{co}_L$ by the call $	exttt{linearisations}(S_l, \texttt{co0})$ can be generated:

```plaintext
let makeCo(s) = linearisations(s, co0)
```

### 2.5.4 Writes to the same location

It has been said that $co$ should be total over all writes sharing the same variable. Thus the relation $\texttt{same-loc-writes}$ is built, which gathers the pairs of writes to the same location as follows:

```plaintext
let same-loc-writes = loc & (W*W)
```

Note that $\texttt{same-loc-writes}$ is an equivalence relation.

Now the set of all possible coherence orders (all the unions of all the possible $\texttt{co}_L$ orders for all locations $L$) will be generated. This is done by using another $\texttt{cat}$ primitive, $\texttt{classes(r)}$, which takes an equivalence relation $r$ as argument and returns its equivalence classes.

### 2.5.5 The set of all possible coherence orders $\texttt{co}_L$ for all locations $L$

The set of all possible coherence orders $\texttt{co}_L$ for all locations $L$ is built as follows:

```plaintext
let allCoL = map makeCo (classes (same-loc-writes))
```

The function $\texttt{map}$ takes as argument a function $f(\texttt{here makeCo})$ and a set $\{e_1, \ldots, e_n\}$ (here $\texttt{classes (same-loc-writes)}$), and returns the set $\{f(e_1), \ldots, f(e_n)\}$. This function is not a primitive; it can be implemented in $\texttt{cat}$ and is defined in A.2 Definition of $\texttt{map}$ (on page 54).

Here the set $W$ of all writes is divided into blocks such that each block is relative to one given shared variable; this is what the call $\texttt{classes (same-loc-writes)}$ does. Example $2+2w$ has $\texttt{classes (same-loc-writes)}$ does. Example $2+2w$ has $\{\{ix, a, d\}, \{iy, b, c\}\}$.

Then for each block of this partition (thanks to the call to $\texttt{map makeCo}$), the set of all its possible coherence orders is created. Example $2+2w$ has for the variable $x$: $\texttt{linearisations}([ix, a, d], \texttt{co0}) = \{[ix; a; d], [ix; d; a]\}$. A list notation is used for a total order. For example, $[ix; a; d]$ stands for the total order $\{[ix, a], [a, d], [ix, d]\}$.

In summary, $\texttt{allCoL}$ is a set of sets of relations, each element being the set of all possible $\texttt{co}_L$ orders for a specific $L$. Example $2+2w$ has $\texttt{allCoL} = \{\{[ix; a; d], [ix; d; a]\}, \{[iy; b; c, fy], [iy; c; b, fy]\}\}$.  

### 2.5.6 Cross product

All possible unions of the $\texttt{co}_L$ for all possible locations $L$ must still be generated. This can be done with the function $\texttt{cross}$, which takes a set of sets $S = \{S_1, S_2, \ldots, S_n\}$ as argument and returns all possible unions built by picking elements from each of the $S_i$:

$$
\{ e_1 \cup e_2 \cup \ldots \cup e_n \mid e_1 \in S_1, e_2 \in S_2, \ldots, e_n \in S_n \}
$$

This function is not a primitive; it can be implemented in $\texttt{cat}$ as defined in A.3 Definition of $\texttt{cross}$ (on page 54). The set of all possible coherence orders is generated by:

```plaintext
let allCo = cross allCoL
```
2. A glimpse of cat  2.5 Building the coherence order

More precisely, the variable allCo is bound to a value that is the set of all possible coherence orders. Example 2+2w has allCo = \{\{[ix; a; d], [iy; b; c]\}, \{[ix; a; d], [iy; c; b]\}, \\
{[ix; d; a], [iy; b; c]\}, \{[ix; d; a], [iy; c; b]\}\}.

2.5.7 All total orders over writes to the same location

To account for all possible coherence orders, this set allCo must be enumerated over. The instruction with v from S will, for each e in S, evaluate the rest of the specification in an extended environment that binds v to e. co would write with co from allCo. All possible co picked in the set allCo of all coherence orders is enumerated.

Figure 2–19 (below) shows one possible choice of co amongst all the possibilities given by the set allCo: \{[ix; d; a], [iy; b; c]\} was picked. Note that this candidate execution corresponds to the postlude of the test shown in Figure 2–18 (on page 32) and is in violation of SC in that it shows a cycle in the union of program order and communications.

2.5.8 Benefiting from this construction of the coherence order co

To benefit from this construction of the coherence order co, all the previous cat specifications (see Figure 2–3 (on page 20), Figure 2–4 (on page 20), Figure 2–5 (on page 22), Figure 2–7 (on page 23), Figure 2–8 (on page 24), Figure 2–10 (on page 25), and Figure 2–16 (on page 29), should include the cat file building-co.cat shown in Figure 2–17 (on page 30).

This can be done easily in cat using the include construct. All previous cat specifications should mention the following statement:

include "building-co.cat"

Similarly, all cat specifications that use the procedure consistent (as shown in Figure 2–7 (on page 23), Figure 2–10 (on page 25), and Figure 2–16 (on page 29)) must include a library file where the procedure is defined.

Moreover, the specifications that use annotations (e.g., Figure 2–10 (on page 25)) or scopes (e.g., Figure 2–16 (on page 29)) must include a file where these annotations and scopes are defined.
3. A cat specification of the HSA memory model

This section details a cat specification of the HSA memory model.

3.1 Features and structure of the HSA model

3.1.1 Features of the HSA model

Features of the HSA model include:

- Accesses can be annotated to form special pairs. As described in 3.2 Declaring tags, scopes, and instructions for HSA (on the next page), these annotations are typically used to:
  - Forbid the incriminated execution of the message-passing idiom by forming special, inter-thread communication pairs.
  - Restore a strong model such as SC.

- Threads belong to scopes and are distributed along a concurrency hierarchy as described in 3.2 Declaring tags, scopes, and instructions for HSA (on the next page).

Note that this document omits considerations about dependencies (address, data, or control) between accesses, or about read-modify-write accesses, although the tool does handle them. The sizes of accesses are not considered. Moreover, to make the exposition a bit more light, considerations about branches and fences are omitted, although the complete HSA model handles them.

3.1.2 The structure of the HSA model

The structure of the HSA model as described in the HSA documentation is roughly as follows:

- Enforce SC per location (see 3.5.3 Consistency of coh and po (on page 45)) and 2.2.2 SC per location (on page 22).

- Forbid the incriminated non-SC execution of the message-passing idiom (see 2.1 Flagging and forbidding the non-SC execution of MP (on page 17) and Figure 2–2 (on page 18) via a special happens-before relation (see 3.6.2 Heterogeneous happens-before (on page 47)). This relation is special in two ways:
  - The extremities (the write and the read) must be annotated adequately as described in 2.3 Using annotations (on page 24).
  - The communication must be at the right scope level as described in 2.4 Scoped models (on page 26).

- Provide means to restore SC at each scope level (see 3.7 SC orders (on page 48)).

- Flag racy executions to forbid potentially racy programs (see 3.8.2 Races (on page 51)).

3.1.3 The relations that are built

The relations that are built in the three models follow the model of the HSA documentation:

---


1. The coh relation gathers the communications over shared variables. The name coh stands for coherence, although coh is not entirely identical to the coherence co0 nor to the traditional total order co over writes to the same location (as shown in 2.5 Building the coherence order (on page 30)). coh is used to:
   - Enforce SC per location (see 2.2.2 SC per location (on page 22))
   - Build a rel-acq relation (see 2.3 Using annotations (on page 24))

2. The hhb (heterogeneous happens-before) relation rules communications through annotated accesses. hhb is used to:
   - Forbid the incriminated execution of the message-passing scenario (see 2 A glimpse of cat (on page 16))
   - Rule out programs with potentially racy executions

The SC relations provide a mean to restore SC (see 2.2.1 Under SC (on page 21)) at each scope via special synchronizing accesses.

3.1.4 Organization of the remaining sections

The remaining sections are organized as follows:

- 3.2 Declaring tags, scopes, and instructions for HSA (below) – Declarations of annotations and scopes
- 3.3 Two running examples (on page 38) – Running examples
- 3.4 Utilities over scopes (on page 41) – Auxiliaries to handle the scope hierarchy
- 3.5 Coherence coh (on page 43) – Building the coh relation
- 3.6 SC orders (on page 48) – Building the SC relations
- 3.7 Heterogeneous happens-before hhb (on page 45) – Building the hhb relation
- 3.8 Data races (on page 49) – One treatment of races

3.2 Declaring tags, scopes, and instructions for HSA

Figure 3–2 (on page 38) summarizes the definitions and notions of this section.

3.2.1 Scopes

The HSA model is a scoped model. Threads are called work-items (wi), which can be gathered into waves (wave), work-groups (wg), agents (agent), and systems (system), with each scope being narrower than the next.

3.2.2 Accesses

Accesses, or operations, can be ordinary or atomic, thus can bear an annotation ordinary or an annotation atomic. However, accesses can be either ordinary or atomic, but not ordinary and atomic at the same time.

All ordinary (resp. atomic) accesses are gathered in the set Ordinary (resp. Atomic) using the cat primitive tag2events (see 2.3.2 Ruling out the incriminated execution (on page 25)).
Accesses can bear memory order annotations 3.2.3 Memory order annotations (below) and scope annotations (see 3.2.4 Scope annotations (below)). Thus programs can be written like the example MP-annots shown in Figure 3-1 (below).

**Figure 3-1 Mixing both memory orders and scope annotations, on MP**

<table>
<thead>
<tr>
<th>LISA MP-annots</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0; y = 0;</td>
</tr>
<tr>
<td>p0</td>
</tr>
<tr>
<td>w(ordinary,rlx,wi) x 53</td>
</tr>
<tr>
<td>w(atomic,screl,system) y 1</td>
</tr>
<tr>
<td>scopes: (system (wi p0) (wi p1)) exists (l: r1 = 1 \ (r2 = 0))</td>
</tr>
</tbody>
</table>

3.2.3 Memory order annotations

Accesses can bear a memory order, much like in C++, giving them various ordering properties. The accesses can be relaxed (rlx), acquire (scacq), release (screl), and both acquire and release (screl). In addition:

- Ordinary accesses can only be relaxed (bear the tag rlx).
- Atomic accesses can bear any memory order tag.
- Reads can be acquire or acquire-release, but not release.
- Writes can be release or acquire-release, but not acquire.

The set of events that are both release and acquire-release are gathered in the Release set, and the set of events that are both acquire and acquire-release are gathered in the Acquire set. Events that are either in the Release or Acquire sets are called Synchronizing.

3.2.4 Scope annotations

Accesses can also bear a scope annotation, synonymous of the ones used to describe the concurrency hierarchy. Ordinary (and thus relaxed) accesses can bear only the work-item wi tag, while atomic accesses can bear any scope tag.

Similar to the information described in 2.4 Scoped models (on page 26), using scope tags on events will indicate if they are active at a scope instance of level lvl as formalized by the active-instance notion described 3.4 Utilities over scopes (on page 41). Note that the active-instance notion presented here is a generalization of the eponymous notion described in 2.4 Scoped models (on page 26).

3.2.5 All together

Figure 3-2 (on the next page) summarizes the following information:

- The scope tags are declared scopes and the associated hierarchy via the functions narrower and wider.
- Two sorts of annotations are declared: operation-kind, which can be ordinary or atomic, and memory-order, which can be rlx, scacq, screl, or scar.
- The annotations that accesses can bear.
- The build of certain sets of events (Release is the set of all events tagged screl or scar).
### 3.3 Two running examples

The running examples used are the tests isa2 and sb, shown below.

#### 3.3.1 Example 1: isa2

Example test isa2 is shown in Figure 3–3 (below). It is a distributed variant of the message-passing idiom MP described in 1 Preamble on axiomatic models (on page 9) and 2 A glimpse of cat (on page 16).
Test isa2 is made of threads P0, P1, and P2, which communicate via shared variables x, y, and z, all initialized to 0 (as indicated by the empty preamble of the test).

- P0 has an ordinary relaxed write of x with value 53 at level wi, followed in program order by an atomic release write of y with value 1.
- P1 has an atomic acquire read of y, which places the value read into a register r0 private to P1, then an atomic release write of z with value 1.
- P2 has an atomic acquire read of z, which places the value read into a register r0 private to P2, then an ordinary relaxed read of x, which places the value read into a register r1 private to P2.

Figure 3–4 (below) shows a particular execution candidate of the test isa2. More precisely, an rf relation that satisfies the postlude exists \( (1:r0=1 \land 2:r0=1 \land 2:r1=0) \) is considered. This means that an execution is selected where the read from y by the thread P1 reads the value 1, which is stored to y by the thread P0, i.e., \( (b, c) \in rf \). Similarly, \( (d, e) \in rf \) is selected. Finally, it is considered that the event f reads the initial value of x, i.e., \( (ix, f) \in rf \).

Recall the notion of scope instance, which is derived from the scope tree of a test: two events belong to the same scope instance of level lvl when they come from threads that belong to the level lvl with respect to a scope tree.

For example, in the test isa2, the threads P0 and P1 belong to the same instance of level work-group, and P2 is in its own instance of level work-group. All three threads are in the same instance of level agent. Figure 3–5 (on the next page) shows the scope instances at level work-group for the isa2 test.
3.3.2 Example 2: sb

Example test sb is shown in Figure 3–6 (below).

```
LISA SB
{ }
| P1 |
w[atomic,scar,wg] x 1 | w[atomic,scar,wg] y 1 |
|    |
    |
    |
r[atomic,scar,wg] r0 y | r[atomic,scar,wg] r0 x |
| scopes: (wg (wi P0) (wi P1)) |
exists (0:r0=0 \ 1:r0=0)
```

sb has threads P0 and P1, which communicate via shared variables x and y, both initialized to 0. According to the scope tree, the two threads belong to the same instance of level work–group, but each thread is in its own scope instance of level work–item.

- P0 has an atomic write of x with value 1, tagged wg, followed in program order by an atomic read of y tagged wg.
- P1 writes y then read x, with similar annotations to that of P0. Note that all accesses bear the scar annotations.

The postlude of the test asks if it is possible to have an execution of sb where the two reads take their values from the initial state as shown in Figure 3–7 (on the facing page).

Figure 3–8 (on the facing page) shows another illustration of the notion of scope instance, giving the only scope instance of level work–group for sb.
3.4 Utilities over scopes

As a scoped model, HSA needs a few utilities to manipulate the scopes. For a given scope tag $lvl$, the following are defined:
3. A cat specification of the HSA memory model  3.4 Utilities over scopes

- The **set active-events** (see 3.4.1 The set active-events (below)), which intuitively gathers the events that are active at level lvl1 (that bear a scope tag of level lvl1 or wider)
- The relation **active-instance** (see 3.4.2 The relation active-instance (below)), which restricts the scope instances of level lvl1 (in the sense of tag2scope) to the events that are active at level lvl1 (to the events that belong to the set active-events)

Figure 3-9 (below) shows these definitions and recalls the function union-scope (see 2.4.3 Ruling out the incriminated execution (on page 29)), which returns, for a function f given as argument, the union of the sets (f s) for s ranging over the possible scopes.

3.4.1 The set active-events

The set active-events gathers, given a scope level lvl1, all the events that bear the scope tag lvl1, or a wider scope tag in the sense of the function wider (see Figure 3-2 (on page 38)). For example, at level agent, the function active-events gathers all events with scope tag agent and scope tag system.

**Example on isa2.** For the set active-events ('wg), the event b bears the scope tag wg, the event c bears the scope tag agent, and the events d and e both bear the tag system. Since agent and system are wider than wg, active-events ('wg) = \{b, c, d, e\}. For the set active-events ('agent), the event c bears the tag agent, and the events d and e both bear the tag system, thus active-events ('agent) = \{c, d, e\}.

```
Figure 3-9 Utilities for HSA scopes
"Handling the scope hierarchy"

let rec active-events(lvl) = match lvl with
  | 'system -> tag2events(lvl)
  | _ -> tag2events(lvl) \ union-scope(lvl1))
end

let active-instance(lvl1) =
  let events = active-events(lvl1) in
  tag2scope(lvl1) \\ (events \ events)

let union-scope f = fold (fun (s,y) -> f s | y) (scopes,())
```

**Example on sb.** For the sets active-events ('wi) and active-events ('wg), all events bear the tag wg, which is wider than wi, thus active-events ('wi) = active-events ('wg) = \{a, b, c, d\}.

3.4.2 The relation active-instance

The relation active-instance gathers, given a scope level lvl1, the pairs of events such that:

- The events come from the same scope instance (from threads that belong to the scope lvl1 in the scope tree). This is what the call to tag2scope (lvl1) in Figure 3-9 (above) does.
- The events are both active at level lvl1 (both bear the scope tag lvl1 or wider). This is what the local definition events in Figure 3-9 (above) does.
Example on isa2. Figure 3–10 (below) shows the relations active-instance for the work-group and agent levels in test isa2. Note that the events \(a\) and \(b\) at the left of Figure 3–10 (below) are not related by active-instance (‘wg’), although these two events belong to a common scope instance of level work-group (see Figure 3–5 (on page 40)). This is due to \(a\) being tagged by wi (work-item), a level that is narrower than wg (work-group), not wider. Note also that the events \(d\) and \(e\) at the left of Figure 3–10 (below) are not related by active-instance (‘agent’) because these events belong to a common scope instance at level agent, which happens to comprise all events, and being tagged system, which is wider than agent.

Example on sb. Figure 3–11 (below) shows the relations active-instance for the levels work-item wi and work-group wg.

3.5 Coherence coh

3.5.1 Definition

For a given location \(L\), the coherence order \(\text{co}_L\) is defined as a total order on all memory writes (gathered in the predefined cat set \(W\)) to location \(L\). The single coherent order \(\text{co}\) is the union of all the \(\text{co}_L\) for all locations. This is identical to what was built in 2.5 Building the coherence order (on page 30).

Figure 3–12 (on the next page) gathers the statements relative to the coherence order \(\text{co}\). \(\text{co}0\) is defined as in 2.5 Building the coherence order (on page 30) (the union of \(\text{co-pre}\) and \(\text{co-post}\)). Thus \(\text{co}0\) relates, for each shared variable \(x\), the initial write to \(x\) to the writes of the body relative to \(x\) (co-pre), and the writes of the body relative to \(x\) to the final write to \(x\) (co-post).
3.5 Coherence coh

Figure 3–12 (below) shows where coh is built. As done in 2 A glimpse of cat (on page 16), the from-read relation fr is built as the sequence of the inverse of read-from (rf^−1) and the relation co just picked. Then coh is defined as the transitive closure of communications as defined in 2 A glimpse of cat (on page 16):let coh = (rf|co|fr)+. In other words, coh is the transitive closure (see the use of +) of the union | of read-from rf, coherence co and from-read fr.

```
let makeCohL(s) = linearisations(s,co0)
let same-loc-writes = loc & (W*W)
let allCoL = map makeCohL (classes (same-loc-writes))
let allCo = cross allCoL
with co from allCo
let fr = rf^−1; co
let coh = (rf|co|fr)+
call consistent(coh,po) as CohPoCons
```

3.5.2 Examples on isa2 and sb

On examples isa2 and sb, execution candidates are shown that exhibit a certain choice of coh, and thus of mincohWR (or rf). These execution candidates are in violation of SC (see 2.2.1 Under SC (on page 21)).

Example on isa2. Figure 3–13 (on the facing page) shows an execution candidate of the test isa2, viz., a certain rf (or mincohWR) relation and a certain coh relation.

Figure 3–13 (on the facing page) shows a candidate execution that satisfies the postlude exists (1:r0=1 /\ 2:r0=1 /\ 2:r1=0). Thus a candidate execution is shown where the read from y by the thread P1 reads the value 1, which is stored to y by the thread P0, i.e., (b, c) ∈ rf. Similarly, (d, e) ∈ rf is selected. Finally, it is considered that the event f reads the initial value of x, which is pictured by an rf arrow from the initial write of x to event f (or alternately (f, a) ∈ fr).

Some coh relation is selected that passes the coherence statements of Figure 3–12 (above). For clarity, Figure 3–13 (on the facing page) does not show the complete coh relation but a sub-relation (edges that can be deduced by transitivity are omitted). The significant pairs of this coh relation are (b, c) and (d, e) ∈ coh (which are the same as inter-thread rf pairs), and (f, a) ∈ coh which originates from event f reading the initial value of x.
3.5.3 Consistency of coh and po

The consistency of coh and po can then be checked:

call consistent(coh, po) as CohPoCons

Recall that consistent(a, b) checks that the sequence a;b is irreflexive (a does not go against b).

3.6 Heterogeneous happens-before hhb

This section describes the definitions relative to the hhb relation. Figure 3-15 (on the next page) shows a fragment of the specification where considerations about fences are omitted for brevity. The complete specification closely follows the text of the documentation.

The relation hhb is built as the transitive closure of the union of the program order po and the union of the scoped synchronization order sso at all scopes. Section 3.6.1 Scoped synchronization order (on the next page) describes how sso is built and illustrates it on example isa2. Section 3.6.2 Heterogeneous happens-before (on page 47) describes hhb in more detail.
3.6.1 Scoped synchronization order

Scoped synchronization order formalizes release-acquire synchronization, with scope restrictions.

**Figure 3-15** A treatment of hhb
"Heterogeneous happens-before"

```plaintext
let rel-acq =
(W & Release) * (R & Acquire)) & coh
| ((F & Release) * Acquire) &
| (po & (_ * W)); coh; (po? & (R * _))
| (Release * (F & Acquire)) &
| (po? & (_ * W)); coh; (po & (R * _))

let sso = active-instance(s) & rel-acq
let hhb = (po | union-scopes sso)+
>irreflexive hhb as HhbCons
call consistent (hhb,coh) as HhbCohCons
```

**Definition.** As shown in **Figure 3-2** (on page 38), sets of tagged events are built as follows:

```plaintext
let Release = tag2events('screl) | tag2events('scar)
let Acquire = tag2events('scacq) | tag2events('scar)
let Synchronizing = Acquire | Release
```

Note that the screl, scacq, and scar tags apply to atomic operations only. As a consequence, the above sets regroup atomic operations only.

The scoped synchronization order is defined for a scope tag lvl as the relation rel-acq, defined below, intersected with the active-instance relation (see 3.4 Utilities over scopes (on page 41)). As shown in **Figure 3-13** (on the previous page), the relation rel-acq is built first, which gathers the pairs (e₁, e₂) of accesses such that e₁ is a write release, e₂ a read acquire, and e₁, e₂) ⊆ coh.

To build the scoped synchronization order, the relation rel-acq must be restricted to events that are within the same active scope instance of level lvl and bearing the scope tag lvl or a wider one:

```plaintext
let sso = active-instance(s) & rel-acq
```

**Example on isa2.** **Figure 3-16** (on the facing page) shows the scope synchronization orders for scopes agent and work-group for the test isa2.

The pair (b, c) is in sso-wg because b is a write release (tag screl), c is a read acquire (tag scacq), and (b, c) ⊆ coh (see **Figure 3-10** (on page 43)).

Furthermore, events b and c are in active-instance ('wg) as shown by the pair (b, c) ⊆ active-wg in **Figure 3-10** (on page 43). Indeed they are both in the same scope instance of level work-group, and both bear a scope tag of level wg (this is the case for event b) or wider (c bears agent, which is wider than wg), hence (b, c) ⊆ sso-wg, since sso-wg was defined as the intersection of the relations rel-acq and active-wg.
Figure 3–16 Scope synchronization orders for scopes agent (sso-agent) and work-group (sso-wg)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>W[ordinary, rl, wi] x 53</td>
<td>W[atomic, scrl, wg] y 1</td>
</tr>
<tr>
<td>sso-wg</td>
<td>sso-agent</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>R[atomic, scacq, agent] y 1</td>
<td>W[atomic, scrl, system] z 1</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>R[atomic, scacq, system] z 1</td>
<td>R[ordinary, rl, wi] x 0</td>
</tr>
</tbody>
</table>

Differences between the simple release-acquire in 2.3 Using annotations (on page 24) and the HSA one are shown in Figure 3–17 (below). The figure shows an execution of a test similar to MP-annots with an extra thread (at the left of the figure), which does an atomic release write of \( y \) with value 1 at scope level system.

- In the simpler setup shown in 2.3 Using annotations (on page 24), where rel-acq builds on read-from only, only the pair \( (c, d) \) belongs to rel-acq.
- In the more complete setup of HSA, both pairs \( (c, d) \) and \( (a, d) \) belong to rel-acq, since in this case, rel-acq builds on coh.

Figure 3–17 Differences between the simple rel-acq in 2.3 Using annotations and the HSA one

3.6.2 Heterogeneous happens-before

**Definition.** Following the HSA document, the HSA happens-before order \( hhb \) is defined as the transitive closure of the union of the program order and of the union of scope synchronization orders at all scope levels using the function \( \text{union-scopes} \):

\[
\text{let } \text{union-scopes} f = \text{fold} (\text{fun} \ (s, y) \rightarrow f \ s \ | \ y) \ (\text{scopes}, \{\}) \\
\text{let } hhb = (\text{po} \ | \ \text{union-scopes} \text{ sso}) +
\]

**Validity conditions.** The HSA document defines three validity conditions on \( hhb \) in that the \( hhb \) relation must be:

- Acyclic (equivalently irreflexive, as \( hhb \) is transitive)
- Consistent with \( coh \)
- Consistent with sequentially consistent orders (see 3.7 SC orders (on the next page))

The first two conditions are expressed as follows:
3.7 SC orders

Figure 3–19 (below) shows a specification of SC orders.

A relation is built between synchronizing events that belong to the same active scope instance of level lvl. Then at each level lvl, a relation $R_{lvl}$ is built as the union of hhb and coh intersected with this relation. Then the SC relation at level lvl is the union of $R_{lvl}$ and the relation $R_{lvl'}$ for lvl' narrower than lvl (in the the hierarchy of scopes defined in Figure 3–2 (on page 38)).

This treatment is equivalent to unrolling the forall loop as follows (only the unrolling for the two tags wi and wave are shown):

```
let SWI = makeSCscope('wi,0)
acyclic SWI as ScCons
let SWAVE = makeSCscope('wave,SWI+)
acyclic SWAVE as ScCons
```

Figure 3–19 Specifications of SC orders

"SC orders 2"

```
let sync-instances(lvl) =
  (Synchronizing * Synchronizing) & active-instance(lvl)
let makeSCscope(lvl,lower) =
  (lower|(hhb | coh)) & sync-instances(lvl)

let rec SClower(lvl) = match lvl with
  | 'wi -> makeSCscope('wi,0)
  | _ -> let S' = SClower(narrower(lvl)) in
        makeSCscope(lvl,S')
end
forall lvl in scopes do
  acyclic (SClower(lvl)) as ScCons
end
```
3.7.1 Example on sb

On sb, the SC order at level $wg$ will forbid the non-SC execution shown in Figure 3–6 (on page 40). The test's scope tree states that the two threads of the test are in the same scope instance of level work-group. As a result, all events, which are synchronizing and tagged by the scope tag $wg$, reside in the same sync instance of level work-group. Thus Figure 3–20 (below) shows a contradiction between a tentative work-group SC order SWG and coh.

![Figure 3–20 Contradiction of the work-group SC order and coh](image)

In contrast, the SC orders at level $wi$ will not forbid the non-SC execution of sb, as shown in Figure 3–21 (below). Note that the relation SWI consists of two independent orders, one per scope instance of level work-item. Each of these SWI orders is consistent with and equal to the local $po$ and with the local view of coh (which is the empty relation).

![Figure 3–21 A successful ordering of the two work-item scope instances](image)

3.8 Data races

Finally, the HSA model declares as undefined programs that have races. Race treatment is summarized in Figure 3–22 (on the next page).

A race is a pair of accesses, $e_1$ and $e_2$, that represent a conflict such that neither ($e_1$, $e_2$) nor ($e_2$, $e_1$) are ordered by the happens-before relation defined in 3.6.2 Heterogeneous happens-before (on page 47).
3.8.1 Conflicts

Conflicts are two accesses, \( e_1 \) and \( e_2 \), that conflict if:

- They are relative to the same address (they are in the \( \text{loc} \) relation).
- They belong to different threads (they are in the \( \text{ext} \) relation).
- At least one is a write (gathered in the set \( W \)). Notice that \( \text{at-least-one}(S) \) is implemented as the union of pairs such that one extremity belongs to the set \( S \) and the other can be anything (belongs to \( _{\_} \)).
- None of them is an initialization write (gathered in the set \( IW \)).
- Their scope instances do not match (they do not belong to the same active instance as modeled by the relation \( \text{matches} \)).

The HSA document has two definitions of conflicts: ordinary and special.

**Ordinary conflicts.** Ordinary conflicts are \"Two operations \( X \) and \( Y \) conflict, if they access one or more common byte locations, at least one is a write, and at least one is an ordinary data operation.\" To paraphrase the definition:

\[
\text{let at-least-one}(S) = (S * _{\_}) \mid (_{\_} * S)
\]

\[
\text{let ordinary-conflicts} = \text{loc} \& \text{at-least-one}(W) \& \text{at-least-one}(\text{Ordinary})
\]

The predefined relation \( \text{loc} \) is used that relates accesses to the same location, and the predefined set of events \( W \) (the set of write operations). The set \( \text{Ordinary} \) is the set of ordinary data operations obtained by \( \text{tag2events}(\text{’ordinary}) \).

**Special conflicts.** Special conflicts occur between accesses to the same location, both of them being atomic, such that at least one is a write and whose scope instances do not match.

The relation \( \text{matches} \) is the union of all active instances over all possible scopes. In \( \text{cat} \), this is written as follows (\( \text{Atomic} \) being the set of events bearing the annotation \( \text{’atomic} \)):

\[
\text{let matches} = \text{union-scopes active-instance}
\]
let special-conflicts = (loc & at-least-one(W) & (Atomic * Atomic)) \ matches

The function union-scopes that returns the union of the application of a function on all scope tags is defined in 2.4.3 Ruling out the incriminated execution (on page 29) and 3.6.2 Heterogeneous happens-before (on page 47).

To paraphrase the definition of conflict:

let conflicts = ((ordinary-conflicts|special-conflicts) & ext) \ at-least-one(IW)

Possible omissions in the documentation. The definition of conflicts in the HSA documentation¹ lacks several conditions, which have been added to the definitions above:

- Accesses must be by different threads, which are considered in the definition of conflicts above by the means of the predefined ext relation that relates operations by different threads.
- Initial writes (gathered in the predefined set IW) do not contribute to conflicts.

To see why these conditions are needed, see Figure 3–23 (below), which shows an execution of the test MP+annots (see Figure 3–1 (on page 37)). Without these conditions, the following could occur:

- A conflict of event a with itself.
- Conflicts of events a and d with the init write event ix.

Figure 3–23 Non-existent conflicts and races

3.8.2 Races

Races are conflicts that are not ordered by HSA happens-before in either direction:

let hsa-race = conflicts \ (hhb | hhb^-1)

The postfix \^-1 operator that evaluates to the inverse of relation r is used.

**Possible omissions in the documentation.** The definition of races in the HSA documentation lacks this condition: accesses must be ordered by \( hhb \) one way or the other, which is considered in the definition of races above by the means of \( hhb^{-1} \).

*Figure 3–23 (on the previous page)* shows why this condition is needed. Without the condition on \( hhb^{-1} \), a race occurs from event \( d \) to event \( a \), as those are ordered by \( hhb^{-1} \).

**Signaling undefined executions** can be done with the flag mechanism:

```
flag ~empty races as Undefined
```

This will flag as Undefined any execution where the set of races is not empty.
4. References


4. Jade Alglave, Patrick Cousot, and Luc Maranget. La langue au chat: cat, a language to describe consistency properties. Under submission.


A. Three cat7 library functions

A.1 Definition of fold

Given a function \( f \), a set \( S = \{e_1, e_2, \ldots, e_n\} \) and an element \( y \), the call \( \text{fold} f (S, y) \) returns the value \( f(e_{i_1}, f(e_{i_2}, \ldots, f(e_{i_n}, y))) \), where \( i_1, i_2, \ldots, i_n \) is a permutation of \( 1, 2, \ldots, n \):

\[
\text{let fold } f = \\
\quad \text{let rec fold_rec (es,y) = match es with} \\
\quad \quad |\|\|\{\} -> y \\
\quad \quad |\| e ++ es -> fold_rec (es,f (e,y)) \\
\quad \quad \text{end in} \\
\quad \text{fold_rec}
\]

A.2 Definition of map

Given a function \( f \) and a set \( S = \{e_1, \ldots, e_n\} \), the call \( \text{map} f S \) returns the set \( \{f(e_1), \ldots, f(e_n)\} \). This function can be implemented directly or more concisely by calling the fold function:

\[
\text{let map } f = \text{fun es -> fold (fun (e,y) -> f e ++ y) (es,\{\})}
\]

A.3 Definition of cross

The function \( \text{cross} \) takes a set of sets \( S = \{S_1, S_2, \ldots, S_n\} \) as argument and returns all possible unions built by picking elements from each of the \( S_i \):

\[
\{ e_1 \cup e_2 \cup \cdots \cup e_n \mid e_1 \in S_1, e_2 \in S_2, \ldots, e_n \in S_n \}
\]

Note that if \( S \) is empty, then \( \text{cross} \) should return one relation exactly: the empty relation (the neutral element of the union operator). This choice for \( \text{cross} (\emptyset) \) is natural when \( \text{cross} \) is defined inductively:

\[
\text{cross}(S_1 ++ S) = \bigcup_{e_1 \in S_1, t \in \text{cross}(S)} \{e_1 \cup t\}
\]

In this specification, \( \text{cross}(S_1 ++ S) \) is built by building the set of all unions of one relation \( e_1 \) picked in \( S_1 \) and of one relation \( t \) picked in \( \text{cross}(S) \). From this inductive specification for \( \text{cross} \), the following concise code is written:

\[
\text{let rec cross S = match S with} \\
\quad |\|\|\{\} -> \{ 0 \} \\
\quad |\| S1 ++ S -> \\
\quad \text{let yss = cross S in} \\
\quad \text{fold (fun (el,r) -> map (fun t -> el | t) yss | r) (S1,\{\})}
\]

end
B. Bell and cat files for the HSA model

B.1 Bell file

"Declaring tags, scopes and instructions for HSA"

enum scopes = 'wi || 'wave || 'wg || 'agent || 'system

let narrower(s) = match s with
  || 'system -> 'agent
  || 'agent -> 'wg
  || 'wg -> 'wave
  || 'wave -> 'wi end

let wider(s) = match s with
  || 'agent -> 'system
  || 'wg -> 'agent
  || 'wave -> 'wg
  || 'wi -> 'wave end

definition narrower(s) = match s with
  || 'system -> 'agent
  || 'agent -> 'wg
  || 'wg -> 'wave
  || 'wave -> 'wi end

definition wider(s) = match s with
  || 'agent -> 'system
  || 'wg -> 'agent
  || 'wave -> 'wg
  || 'wi -> 'wave end

definition operation-kind = 'ordinary || 'atomic

definition memory-order = 'rlx || 'scacq || 'screl || 'scar

definition own = 'read-only || 'read-write

instructions;R[{'ordinary'},{'rlx'},{'wi'},{'read-only'}]
instructions;R[{'atomic'},{'rlx'},{'scaq'},{'scar'},{'scopes'},{'read-write'}]
instructions;W[{'ordinary'},{'rlx'},{'wi'},{'read-write'}]
instructions;W[{'atomic'},{'rlx'},{'scaq'},{'scar'},{'scopes'},{'read-write'}]
instructions;RMW[{'atomic'},{'memory-order'},{'scopes'}]
instructions;F[{'scaq'},{'screl'},{'scar'},{'scopes'}]

let Release = tag2events('screl') | tag2events('scar')
let Acquire = tag2events('scaq') | tag2events('scar')
let Synchronizing = Acquire | Release
let Ordinary = tag2events('ordinary')
let Atomic = tag2events('atomic')
let Read-only = tag2events('read-only')
let Read-write = tag2events('read-write')

B.2 Cat file

B.2.1 Utilities: hsa-lib.cat

"Utilities for HSA models"

procedure consistent(a, b) =
  irreflexive a;b
end

procedure includes(a,b) = empty b \ a end

procedure equals(a,b) =
  call includes(a,b)
  call includes(b,a)
end
B. Bell and cat files for the HSA model  B.2 Cat file

B.2. Handling the scope hierarchy: scopes.cat

"Handling the scope hierarchy"

let rec active-events(lvl) = match lvl with
  || 'system -> tag2events(lvl)
  || _ _ -> tag2events(lvl) | active-events(wider(lvl))
end

let active-instance(lvl) =
  let events = active-events(lvl) in
  tag2scope(lvl) & (events * events)

let union-scopes f = fold (fun (s,y) -> f s | y) (scopes,())

B.2.3 Coherence: coh.cat

"Coherence 2"

let makeCohL(s) = linearisations(s,co0)
let same-loc-writes = loc & (W*W)
let allC0L = map makeCohL (classes (same-loc-writes))
let allCo = cross allC0L
with co from allCo

let fr = rf^-1; co
let coh = (rf|co|fr)+

call consistent(coh,po) as CohPoCons

B.2.4 Heterogeneous happens-before: hhb.cat

"Heterogeneous happens-before"

let rel-acq =
  ((W & Release) * (R & Acquire)) & coh
| ((F & Release) * Acquire) &
  ((po & (_ * W)); coh; (po? & (R * _)))
| (Release * (F & Acquire)) &
  ((po? & (_ * W)); coh; (po & (R * _)))

let sso s = active-instance(s) & rel-acq

let hhb = (po | union-scopes sso)+
irreflexive hhb as HhbCons
  call consistent (hhb,coh) as HhbCohCons

B.2.5 SC orders: sc.cat

"SC orders 2"

let sync-instances(lvl) =
  (Synchronizing * Synchronizing) & active-instance(lvl)
let makeSCscope(lvl,lower) =
  (lower | (hhb | coh)) & sync-instances(lvl)

let rec SClower(lvl) = match lvl with
  | 'wi -> makeSCscope('wi,0)
  | _ -> let S' = SClower(narrower(lvl)) in
  2   makeSCscope(lvl,S'+)
end

forall lvl in scopes do
  acyclic (SClower(lvl)) as ScCons
end

B.2.6 Races: hsa-race.cat

"HSA races"

let at-least-one(S) = (S * _ | _ * S)

let ordinary-conflicts =
  loc & at-least-one(W) & at-least-one(Ordinary)

let matches = union-scopes active-instance

let special-conflicts =
  (loc & at-least-one(W) & (Atomic * Atomic)) \ matches

let conflicts =
  ((ordinary-conflicts|special-conflicts) & ext) \ at-least-one(IW)

let hsa-race = conflicts \ (hhb | hhb^-1)

flag ~empty hsa-race as undefined

B.2.7 All together

"HSA"

include  "hsa-lib.cat"
include  "scopes.cat"

(* Coherence *)
include  "coh.cat"

(* Heterogenous happens before *)
include  "hhb.cat"

(* SC orders *)
include  "sc.cat"

(* Races *)
include  "hsa-race.cat"