Global and Local Monitors to Enforce Noninterference in Concurrent Programs

Global and Local Monitors to Enforce Noninterference in Concurrent Programs

Aslan Askarov
Stephen Chong
and
Heiko Mantel

TR-02-15

Computer Science Group
Harvard University
Cambridge, Massachusetts

Global and Local Monitors to Enforce Noninterference in Concurrent Programs

Aslan Askarov
Aarhus University
aslan@cs.au.dk

Stephen Chong
Harvard University
chong@seas.harvard.edu

Heiko Mantel
TU Darmstadt
mantel@cs.tu-darmstadt.de

***Abstract*—Controlling confidential information in concurrent systems is difficult, due to covert channels resulting from interaction between threads. This problem is exacerbated if threads share resources at fine granularity.**

In this work, we propose a novel monitoring framework to enforce strong information security in concurrent programs. Our monitors are hybrid, combining dynamic and static program analysis to enforce security in a sound and rather precise fashion. In our framework, each thread is guarded by its own local monitor, and there is a single global monitor. We instantiate our monitoring framework to support rely-guarantee style reasoning about the use of shared resources, at the granularity of individual memory locations, and then specialize local monitors further to enforce flow-sensitive progress-sensitive information-flow control. Our local monitors exploit rely-guarantee-style reasoning about shared memory to achieve high precision. Soundness of rely-guarantee-style reasoning is guaranteed by all monitors cooperatively. The global monitor is invoked only when threads synchronize, and so does not needlessly restrict concurrency. We prove that our hybrid monitoring approach enforces a knowledge-based progress-sensitive noninterference security condition.

***Keywords*-Language-based security; information-flow control for concurrent systems; hybrid information-flow monitor.**

I. Introduction

Computer systems increasingly exhibit concurrency, including systems that handle confidential information. Providing confidentiality in concurrent systems is challenging, as sharing of resources and synchronization may create *covert channels*, thus facilitating inadvertent leakage of sensitive information. An additional challenge is to enforce confidentiality without unnecessarily limiting or reducing concurrency.

Most previous mechanisms to control the flow of sensitive information (and to prevent information leaks) are inadequate in the presence of modern concurrency features. Operating system abstractions (e.g., [2, 3]) are too coarse grained to control information flow between concurrent threads that share fine-grained resources. Purely static mechanisms (e.g., [4, 5, 6, 7, 8]) are often too restrictive with respect to sharing of resources and synchronization between threads. Purely dynamic mechanisms (such as information-flow control monitors [9, 10, 11, 12]) have not yet been extended to handle fine-grained concurrency, due in part to the difficulties in dynamically and efficiently preventing covert channels resulting from thread interactions.

This is the full version of [1] and provides detailed proofs and additional material such as omitted calculus rules.

We investigate hybrid information-flow control for enforcing confidentiality in concurrent programs. *Hybrid information-flow control monitors* (or, simply, *hybrid monitors*) combine static and dynamic program analysis to secure the flow of information in a system. We present a general framework for monitoring concurrent programs, and instantiate this framework for information-flow security. In our framework, monitoring occurs at two levels: each thread is guarded by a local monitor, and there is a single global monitor to provide control across threads. We enable modular, rely-guarantee-style reasoning about the behavior of multi-threaded programs by extending the concepts of modes and mode states [13] to a dynamic setting. Our global monitor ensures that assumptions made by threads regarding the exclusive or shared use of memory are justified. Our local monitors ensure that individual threads provide the guarantees they promise. Our local monitors additionally control information flow within the guarded threads. The global monitor is not concerned with information-flow control; its sole purpose is to ensure that all assumptions are justified. Crucially, these assumptions can be exploited by the local monitors to establish information-flow control, both effectively and precisely.

Threads' assumptions about memory resources may change only at synchronization points. Thus, the global monitor is accessed only when threads synchronize, and the global monitor does not needlessly restrict concurrency in the program. Existing hybrid information-flow control monitors for concurrent programs (e.g., [14]) use a single monitor shared by all threads, thus causing unnecessary synchronization between threads.

This article's contributions can be summarized as follows.

- We provide efficient and precise information-flow control for concurrent programs using hybrid monitors. Our monitoring approach is novel: each thread has its own local monitor, and there is a single global monitor. Efficiency is achieved because the global monitor is accessed only at thread synchronization points, and so does not needlessly restrict concurrency. We prove that our monitors enforce a progress-sensitive noninterference-like security guarantee.
- The generic framework for monitoring concurrent programs is itself a novel technical contribution. It enables sound rely-guarantee-style reasoning in a dynamic setting, where the global monitor ensures the compatibility of assumptions made with guarantees provided by threads, and the local monitors enforce the guarantees promised by threads. Local

monitors can be specialized to impose further constraints on the guarded thread. By exploiting assumptions, local monitors are able to effectively enforce also global system properties. This is the feature from which we benefit in our hybrid solution for information-flow security.

We consider a simple imperative concurrent calculus, with input and output operators and barrier synchronization. Our local monitors support flow-sensitive security types [15], which facilitates precise reasoning about information flow via program variables. A thread may soundly allow the security level of a variable to change only if the thread has exclusive read-access or exclusive write-access to the variable.

Our local monitors separately track upper bounds on information that may be revealed by the relative order of events (the *timing level* of a thread) and information that may be revealed by learning whether control has reached the current program point or diverged earlier (the *termination level* of a thread). These levels are analogous to the *pc level* traditionally used in security-type systems for non-concurrent programs [4]. The termination level enables us to enforce *progress sensitive* security [16], a generalization of *termination sensitive* security [17] to interactive programs. Most existing work on enforcing information-flow security ignores progress sensitivity due to the complexity and imprecision of the enforcement mechanism. Use of the timing level enables us to prevent *internal timing leaks* [18]. Although the termination level of a thread increases monotonically, we provide additional precision by lowering the timing level of threads at synchronization points, since synchronization between threads restricts the relative order of events before and after the synchronization. Our work leverages the insight that resetting the timing levels of threads at synchronization points is analogous to lowering the pc level at post-dominators of control-flow branches [18, 4, 19].

II. Information Flow in Concurrent Programs

In this section, we provide intuition for how concurrency and progress sensitivity complicate the enforcement of information-flow security, and for how our monitors enforce security in this challenging setting. We then present our general framework for monitoring concurrent programs (Sections III–IV), instantiate it for rely-guarantee reasoning (Section V), and further instantiate it to enforce information-flow security (Sections VI–VII).

Information flow through concurrent access: Confidential information may be inadvertently leaked through thread interaction. For example, consider the following (insecure) program, which consists of two threads that execute concurrently.

Thread 1: input H to foo; output foo $\times$ 2 to H
Thread 2: foo := 42; output foo to L

The first thread inputs confidential information from high-security channel H, stores it in variable foo, and then outputs 2 times foo to channel H. The other thread sets foo to the constant integer 42, then outputs foo to low-security channel L, which we assume can be observed by an adversary. Each of these threads is intuitively secure if it were executed in isolation. However, when executed concurrently, since they both access foo, it is possible that Thread 2 will output confidential information to channel L, violating security by revealing confidential information to the attacker. Indeed, if an observer of channel L sees an output of anything other than 42, she can infer the confidential input.

Fine-grained resource sharing: Following the work of Mantel, Sands, and Sudbrock [13], we support fine-grained reasoning about threads' access to shared resources (like the variable foo in the example above) in order to prevent security violations via thread interaction through such resources. Each thread uses rely-guarantee reasoning about what variables it and other concurrently executing threads might read or write. Consider, for example, the following (secure) program, where Thread 1 assumes that no other threads will read variable w (note that Thread 1 stores confidential information in w), and Thread 2 assumes that no other threads will write variable v (note that Thread 2 sends the content of v to channel L). Provided both assumptions hold, the program is secure, even in the presence of additional threads.

Thread 1: input H to w; output w + v to H
Thread 2: v := 14; output v $\times$ 42 to L

Threads' assumptions and guarantees originate from protocols the concurrently running threads comply with, where coordination among threads is often achieved through synchronization mechanisms. In the following (secure) example, Thread 1 has exclusive access to variable y before the barrier, and Thread 2 has exclusive access after the barrier.

Thread 1: input L to y; barrier
Thread 2: barrier; output y to L

Assumptions and guarantees of threads must be compatible when threads are composed. We assume that a thread's assumptions are stated explicitly. We use the global monitor to impose the corresponding guarantees on all other threads, and use the local monitors of the threads ensure that these guarantees are indeed provided. Thus, our approach supports fine-grained resource sharing while preventing information leakage through concurrent access.

Information flow through nondeterminism: Information can also be leaked when the relative order of observable events (such as outputs to low-security channels) depend on confidential information. In the following (insecure) example, the output on channel L is either 0 followed by 1, or vice versa, where the order likely depends on confidential information.

Thread 1: input h to H; while h > 0 do h := h − 1 od;
output 1 to L
Thread 2: output 0 to L

Our monitors prevent information leakage by this program as follows: Regardless of the value of the confidential input, Thread 1's local monitor notices that the relative ordering between its own output and output of other threads might be influenced by the while loop, and, hence, the local monitor intervenes by blocking the thread's execution before the output occurs. The following program is, however, secure and permitted by our monitors.

Thread 1: input h to H; while h > 0 do h := h − 1 od;
barrier; output 1 to L
Thread 2: barrier; output 0 to L

Even though the relative order of the low outputs is undetermined, the threads cannot perform the output until after the barrier. Intuitively, the program is secure because the synchronization between threads ensures that the order of the low output does not depend on confidential information, even though the low output is nondeterministic [20, 21].

To correctly and precisely track what information might be revealed by the relative timing of events, each local monitor tracks the *timing level*, a security level that is an upper bound on information that has influenced the timing of the thread's execution with respect to other threads. Since after synchronization, the relative timing of threads is independent of information that affected the timing before the synchronization, the timing level of a thread can be lowered immediately after synchronization. This is similar to information-flow control in single-threaded programs, where the *pc level* (a bound on the information that influences whether the current statement is executed) can be lowered at post-dominators of control flow decisions. That is, synchronization points are the post-dominators of concurrent programs, and allow a similar improvement in precision.

Precision of hybrid monitors: Our use of monitors improves the precision of security enforcement, since we consider the security of a single execution, and do not need to determine whether all possible program executions are secure. For example, our monitor permits complete execution of the following single-threaded program if the low input is positive. In contrast, a static analysis would have to classify this program as insecure, because executing it will leak confidential information if the low input is negative or zero.

```
input $L$ to low;
if low $>$ 0 then input $L$ to x else input $H$ to x fi;
output x to $L$
```

Progress sensitivity: If confidential information influences whether a program terminates, observing the termination or non-termination of a program can leak confidential information. This is exacerbated in interactive settings [16] (i.e., where the program produces observable events before the end of the program) and concurrent settings (where many threads may each reveal a small amount of information). For example, in the following (insecure) program, the program silently diverges after outputting a low value that is equal to the secret (i.e., it diverges without producing any further output). Thus, an observer of channel L can learn the secret by noting the last value output.

```
input $H$ to high; low := 0;
while 1 do
  output low to $L$;
  if low<high then low := low+1 else (while 1 do skip od) fi
od
```

An additional concern is that an enforcement mechanism itself might reveal information. In the following (insecure) program, if the secret is positive, a monitor might intervene to prevent the insecure output of high to L by blocking the thread. As a consequence, the output of 42 to channel L would never occur and, hence, an observer of L could infer that the secret must be a positive value, once she realizes that 42 will not be output.

```
input $H$ to high;
if high $>$ 0 then output high to $L$ else skip fi;
output 42 to $L$
```

We prevent information leaks via termination and monitor interventions by tracking the *termination level* and *blocking level* of each thread within our local monitors. These are upper bounds on information that might have influenced, respectively, the termination behavior of loops and the blocking behavior of the monitor. By tracking these levels, we ensure that the termination and blocking behavior does not leak information.

III. Monitored Multi-threaded Computations

We propose a formal model for multi-threaded programs that interact with their environment via channels and whose concurrent threads communicate with each other using shared memory. In our model, we reuse the concepts of modes and of mode states from [13] to facilitate rely-guarantee-style reasoning about the behavior of such programs.

The novelty of our model is that it supports reasoning about multi-threaded programs that are monitored. In our model, programs can be monitored at two levels: local monitors provide control over individual threads while global monitors provide control across threads. By lifting the concept of mode states to threads that are monitored, we enable rely-guarantee-style reasoning both about monitored programs and within monitors.

We recall the concepts of modes and of mode states in Section III-A, and also define a formal notation for mode state changes. In Section III-B, we introduce judgments that capture the behavior of local and global monitors. This provides the basis for lifting the concepts of mode states to multi-threaded programs that are monitored in Section III-C.

Notation: We denote the powerset of a set S by $\mathcal{P}(S)$. We use $A \to B$ and $A \rightharpoonup B$ to denote the set of all total functions and all partial functions, respectively, with domain A and range B. We denote the pre-image of a function $f : A \rightharpoonup B$ by $pre(f)$, i.e., $pre(f) = \{a \in A \mid f(a) \in B\}$. We denote the update of a function f by $f[d \mapsto v]$, where $f[d \mapsto v](d) = v$ and $f[d \mapsto v](d') = f(d')$ for $d' \in pre(f) \setminus \{d\}$.

We refer to partial functions with domain $\mathbb{N}_0$, range A, and a finite pre-image of consecutive numbers starting at 0 as *lists over* A, and denote the set of all such lists by A^*. We use ϵ to denote the empty list, $l{\cdot}a$ to denote the list that results from appending an element $a \in A$ to the end of a list $l \in A^*$, and $l_1{\cdot}l_2$ to denote the concatenation of two lists $l_1, l_2 \in A^*$. The function filter removes from a list over A those elements that do not satisfy a predicate $p \subseteq A$, i.e., $\mathsf{filter}(p, \epsilon) = \epsilon$, $\mathsf{filter}(p, l{\cdot}a) = \mathsf{filter}(p, l){\cdot}a$ if $a \in p$, and $\mathsf{filter}(p, l{\cdot}a) = \mathsf{filter}(p, l)$ if $a \notin p$.

Additional notation and notions are defined in Appendix A.

A. Modes, Mode States, and Annotations

We use *modes* [13] to capture a program developer's expectations about the environment in which a program shall run as well as his intentions. For instance, a program might be designed to use a particular communication protocol with its environment and, hence, a thread running such a program will provide the desired functionality only in an environment that complies with this protocol. Such a convention incorporates both an expectation (namely that the environment will follow the protocol) and an intention (namely that the thread itself is programmed correctly to follow the protocol).

In general, modes can be used to express assumptions and guarantees about arbitrary entities that are relevant for a program's behavior. In this article, we restrict ourselves to modes that express assumptions and guarantees about particular entities, namely program variables. More concretely, we focus on assumptions about which variables might be read and written by a thread's environment and the dual guarantees. We use Var to denote the set of all variables.

We use the symbols A-NR and A-NW to express the assumption that a particular variable will not be read and will not be written, respectively, by the environment of a given thread. Moreover, we use the symbols G-NR and G-NW to express the guarantee that a thread will not read and will not write, respectively, a particular variable. That is, the modes A-NR and G-NR are dual to each other, and so are the modes A-NW and G-NW. We define the function $\mathit{invert} : \mathit{Mod} \to \mathit{Mod}$ to transform a mode $\mathit{mod} \in \mathit{Mod}$ into its dual mode:

$$\begin{array}{ll} \mathit{invert}(\textsf{G-NR}) = \textsf{A-NR} & \mathit{invert}(\textsf{G-NW}) = \textsf{A-NW} \\ \mathit{invert}(\textsf{A-NR}) = \textsf{G-NR} & \mathit{invert}(\textsf{A-NW}) = \textsf{G-NW} \end{array}$$

We refer to the modes A-NR and A-NW as the *no-read* and the *no-write assumption*. Moreover, we refer to the modes G-NR and G-NW as the *no-read* and the *no-write guarantee*. Formally, we define the *set of assumptions* by $\mathit{Asm} = \{\textsf{A-NR}, \textsf{A-NW}\}$, the set of guarantees by $\mathit{Gua} = \{\textsf{G-NR}, \textsf{G-NW}\}$, and the *set of all modes* by $\mathit{Mod} = \mathit{Asm} \cup \mathit{Gua}$.

We use *mode states* [13] to track which assumptions are made and which guarantees are provided. Formally, a mode state is a function $\mathit{mdst} : \mathit{Mod} \to \mathcal{P}(\mathit{Var})$ that returns the set of all variables that are in a given mode. Accordingly, we define the set of all mode states by $\mathit{MdSt} = \mathit{Mod} \to \mathcal{P}(\mathit{Var})$.

We use terms of the form $\textsf{acq}(\mathit{mod}, X)$ and $\textsf{rel}(\mathit{mod}, X)$ to specify that the mode mod is acquired and released, respectively, for all variables in the set $X \subseteq \mathit{Var}$. We call such terms *annotations* and define the set of all annotations by $\mathit{Ann} = \{\textsf{acq}(\mathit{mod}, X), \textsf{rel}(\mathit{mod}, X) \mid \mathit{mod} \in \mathit{Mod}, X \subseteq \mathit{Var}\}$. For singleton sets of variables, we use a shorthand notation and write $\textsf{acq}(\mathit{mod}, x)$ instead of $\textsf{acq}(\mathit{mod}, \{x\})$. Similarly, we write $\textsf{rel}(\mathit{mod}, x)$ instead of $\textsf{rel}(\mathit{mod}, \{x\})$.

We introduce the predicate $\textit{\textbf{Has-Mode-In}} \subseteq \mathit{Ann} \times \mathcal{P}(\mathit{Mod})$ to identify annotations with particular modes. We define this predicate by ann $\textit{\textbf{Has-Mode-In}}$ M iff $\mathit{ann} \in \{\textsf{acq}(\mathit{mod}, X), \textsf{rel}(\mathit{mod}, X) \mid X \subseteq \mathit{Var} \wedge \mathit{mod} \in M\}$ and the projection of a list of annotations γ to a set of modes M by $\gamma \upharpoonright M = \textsf{filter}(\lambda \mathit{ann} \in \mathit{Ann} : \mathit{ann}\ \textit{\textbf{Has-Mode-In}}\ M, \gamma)$.

To model the effects of annotations on mode states, we define the function $\mathit{update} : (\mathit{MdSt} \times \mathit{Ann}) \to \mathit{MdSt}$ by

$$\begin{aligned} &\mathit{update}(\mathit{mdst}, \textsf{acq}(\mathit{mod}, X)) = \\ &\qquad \mathit{mdst}[\mathit{mod} \mapsto (\mathit{mdst}(\mathit{mod}) \cup X)] \\ &\mathit{update}(\mathit{mdst}, \textsf{rel}(\mathit{mod}, X)) = \\ &\qquad \mathit{mdst}[\mathit{mod} \mapsto (\mathit{mdst}(\mathit{mod}) \setminus X)] \end{aligned}$$

Overloading notation, we lift the function $\mathit{update} : (\mathit{MdSt} \times \mathit{Ann}) \to \mathit{MdSt}$ to a function $\mathit{update} : (\mathit{MdSt} \times \mathit{Ann}^\star) \to \mathit{MdSt}$ on lists of annotations by $\mathit{update}(\mathit{mdst}, \epsilon) = \mathit{mdst}$ and $\mathit{update}(\mathit{mdst}, \gamma \cdot \mathit{ann}) = \mathit{update}(\mathit{update}(\mathit{mdst}, \gamma), \mathit{ann})$, where $\mathit{mdst} \in \mathit{MdSt}$, $\mathit{ann} \in \mathit{Ann}$, and $\gamma \in \mathit{Ann}^\star$.

For instance, the annotation $\textsf{acq}(\textsf{A-NW}, x)$ can be used to specify that a thread from now on runs under the assumption that variable x is not written by other threads, $\mathit{mdst}(\textsf{A-NW})$ equals the set of all variables for which the no-write assumption is made in mode state mdst, and, in particular, $x \in (\mathit{update}(\mathit{mdst}, \textsf{acq}(\textsf{A-NW}, x)))(\textsf{A-NW})$ holds.

This fact is explicated by the following proposition.

Proposition 1. *Let $\gamma, \gamma' \in \mathit{Ann}^\star$, $\mathit{mod} \in \mathit{Mod}$, and $\mathit{mdst}, \mathit{mdst}' \in \mathit{MdSt}$ be arbitrary. If $\mathit{mdst}(\mathit{mod}) = \mathit{mdst}'(\mathit{mod})$ and $\gamma \upharpoonright \{\mathit{mod}\} = \gamma' \upharpoonright \{\mathit{mod}\}$ then $(\mathit{update}(\mathit{mdst}, \gamma))(\mathit{mod}) = (\mathit{update}(\mathit{mdst}', \gamma'))(\mathit{mod})$.*

Proof: Let $\mathit{mod} \in \mathit{Mod}$ be arbitrary. We first prove the following implication by induction over the length of γ:

$$\begin{aligned} &\forall \gamma \in \mathit{Ann}^\star : \forall \mathit{mdst}, \mathit{mdst}' \in \mathit{MdSt} : \\ &\quad (\mathit{mdst}(\mathit{mod}) = \mathit{mdst}'(\mathit{mod}) \implies \\ &\qquad ((\mathit{update}(\mathit{mdst}, \gamma))(\mathit{mod}) = \\ &\qquad\quad (\mathit{update}(\mathit{mdst}', \gamma \upharpoonright \{\mathit{mod}\}))(\mathit{mod}))) \end{aligned} \tag{1}$$

Let $\mathit{mdst}, \mathit{mdst}' \in \mathit{MdSt}$ be arbitrary with $\mathit{mdst}(\mathit{mod}) = \mathit{mdst}'(\mathit{mod})$.

Base case: Assume $\gamma = \epsilon$. From $\gamma = \epsilon$, we obtain $\gamma \upharpoonright \{\mathit{mod}\} = \epsilon$. From the definition of update, $\gamma = \epsilon$, $\mathit{mdst}(\mathit{mod}) = \mathit{mdst}'(\mathit{mod})$, and $\gamma \upharpoonright \{\mathit{mod}\} = \epsilon$, we obtain $(\mathit{update}(\mathit{mdst}, \gamma))(\mathit{mod}) = \mathit{mdst}(\mathit{mod}) = \mathit{mdst}'(\mathit{mod}) = (\mathit{update}(\mathit{mdst}', \gamma \upharpoonright \{\mathit{mod}\}))(\mathit{mod})$.

Step case: Assume $\gamma = \gamma^\times \cdot \mathit{ann}$ for some $\gamma^\times \in \mathit{Ann}^\star$ and $\mathit{ann} \in \mathit{Ann}$. Let $\mathit{mdst}^\times = \mathit{update}(\mathit{mdst}, \gamma^\times)$ and $\mathit{mdst}'^\times = \mathit{update}(\mathit{mdst}', \gamma^\times \upharpoonright \{\mathit{mod}\})$. From the induction assumption, we obtain $\mathit{mdst}^\times(\mathit{mod}) = \mathit{mdst}'^\times(\mathit{mod})$.

If $\mathit{ann} \upharpoonright \{\mathit{mod}\} = \epsilon$ then $(\mathit{update}(\mathit{mdst}, \gamma))(\mathit{mod}) = (\mathit{update}(\mathit{mdst}, \gamma^\times))(\mathit{mod})$. From $\mathit{mdst}^\times(\mathit{mod}) = \mathit{mdst}'^\times(\mathit{mod})$ and $\gamma \upharpoonright \{\mathit{mod}\} = (\gamma^\times \cdot \mathit{ann}) \upharpoonright \{\mathit{mod}\} = (\gamma^\times \upharpoonright \{\mathit{mod}\}) \cdot \epsilon = \gamma^\times \upharpoonright \{\mathit{mod}\}$ we obtain $(\mathit{update}(\mathit{mdst}, \gamma))(\mathit{mod}) = (\mathit{update}(\mathit{mdst}, \gamma^\times))(\mathit{mod}) = (\mathit{update}(\mathit{mdst}', \gamma^\times \upharpoonright \{\mathit{mod}\}))(\mathit{mod}) = (\mathit{update}(\mathit{mdst}', \gamma \upharpoonright \{\mathit{mod}\}))(\mathit{mod})$.

If $\mathit{ann} \upharpoonright \{\mathit{mod}\} \neq \epsilon$ then $\mathit{ann} \upharpoonright \{\mathit{mod}\} = \mathit{ann}$. We distinguish two cases:

If $\mathit{ann} = \textsf{acq}(\mathit{mod}, x)$ for some $x \in \mathit{Var}$ then $(\mathit{update}(\mathit{mdst}, \gamma))(\mathit{mod}) = (\mathit{update}(\mathit{mdst}^\times, \mathit{ann}))(\mathit{mod}) = \mathit{mdst}^\times(\mathit{mod}) \cup \{x\} = \mathit{mdst}^{\times'}(\mathit{mod}) \cup \{x\} =$

$(update(mdst^{\times'}, ann))(mod) = (update(mdst', \gamma \restriction \{mod\}))(mod)$.

If $ann = \mathsf{rel}(mod, x)$ for some $x \in Var$ then $(update(mdst, \gamma))(mod) = (update(mdst^{\times}, ann))(mod) = mdst^{\times}(mod) \setminus \{x\} = mdst^{\times'}(mod) \setminus \{x\} = (update(mdst^{\times'}, ann))(mod) = (update(mdst', \gamma \restriction \{mod\}))(mod)$.

This concludes the proof of Proposition 1. It remains to argue that the overall proposition follows.

Let $mdst, mdst' \in MdSt$ and $\gamma, \gamma' \in Ann^{\star}$ be arbitrary with $mdst(mod) = mdst'(mod)$ and $\gamma \restriction \{mod\} = \gamma' \restriction \{mod\}$. From Proposition 1 and $mdst(mod) = mdst'(mod)$, we obtain $(update(mdst, \gamma))(mod) = (update(mdst', \gamma \restriction \{mod\}))(mod)$. From $\gamma \restriction \{mod\} = \gamma' \restriction \{mod\}$, we obtain $(update(mdst', \gamma \restriction \{mod\}))(mod) = (update(mdst', \gamma' \restriction \{mod\}))(mod)$. From Proposition 1 and $mdst'(mod) = mdst'(mod)$, we obtain $(update(mdst', \gamma' \restriction \{mod\}))(mod) = (update(mdst', \gamma'))(mod)$. Thus, $(update(mdst, \gamma))(mod) = (update(mdst', \gamma'))(mod)$ holds. ■

B. Monitors and Events

To control the behavior of individual threads, each thread is guarded by a *local monitor* which is invoked each time the thread performs a computation step. We use *local events* to capture information about computation steps needed by a local monitor. We denote the set of all local events by Ev, use $\alpha \in Ev$ as a meta-variable, use $LMon$ to denote the set of all possible internal states of local monitors, and use $lmon \in LMon$ as a meta-variable for local monitor states.

We employ the judgment $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ to capture that a local monitor in state $lmon \in LMon$ permits the combination of $\alpha \in Ev$ and $\delta \in Ann^{\star}$. The local monitor's internal state is updated to $lmon' \in LMon$.

To control the behavior across threads, a multi-threaded program is guarded by one *global monitor*. We use *global events* to capture the information about such steps that is needed by the global monitor. We denote the set of all global events by GEv, use $\beta \in GEv$ as a meta-variable, use $GMon_n$ to denote the set of all possible internal states of global monitors for pool states with n threads, and identify the initial global monitor states in these sets by $gmon_{init,n} \in GMon_n$. We define the set of all global monitor states by $GMon = \bigcup_{n \in \mathbb{N}_0} GMon_n$ and use $gmon \in GMon$ as a meta-variable.

In this article, we use the global monitor only to control mode-state updates. To simplify our exposition, we assume that threads update their mode state only when synchronizing with other threads, and we consider only one synchronization primitive, namely *global barrier synchronization*. Consequently, the global monitor needs to be invoked only when the program passes a barrier and only needs to distinguish between two global events: sync for barrier synchronization and ϵ for all other steps. Throughout this article, the set of all global events is $GEv = \{\mathsf{sync}, \epsilon\}$. We use a function $\chi : Ev \to GEv$ to extract the corresponding global event from a local event.

When passing a barrier, each alive thread requests changes to its mode state, and the global monitor decides if a given combination of requests by the individual threads is permissible. Moreover, the global monitor may decide to impose mode-state changes on threads that differ from the requested mode-state changes. We employ the judgment $gmon \longrightarrow^{\Gamma,\Delta} gmon'$ to capture that a global monitor in state $gmon \in GMon$ accepts the combination of mode-state-change requests modeled by Γ, imposes the mode-state changes modeled by Δ in response, and updates its internal state to $gmon'$. Formally, Γ and Δ are functions of type $N \to Ann^{\star}$, where $N \subseteq \mathbb{N}_0$ is a finite set. That is, Γ and Δ return a list of annotations for each number in their pre-image. As explained later, whenever we use this judgment, N is the set of identifiers of all threads that are alive when the global monitor is invoked.

We provide a concrete instantiation of $GMon_n$ and $gmon_{init,n}$ together with a calculus for global monitor transitions (i.e., for deriving $gmon \longrightarrow^{\Gamma,\Delta} gmon'$) in Section V. In Section VI, we provide instantiations of Ev and of χ. An instantiation of $LMon$ and a calculus for local monitor transitions are presented in Section VII.

C. Monitored, Multi-threaded Computations

We capture control states of individual threads by terms that we call *commands*, use Com to denote the set of all commands, and express that a thread has terminated by the special command $\mathsf{term} \in Com$.

We capture local states of individual threads by *thread states* and collections of the local states of multiple threads by *pool states*. Formally, a *thread state* is a triple $[com, lmon, mdst]$, where $com \in Com$, $lmon \in LMon$, and $mdst \in MdSt$. Accordingly, we define the set of all thread states by $ThSt = Com \times LMon \times MdSt$. Formally, a *pool state* is a list of thread states. We define the set of all pool states with n threads by $PSt_n = \{0, \ldots, n-1\} \to ThSt$ and define the set of all pool states with arbitrarily many threads by $PSt = \bigcup_{n \in \mathbb{N}_0} PSt_n$. We use $thread \in ThSt$ as meta-variable for thread states and $pool \in PSt$ as a meta-variable for pool states.

From a thread state $thread = [com, lmon, mdst]$, we retrieve its components with selector functions, defined by $thread.\mathsf{com} = com$, $thread.\mathsf{lmon} = lmon$, and $thread.\mathsf{mdst} = mdst$. For instance, $(pool(i)).\mathsf{mdst}$ equals the mode state of the ith thread in the pool state $pool$.

We introduce the predicate $Alive \subseteq ThSt$ to capture that a thread has not yet terminated by $Alive(thread) \equiv thread.\mathsf{com} \neq \mathsf{term}$. To retrieve the identifiers of those threads in a given pool state $pool \in PSt$ that have not yet terminated and that have terminated, respectively, we define the functions $alive : PSt \to \mathcal{P}(\mathbb{N}_0)$ and $terminated : PSt \to \mathcal{P}(\mathbb{N}_0)$ by $alive(pool) = \{i \in pre(pool) \mid Alive(pool(i))\}$ and $terminated(pool) = pre(pool) \setminus alive(pool)$.

Threads communicate with each other via a globally shared memory and with their environment via channels. We model the *state of the shared memory* by a function from variables to values and the history of prior communications on a given channel by a trace. Formally, we use Val to denote the set of all *values*, define the set of states of the shared memory by $Mem = Var \to Val$, and use Ch to denote the set of all *channels*. We model that a value $v \in Val$ is received from channel $ch \in Ch$ and that v is output on ch by the terms $\mathsf{inp}(ch, v)$ and $\mathsf{out}(ch, v)$, respectively. We refer to such terms as *interactions* and denote the set of all interactions by IO. We call lists of interactions *traces*, define the set of all traces by $Tr = IO^\star$, and use $\tau \in Tr$ as a meta-variable for traces.

We capture the behavior of a program's environment by a *communication strategy*. A strategy determines which input the environment supplies to a program on a given channel after a sequence of prior interactions. Formally, a strategy is a function $\sigma : (Tr \times Ch) \to Val$ such that $\sigma(\tau, ch)$ is the value supplied next on channel ch if all prior interactions are captured by the trace τ. We use Σ to denote the set of all such strategies.

Configurations: For capturing snapshots during program execution, we employ three layers of configurations: *global configurations*, *local configurations*, and *command configurations*, where global and local configurations incorporate the state of a global monitor and of a local monitor, respectively.

We use *global configurations* to model global snapshots during a program run. Formally, a global configuration $gcnf$ is a quadruple $\langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle$, where $pool \in PSt$, $mem \in Mem$, $\tau \in Tr$, and $gmon \in GMon$. In the global configuration $gcnf$, the pool state $pool$ captures the local state of each thread of the multi-threaded program, the memory state mem captures the content of all memory locations, the trace τ captures the inputs and outputs that have occurred so far, and $gmon$ captures the internal state of the global monitor.

We use *local configurations* to capture the local view of individual threads during a run. Formally, a local configuration $lcnf$ is a triple $\langle thread, mem, \tau\rangle$ where $thread \in ThSt$, $mem \in Mem$, and $\tau \in Tr$. While the thread state $thread$ models a thread's local state, mem and τ model the content of the memory and the prior communications, respectively. In a global configuration $\langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle$, the local configuration of thread i is $\langle pool(i), mem, \tau\rangle$.

We use *command configurations* to define the local effects of computation steps. Formally, a command configuration $ccnf$ is a triple (com, mem, τ), where $com \in Com$, $mem \in Mem$, and $\tau \in Tr$. While com models a thread's internal state, mem and τ model the memory content and the prior communications, respectively. The command configuration of a local configuration $\langle [com, lmon, mdst], mem, \tau\rangle$ is (com, mem, τ).

We use $GCnf$, $LCnf$, and $CCnf$ to denote the set of all global, local, and command configurations, respectively.

Judgments: For capturing the effects of computation steps at the level of global, local, and command configurations, respectively, we employ the following three judgments:

$$gcnf \twoheadrightarrow_\sigma gcnf' \qquad lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf' \qquad ccnf \overset{\alpha,\gamma}{\rightarrowtriangle}_\sigma ccnf'$$

A strategy $\sigma \in \Sigma$ appears as a subscript of the arrow in all three judgments. It captures the communication strategy of the program's environment. For instance, the first judgment captures that a transition from a global configuration $gcnf$ to a global configuration $gcnf'$ is possible under the strategy σ. The arrow in the second judgment carries three additional annotations: a global event $\beta \in GEv$ and two lists of annotations $\gamma, \delta \in Ann^\star$ that serve different purposes. While γ captures which changes to its mode state a thread desires, δ captures which mode state changes are imposed on the thread in response. The arrow in the third judgment carries as annotations a local event $\alpha \in Ev$ and a list of annotations $\gamma \in Ann^\star$ that captures which changes to its mode state a thread desires.

In the remainder of this section, we provide calculi for the judgments $lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf'$ and $gcnf \twoheadrightarrow_\sigma gcnf'$. An exemplary calculus for the judgment $ccnf \overset{\alpha,\gamma}{\rightarrowtriangle}_\sigma ccnf'$ is provided in Section VI together with an instantiation of Com.

As usual, these calculi induce transition relations. For instance, the calculus for local configurations induces a family of transition relations $(\xrightarrow{\beta,\gamma,\delta}_\sigma)_{\beta\in GEv,\gamma,\delta\in Ann^\star,\sigma\in\Sigma}$, where $\xrightarrow{\beta,\gamma,\delta}_\sigma$ relates two local configurations $lcnf$ and $lcnf'$ iff $lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf'$ is derivable. We use the usual notation for the reflexive, transitive closure of such relations, i.e., for instance, $(\xrightarrow{\beta,\gamma,\delta}_\sigma)^*$ denotes the reflexive, transitive closure of $\xrightarrow{\beta,\gamma,\delta}_\sigma$.

Local Transitions: The judgment $lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf'$ defines which transitions between local configurations are possible. The only rule for deriving instances of this judgment is depicted in Fig. 1.

The judgment for transitions between command configurations in the first premise of the rule in Fig. 1 reflects that a thread's behavior is determined by the command that this thread is executing. In this article, we employ local monitors to guard the behavior of individual threads. Accordingly, the judgment for transitions between local monitor states is used in the second premise to capture that the local monitor must deem the step acceptable. Note that the transition between local monitor states is based on δ and not on γ, i.e., it is based on the mode state changes imposed on the thread, not on the mode state changes requested by the thread. Also note that δ is used to update the mode state in the third premise. The global event β is extracted from the local event α using the function $\chi : Ev \to GEv$ in the last premise of the rule.

This fact is explicated by the following proposition.

Proposition 2. *Let $com, com' \in Com$, $lmon, lmon' \in LMon$, $mdst_1, mdst_1', mdst_2 \in MdSt$, $mem, mem' \in Mem$, $\tau, \tau' \in Tr$, $\beta \in GEv$, $\gamma, \delta \in Ann^\star$, and $\sigma \in \Sigma$ be arbitrary.*

If

$$\langle [com, lmon, mdst_1], mem, \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle [com', lmon', mdst_1'], mem', \tau'\rangle$$

is derivable then there exists a mode state $mdst_2' \in MdSt$ such that

$$\langle [com, lmon, mdst_2], mem, \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle [com', lmon', mdst_2'], mem', \tau'\rangle$$

is derivable.

$$\frac{(com, mem, \tau) \overset{\alpha,\gamma}{\rightarrowtriangle}_{\sigma} (com', mem', \tau') \qquad lmon \xrightarrow{\delta,\alpha}_{perm} lmon' \qquad mdst' = update(mdst, \delta) \qquad \beta = \chi(\alpha)}{\langle [com, lmon, mdst], mem, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle [com', lmon', mdst'], mem', \tau' \rangle}$$

Fig. 1. Transitions between local configurations

$$\frac{i \in alive(pool) \qquad \langle pool(i), mem, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle thread', mem', \tau' \rangle \qquad \beta = \epsilon \qquad \gamma = \delta = \epsilon \qquad gmon' = gmon}{\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle \twoheadrightarrow_{\sigma} \langle\!\langle pool[i \mapsto thread'], mem', \tau', gmon' \rangle\!\rangle}$$

$$\frac{\begin{array}{c} alive(pool) \neq \emptyset \qquad \Gamma, \Delta : alive(pool) \longrightarrow Ann^{\star} \qquad gmon \longrightarrow^{\Gamma,\Delta} gmon' \qquad pool' \in PSt_{|pre(pool)|} \\ \forall j \in terminated(pool) : pool'(j) = pool(j) \qquad \forall i \in alive(pool) : (\langle pool(i), mem, \tau \rangle \xrightarrow{\mathsf{sync},\Gamma(i),\Delta(i)}_{\sigma} \langle pool'(i), mem, \tau \rangle) \end{array}}{\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle \twoheadrightarrow_{\sigma} \langle\!\langle pool', mem, \tau, gmon' \rangle\!\rangle}$$

Fig. 2. Transitions between global configurations

Proof: The mode state before a transition between local configurations can neither prevent the transition nor affect the thread state, memory, or trace after the transition according the rule in Fig. 1. ■

Global Transitions: Transitions between global configurations are captured by the judgment $gcnf \twoheadrightarrow_{\sigma} gcnf'$. To simplify our presentation, we make three restrictions. First, we assume that the process structure is static, i.e., if $\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle \twoheadrightarrow_{\sigma} \langle\!\langle pool', mem', \tau', gmon' \rangle\!\rangle$ then $pre(pool') = pre(pool)$. Second, we assume that threads are scheduled nondeterministically. Third, as stated before, we assume that barrier synchronization is the only synchronization primitive and that threads only request changes to their mode state when they pass a barrier.

The two rules for deriving instances of the judgment $gcnf \twoheadrightarrow_{\sigma} gcnf'$ are depicted in Fig. 2. The first rule captures steps by individual threads, and the second rule captures a barrier synchronization across all threads.

According to the first rule in Fig. 2, an alive thread i (first premise) may be chosen nondeterministically to perform a computation step (second premise). This step must not involve synchronization (third premise), must neither request nor impose mode state changes (fourth premise), and must not affect the global monitor state (fifth premise). Such a step by an individual thread might affect the thread's control state, the state of the local monitor supervising this thread, the shared memory, and the trace. It cannot affect the mode state of this thread (since $\delta = \epsilon$), the local states of other threads (conclusion of the rule), and the global monitor state (since $gmon' = gmon$).

The second rule in Fig. 2 captures synchronization steps. This rule requires that all alive threads jointly pass a barrier (last premise of the rule), which faithfully reflects the intuition of a barrier synchronization. In order to perform a synchronization step, at least one thread must be alive (first premise). The function Γ captures which mode state changes are requested by the individual alive threads (last premise). That is, $\Gamma(i)$ is the list of annotations capturing the mode state changes requested by the ith thread, where $i \in alive(pool)$. The function Δ captures the mode state changes that are imposed on the individual threads (last premise). Which mode state changes are imposed on the individual threads in response to their requests is determined by the global monitor (third premise). Note that the set of threads cannot change during a synchronization step (fourth premise) and that the thread states of terminated threads remain unmodified (fifth premise). Also note that a synchronization step cannot affect the shared memory or the trace (conclusion of the rule). Synchronization steps can only affect the thread state of all alive threads and the state of the global monitor (conclusion of the rule).

Reachability: We say that *a global configuration $gcnf'$ is reachable from a global configuration $gcnf$ under a strategy σ* iff $gcnf\ (\twoheadrightarrow_{\sigma})^{*}\ gcnf'$ holds. We assume that runs of multi-threaded programs start in an initial memory mem_{init} that assigns a dedicated value $v_{init} \in Val$ to all variables (i.e., $\forall x \in Var : mem_{init}(x) = v_{init}$) and with an empty initial trace τ_{init} (i.e., $\tau_{init} = \epsilon$). We say that *a global configuration $gcnf'$ is reachable from a pool state $pool$ under a strategy σ* iff $gcnf'$ is reachable from the global configuration $\langle\!\langle pool, mem_{init}, \tau_{init}, gmon_{init,n} \rangle\!\rangle$ under σ, where $n = |pre(pool)|$. We use $greach_{\sigma}(gcnf)$ and $reach_{\sigma}(pool)$ to denote the set of all global configurations reachable from $gcnf \in GCnf$ and $pool \in PSt$, respectively, under $\sigma \in \Sigma$.

IV. Semantics of Modes

We formally define what it means for a thread to provide a guarantee and what it means for an assumption to be justified. While we define the semantics of G-NR and G-NW in terms of transitions between local configurations, we define the semantics of A-NR and A-NW for a given thread in terms of guarantees given by other threads in a global configuration. Based on the formal semantics of modes, we define conditions that allow one to soundly exploit assumptions when reasoning about the behavior of multi-threaded programs.

Semantics of G-NW and G-NR: We say that *a local configuration $lcnf = \langle thread, mem, \tau \rangle$ provides the no-write guarantee for a variable x* iff the value of x will remain unmodified by each next possible step of the thread. This requirement is captured by the following formula:

$$\begin{aligned}&\forall \sigma \in \Sigma : \forall \beta \in \mathit{GEv} : \forall \gamma, \delta \in \mathit{Ann}^{\star} : \\ &\forall \langle \mathit{thread}', \mathit{mem}', \tau' \rangle \in \mathit{LCnf} : \\ &\langle \mathit{thread}, \mathit{mem}, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle \mathit{thread}', \mathit{mem}', \tau' \rangle \\ &\implies \mathit{mem}'(x) = \mathit{mem}(x)\end{aligned}$$

We say that *a local configuration $lcnf = \langle thread, mem, \tau \rangle$ provides the no-read guarantee for a variable y* iff

$$\begin{aligned}&\forall \sigma \in \Sigma : \forall \beta \in \mathit{GEv} : \forall \gamma, \delta \in \mathit{Ann}^{\star} : \\ &\forall \langle \mathit{thread}', \mathit{mem}', \tau' \rangle \in \mathit{LCnf} : \\ &\langle \mathit{thread}, \mathit{mem}, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle \mathit{thread}', \mathit{mem}', \tau' \rangle \\ &\implies \\ &\forall v \in \mathit{Val} : \langle \mathit{thread}, \mathit{mem}[y \mapsto v], \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle \mathit{thread}', \mathit{mem}', \tau' \rangle \\ &\vee \\ &\forall v \in \mathit{Val} : \langle \mathit{thread}, \mathit{mem}[y \mapsto v], \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \\ &\qquad\qquad \langle \mathit{thread}', \mathit{mem}'[y \mapsto v], \tau' \rangle\end{aligned}$$

That is, a no-read guarantee for a variable y ensures that changing the value of y before a computation step does not alter the effects of this computation step. The two disjuncts on the right-hand side of the implication correspond respectively to the cases where y is overwritten and where y is not overwritten in a computation step.

A local configuration $lcnf = \langle [com, lmon, mdst], mem, \tau \rangle$ provides its guarantees iff it provides the no-write guarantee for each variable $x \in mdst(\mathsf{G\text{-}NW})$ and the no-read guarantee for each $y \in mdst(\mathsf{G\text{-}NR})$. Consequently, if $lcnf$ provides its guarantees then the values of all variables in $mdst(\mathsf{G\text{-}NW})$ will remain unchanged by the thread's next step and the values of all variables in $mdst(\mathsf{G\text{-}NR})$ will not affect the thread's next step. We say that *a thread state $thread$ provides its guarantees* iff, for all $mem \in Mem$ and all $\tau \in Tr$, the local configuration $\langle thread, mem, \tau \rangle$ provides its guarantees. Moreover, we say that *$gcnf = \langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$ provides its guarantees* iff $pool(i)$ provides its guarantees for all $i \in pre(pool)$.

Semantics of A-NW and A-NR: Given a global configuration $gcnf = \langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$, we say that *$gcnf$ justifies the assumption* A-NW *of a thread $i \in pre(pool)$ about a variable x* iff every other alive thread has acquired the mode G-NW for x. Similarly, we say that *$gcnf$ justifies the assumption* A-NR *of a thread $i \in pre(pool)$ about a variable y* iff every other alive thread has acquired the mode G-NR for y.

A global configuration $gcnf = \langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$ justifies its assumptions iff $gcnf$ justifies both the assumption A-NW about each variable in $(pool(i)).\mathsf{mdst}(\mathsf{A\text{-}NW})$ and the assumption A-NR about each variable in $(pool(i)).\mathsf{mdst}(\mathsf{A\text{-}NR})$ for all $i \in pre(pool)$, or formally:

$$\begin{aligned}&\forall i \in pre(pool) : \forall x \in Var : \forall mod \in Asm : \\ &x \in (pool(i)).\mathsf{mdst}(mod) \\ &\implies \forall j \in (alive(pool) \setminus \{i\}) : \\ &\qquad x \in (pool(j)).\mathsf{mdst}(invert(mod))\end{aligned} \tag{2}$$

Note that assumptions of all threads, including terminated threads, must be justified. In contrast, only alive threads need to explicitly provide the dual guarantees for assumptions of other threads. Terminated threads need not acquire the modes G-NW and G-NR because, it is clear that they will not be able to write or read variables in the future.

Sound use of modes: We say that *a global configuration $gcnf$ ensures a sound use of modes* iff, for each strategy $\sigma \in \Sigma$, each reachable global configuration $gcnf' \in greach_{\sigma}(gcnf)$ provides its guarantees and justifies its assumptions.

Moreover, we say that *a pool state $pool$ ensures a sound use of modes* iff $\langle\!\langle pool, mem_{init}, \tau_{init}, gmon_{init,n} \rangle\!\rangle$ ensures a sound use of modes. If a pool state $pool$ ensures a sound use of modes then, at each intermediate state during each possible run, each assumption made is justified by guarantees that are, indeed, provided. Hence, all assumptions made can be exploited soundly when reasoning about possible behaviors.

V. A Monitoring Framework

We propose a framework for monitoring multi-threaded programs based on our model of computation from Section III. Our monitoring framework consists of the definition of a global monitor and of a local monitor. The role of these monitors is complementary. Our global monitor ensures that assumptions made by threads are, indeed, justified. Our local monitor ensures that an individual thread, indeed, provides the guarantees that it promises to provide. The combination of one global monitor and a local monitor at each thread jointly ensure a sound use of modes and, hence, the soundness of modular, rely-guarantee-style reasoning about the behavior of multi-threaded programs.

Our local monitor can be specialized to enforce additional properties. By exploiting assumptions, local monitors can not only establish properties of individual threads, but also global properties of entire multi-threaded programs. We present a specialization of local monitors for information-flow control in Section VII and demonstrate that this specialization soundly enforces end-to-end information-flow security.

Global Monitoring of Multi-threaded Programs: We define a global monitor that grants all acquisitions and releases of assumptions exactly as desired by each thread. In addition, our global monitor ensures that all assumptions of all threads are justified. To justify all assumptions, guarantees might be needed that differ from the guarantees that the individual threads desire to provide. Consequently, our global monitor cannot always grant modifications of guarantees according to these desires.

Our global monitor keeps track of both the assumptions that each alive thread currently makes and the assumptions that each terminated thread had made when it terminated. Formally, *a global monitor state for n threads* is a function that returns a mode state for each thread identifier in $\{0, \ldots, n-1\}$. Accordingly, we instantiate the *set of all global monitor states for n threads* by $GMon_n = \{0, \ldots, n-1\} \longrightarrow MdSt$ and $gmon_{init,n} \in GMon_n$, the *initial global monitor state for n threads*, by $gmon_{init,n}(i) = \{\}$ for all $i \in \{0, \ldots, n-1\}$.

We say that *a global monitor state $gmon \in GMon$* (recall $GMon = \bigcup_{n \in \mathbb{N}_0} GMon_n$) *is compatible with a pool state $pool \in PSt$* iff $pre(gmon) = pre(pool)$ and

$$\frac{\begin{array}{c} \mathit{gmon}' = \mathit{gmon\text{-}update}(\mathit{gmon}, \Gamma) \\ \mathit{pre}(\Gamma) \subseteq \mathit{pre}(\mathit{gmon}) \qquad \Delta = \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma) \end{array}}{\mathit{gmon} \longrightarrow^{\Gamma,\Delta} \mathit{gmon}'}$$

Fig. 3. Transitions between global monitor states

if $(\mathit{gmon}(i))(\mathit{mod}) = (\mathit{pool}(i)).\mathsf{mdst}(\mathit{mod})$ for each $i \in \mathit{pre}(\mathit{pool})$ and $\mathit{mod} \in \mathit{Asm}$.

When threads modify their mode state, the global monitor state is updated accordingly. To capture such updates of the global monitor state, we define the function $\mathit{gmon\text{-}update} : (\mathit{GMon} \times (\mathbb{N}_0 \rightharpoonup \mathit{Ann}^\star)) \longrightarrow \mathit{GMon}$ by

$$\begin{array}{l} \mathit{gmon\text{-}update}(\mathit{gmon}, \Gamma) = \\ \lambda i \in \mathit{pre}(\mathit{gmon}) : \\ \mathit{if}\ i \in \mathit{pre}(\Gamma)\ \mathit{then}\ \mathit{update}(\mathit{gmon}(i), (\Gamma(i) \upharpoonright \mathit{Asm})) \\ \qquad\qquad\quad \mathit{else}\ \mathit{gmon}(i) \end{array}$$

Note that acquisitions and releases of guarantees in $\Gamma(i)$ are ignored when updating the global monitor state. Our global monitor does not keep track of which guarantees threads provide.

Our global monitor uses its internal state to determine which guarantees must be imposed on each individual thread. To determine the list of annotations that our global monitor imposes on the individual threads, we define the function $\mathit{gmon\text{-}impose} : (\mathit{GMon} \times (\mathbb{N}_0 \rightharpoonup \mathit{Ann}^\star)) \longrightarrow (\mathbb{N}_0 \rightharpoonup \mathit{Ann}^\star)$ by

$$\begin{array}{l} \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma) = \\ \quad \mathit{let}\ \mathit{NW} = \lambda i \in \mathit{pre}(\mathit{gmon}') : \\ \qquad\qquad \bigcup\{(\mathit{gmon}'(j))(\mathsf{A\text{-}NW}) \mid j \in \mathit{pre}(\mathit{gmon}') \setminus \{i\}\} \\ \qquad \mathit{NR} = \lambda i \in \mathit{pre}(\mathit{gmon}') : \\ \qquad\qquad \bigcup\{(\mathit{gmon}'(j))(\mathsf{A\text{-}NR}) \mid j \in \mathit{pre}(\mathit{gmon}') \setminus \{i\}\} \\ \quad \mathit{in}\ \lambda i \in \mathit{pre}(\Gamma) : (\Gamma(i) \upharpoonright \mathit{Asm}) \\ \qquad\qquad \cdot \mathit{acq}(\mathsf{G\text{-}NW}, \mathit{NW}(i)) \cdot \mathit{acq}(\mathsf{G\text{-}NR}, \mathit{NR}(i)) \\ \qquad\qquad \cdot \mathit{rel}(\mathsf{G\text{-}NW}, \mathit{pre}(\mathit{gmon}') \setminus \mathit{NW}(i)) \\ \qquad\qquad \cdot \mathit{rel}(\mathsf{G\text{-}NR}, \mathit{pre}(\mathit{gmon}') \setminus \mathit{NR}(i)) \end{array}$$

For each pair (gmon', Γ) with $\mathit{pre}(\Gamma) \subseteq \mathit{pre}(\mathit{gmon}')$, the function $\mathit{gmon\text{-}impose}$ is well defined and returns a function with the same pre-image as Γ.

Fig. 3 presents our global monitor. It is the only rule for deriving instances of the judgment for transitions between global monitor states. In the first premise of this rule, the function $\mathit{gmon\text{-}update}$ is used to update the global monitor state based on the acquisitions and releases of assumptions in Γ. The second premise ensures that the global monitor is aware of all threads that request mode state changes. In the third premise, the function $\mathit{gmon\text{-}impose}$ is used to determine Δ, i.e., the mode state changes to be imposed on all threads that requested mode state changes. Due to the second premise of the rule, $\mathit{gmon\text{-}impose}$ is well defined for the arguments used.

Note that, for each $i \in \mathit{pre}(\mathit{gmon}')$ and each assumption in $\mathit{gmon}'(i)$, the corresponding guarantee is acquired in $(\mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma))(j)$ for all $j \in \mathit{pre}(\Gamma) \setminus \{i\}$. That is, programs do not need to explicitly contain annotations to acquire guarantees, since guarantees will be imposed on threads if needed. This means that no program analysis or human effort is required to determine the guarantees that threads provide. Annotations for assumptions, however, do need to be provided explicitly. There are practical analyses that can infer, e.g., whether a memory location is thread-local (i.e., exclusively accessed by a thread). Such analyses might be suitable building blocks to infer assumption annotations for programs.

Note also that guarantees are acquired in $(\mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma))(i)$ even if the ith thread is providing these guarantees already. Analogously, guarantees are released even if the ith thread is not providing them. Such unnecessary acquisitions and releases of guarantees could be avoided by letting the global monitor keep track of the guarantees that the individual threads provide. We refrain from elaborating this optimization here in more detail.

Local Monitoring of Individual Threads: Our local monitor keeps track of assumptions that a monitored thread makes and of guarantees that a thread provides. In this section, we assume that the state of a local monitor incorporates a mode state, but otherwise leave local monitor states under-specified. We use $\mathit{lmon}.\mathsf{mdst}$ to denote the mode state within $\mathit{lmon} \in \mathit{LMon}$, and we say that *a thread state* $[\mathit{com}, \mathit{lmon}, \mathit{mdst}]$ *is well formed* iff $\mathit{lmon}.\mathsf{mdst} = \mathit{mdst}$ holds.

We say that *a calculus for local monitor transitions properly tracks modes* iff the derivability of $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{\mathit{perm}} \mathit{lmon}'$ implies $\mathit{lmon}'.\mathsf{mdst} = \mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta)$. Moreover, we say that *a calculus for local monitor transitions enforces guarantees* iff it ensures that every well-formed thread state provides its guarantees. We leave the calculus for local monitor transitions unspecified. Such a calculus and a concrete definition of LMon are provided in Section VII.

Sound Use of Modes: We say that *a global configuration* $\mathit{gcnf} = \langle\!\langle \mathit{pool}, \mathit{mem}, \tau, \mathit{gmon} \rangle\!\rangle$ *is well formed* iff gmon is compatible with pool and $\mathit{pool}(i)$ is a well-formed thread state for each $i \in \mathit{pre}(\mathit{pool})$.

The following theorem states that our framework soundly enables rely-guarantee-style reasoning. This result is conditional on two assumptions about the calculus for local monitor transitions, which we discharge in Section VII (see Theorem 2).

Theorem 1. *Let gcnf be a well-formed global configuration that justifies its assumptions. If the calculus for local monitor transitions properly tracks modes and enforces guarantees then gcnf ensures a sound use of modes.*

Proof sketch: Well-formedness is an invariant for global configurations and justifying all assumptions is an invariant for well-formed global configurations if the calculus for local monitor transitions properly tracks modes. From these invariants, we conclude that every global configuration gcnf' that is reachable from gcnf is also well formed and justifies its assumptions by induction on the number of steps from gcnf to gcnf'. From the well-formedness of gcnf' and the assumption that the calculus for local monitor transitions enforces guarantees, we conclude that gcnf' provides its guarantees. Hence, gcnf ensures a sound use of modes. ■

A detailed proof of Theorem 1 appears in Appendix D.

VI. Example Programming Language

As an example language, we use a simple concurrent imperative language that supports multi-threading, communication between threads using shared memory, coordination between threads using barrier synchronization, and interaction between a program and its environment using channels.

Expressions: We use Exp to denote the set of expressions in our language and leave this set under-specified. We assume that the expressions are free of side effects, and use judgment $e, \mathit{mem} \Downarrow v$ to model that $e \in \mathit{Exp}$ evaluates to $v \in \mathit{Val}$ in memory state $\mathit{mem} \in \mathit{Mem}$. Function $\mathit{vars} : \mathit{Exp} \to \mathcal{P}(\mathit{Var})$ retrieves from an expression $e \in \mathit{Exp}$ a set of variables that contains all variables that the value of e might depend on. That is, for all $e \in \mathit{Exp}$ and $\mathit{mem}, \mathit{mem}' \in \mathit{Mem}$, we have

$$\begin{array}{l}(\forall x \in \mathit{vars}(e) : \mathit{mem}(x) = \mathit{mem}'(x)) \\ \implies (e, \mathit{mem} \Downarrow v) \implies (e, \mathit{mem}' \Downarrow v)\end{array}$$

Commands: The *set of commands* Com is defined by:

$$\begin{array}{rcl}\mathit{com} & ::= & x := e \mid \mathsf{skip} \mid \mathit{com}; \mathit{com} \mid \\ & & \mathsf{if}\ e\ \mathsf{then}\ \mathit{com}\ \mathsf{else}\ \mathit{com}\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od} \mid \\ & & \mathsf{input}\ \mathit{ch}\ \mathsf{to}\ x \mid \mathsf{output}\ e\ \mathsf{to}\ \mathit{ch} \mid //\gamma//\ \mathsf{barrier} \mid \\ & & \mathsf{stop} \mid \mathsf{join} \mid \mathsf{more}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od} \mid \mathsf{term}\end{array}$$

where $x \in \mathit{Var}$, $e \in \mathit{Exp}$, $\mathit{ch} \in \mathit{Ch}$, and $\gamma \in \mathit{Ann}^\star$. Terms of the form stop, join, more e do com od, and term capture snapshots of the control state at intermediate computation points, and are not meant to be part of the surface syntax. The sub-language without these terms is the programming language to be used by a programmer.

The behavior of assignments, skip, semicolon, conditionals, and loops is as usual. A command input ch to x reads the next input from the channel ch into the variable x, and a command output e to ch sends the value of the expression e on the channel ch. The command barrier causes a thread to block until all non-terminated threads jointly pass the barrier. Annotations that request a mode state change are placed as a comment in front of barrier commands as, e.g., in $//\epsilon \cdot \mathsf{acq}(\mathsf{A\text{-}NR}, x)//$ barrier; skip. The control state stop models that the execution of a subprogram has completed. The control state join models that the join point of a conditional has been reached. The control state more e do com od models that a loop with the guard e and the body com has been entered and that it will be decided next whether to execute the body or to leave the loop. Finally, the control state term models that the execution of an entire program has terminated.

Local Events: We define the *set of local events* Ev for our example language by the grammar:

$$\begin{array}{rcl}\alpha & ::= & \mathsf{a}(x, e) \mid \mathsf{s} \mid \mathsf{b}(e, \mathit{com}_1, \mathit{com}_2) \mid \mathsf{join} \mid \\ & & \mathsf{enter}(e, \mathit{com}) \mid \mathsf{more}(e, \mathit{com}) \mid \mathsf{leave}(e, \mathit{com}) \mid \\ & & \mathsf{input}(x, \mathit{ch}, v) \mid \mathsf{output}(\mathit{ch}, e, v) \mid \mathsf{sync} \mid \mathsf{term}\end{array}$$

and $\chi : \mathit{Ev} \to \mathit{GEv}$, the abstraction function from local events to global events, as follows:

$$\chi(\alpha) = \begin{cases}\mathsf{sync} & \text{if } \alpha = \mathsf{sync} \\ \epsilon & \text{otherwise}\end{cases}$$

A local event $\mathsf{a}(x, e)$ models that the value of $e \in \mathit{Exp}$ is being assigned to $x \in \mathit{Var}$. The local event s models that a skip command is being executed. A local event $\mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)$ models that a conditional with guard $e \in \mathit{Exp}$ and the branches com_1 and com_2 is being executed, where com_1 and com_2 are the "then" and "else" branches respectively. The local event join models that the join point of a conditional is being passed. The local event $\mathsf{enter}(e, \mathit{com})$ models that a loop while e do com od is being entered. A local event $\mathsf{more}(e, \mathit{com})$ models that the guard $e \in \mathit{Exp}$ of a loop with body com has evaluated to a non-zero value, and the loop body com is being entered. The local event $\mathsf{leave}(e, \mathit{com})$ models that a loop with guard $e \in \mathit{Exp}$ and with body com is being left. A local event $\mathsf{output}(\mathit{ch}, e, v)$ models that a value $v \in \mathit{Val}$ resulting from the evaluation of expression $e \in \mathit{Exp}$ is being output to channel $\mathit{ch} \in \mathit{Ch}$. A local event $\mathsf{input}(x, \mathit{ch}, v)$ models that $v \in \mathit{Val}$ is being received from $\mathit{ch} \in \mathit{Ch}$ and stored in $x \in \mathit{Var}$. The local events sync and term model that a barrier is being passed and that the thread is about to terminate, respectively.

Note that our language for local events closely resembles the syntax of our programming language, though there is no one-to-one correspondence. Note also that some of our local events capture information that goes beyond the actual next computation step. For instance, $\mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)$ provides complete information about both branches of a conditional. That is, this local event captures information about the next computation steps, about computation steps that will occur sometime in the future, and about computation steps that would have occurred if the control flow were resolved differently.

Formal Semantics: Fig. 4 shows selected inference rules for the calculus that defines which transitions on command configurations are possible. The first two rules capture the execution of assignments and skip. The third rule captures the passing of a barrier. The fourth rule captures the choice of a branch in a conditional. It inserts join into the resulting control state to mark the join point. The fifth rule captures the passing of a join point. All rules are shown in Appendix B, including rules for sequential composition, loops, input and output, and termination of programs.

The requested mode state change, i.e., the first label on the arrow, is empty (i.e., $\gamma = \epsilon$) in the conclusions of all rules except for in the rule for barriers and one of the rules for sequential composition. In the rule for barriers, the list of annotations is retrieved from the comment that precedes the barrier command. In the sequential composition rule, the list of annotations is simply propagated from the premise to the conclusion. This reflects our simplifying assumption from Section III-B, that threads request mode state changes only when synchronizing with other threads.

VII. Enforcing Information Flow Security through Local Monitoring

We present our novel hybrid approach to establish information-flow security for multi-threaded programs, building on our monitoring framework from Section V. We specialize our generic local monitor definition to a monitor that tracks

$$\frac{e, mem \Downarrow v}{(x := e, mem, \tau) \overset{\mathsf{a}(x,e),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau)} \qquad \frac{}{(\mathsf{skip}, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)} \qquad \frac{}{(//\gamma//\ \mathsf{barrier}, mem, \tau) \overset{\mathsf{sync},\gamma}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)}$$

$$\frac{e, mem \Downarrow v \qquad (v \neq 0 \implies i = 1) \qquad (v = 0 \implies i = 2)}{(\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}, mem, \tau) \overset{\mathsf{b}(e,com_1,com_2),\epsilon}{\twoheadrightarrow_\sigma} (com_i; \mathsf{join}, mem, \tau)} \qquad \frac{}{(\mathsf{join}, mem, \tau) \overset{\mathsf{join},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)}$$

Fig. 4. Transitions between command configurations: selected rules

and controls information flow. This specialization satisfies the requirements of Section V, modes are properly tracked and guarantees are enforced. Our solution does not require any modification of the global monitor definition from Section V.

We capture information-flow requirements by multi-level security policies and prove the soundness of our approach with respect to a knowledge-based definition of information-flow security *à la* [22]. We are able to establish such an end-to-end security property through thread-local checks by exploiting the assumptions that a thread makes about its environment. The ability to perform rely-guarantee-style reasoning about information-flow security within local monitors of individual threads is a distinctive technical feature of our approach. The practical value of this feature is that it substantially improves precision of local monitoring. Without being able to exploit assumptions, a local monitor would have to conservatively secure the guarded thread for all possible environments, resulting in severe restrictions on the behavior of threads.

The knowledge-based security definition requires that an attacker cannot distinguish a given program run from certain other hypothetical runs. To perform such counter-factual reasoning, information about other possible runs is needed within local monitors. This information is provided by the local events that are emitted during steps of a thread. For instance, the evaluation of the guard of a conditional emits a local event $\mathsf{b}(e, com_1, com_2)$, which provides information about the guard and both branches of the conditional. That is, the approach to information-flow security proposed in this section is a hybrid approach.

A. Information-Flow Security

A *security policy* is a tuple $SP = (Lev, \sqsubseteq, \sqcup, \bot)$ consisting of a set of security levels Lev, a partial order $\sqsubseteq \subseteq Lev \times Lev$, a least-upper-bound operator $\sqcup : (Lev \times Lev) \longrightarrow Lev$, and a least security level $\bot \in Lev$. A *domain assignment* is a function $chlev : Ch \rightarrow Lev$ that associates a security level with each channel. Intuitively, the security level $chlev(ch)$ of a channel ch is the upper bound on the confidentiality of information that the channel's endpoint (e.g., a user or another system) is permitted to learn. Thus, $chlev(ch)$ constitutes an upper bound on the confidentiality of information that might be received from ch and of information that may be sent over ch. When it is clear from context, we conflate channels with their security levels, and write, e.g., $ch \sqsubseteq \ell$ instead of $chlev(ch) \sqsubseteq \ell$.

Attacker Model: We assume that each attacker is associated with a security level, where an attacker at level ℓ can observe all interactions on channels ch with $ch \sqsubseteq \ell$, but cannot observe interactions on other channels. To express what an attacker at level ℓ observes during a program run, we project the trace emitted during the run to the level ℓ. We define *the projection of trace τ to security level ℓ* by

$$\tau{\downarrow}\,\ell = \mathsf{filter}(\tau, \{\mathsf{inp}(ch, v), \mathsf{out}(ch, v) \mid ch \in Ch, chlev(ch) \sqsubseteq \ell, v \in Val\}$$

That is, if τ is the trace produced by some run of a multi-threaded program then an attacker at level ℓ observes $\tau{\downarrow}\,\ell$. Based on his observations, an attacker can try to infer information about the communication strategy used. To capture an upper bound on the attacker's knowledge about which communication strategy might be in use, we define the function $\kappa : (Lev \times PSt \times Tr) \longrightarrow \mathcal{P}(\Sigma)$ by

$$\kappa(\ell, pool, \tau) = \left\{ \sigma \in \Sigma \;\middle|\; \begin{array}{l} \exists \langle\!\langle pool', mem', \tau', gmon' \rangle\!\rangle \in reach_\sigma(pool) : \\ \quad \tau'{\downarrow}\,\ell = \tau{\downarrow}\,\ell \end{array} \right\}$$

The set $\kappa(\ell, pool, \tau)$ contains all strategies that are compatible with the observation $\tau{\downarrow}\,\ell$ and that thus, from the perspective of an attacker at level ℓ, might be in use. The smaller the set $\kappa(\ell, pool, \tau)$, the more accurate the attacker's knowledge. The longer the trace that the attacker observes, the more accurate is the attacker's knowledge, i.e., attacker knowledge is monotonic in the length of the trace the attacker observes.

Note that our definition of κ conservatively allows an attacker to know the program. That is, $\kappa(\ell, pool, \tau)$ is an upper bound on the knowledge of an attacker at level ℓ after this attacker observes trace τ, even if the attacker knows the program that is contained in state $pool$. However, we assume the attacker has no a-priori knowledge about which strategy is used.

Security Property: We regard strategies as confidential information. An attacker at level ℓ should not be able to distinguish two strategies that provide identical inputs at level ℓ and below when all prior interactions at level ℓ and below are identical. We capture classes of strategies that should be indistinguishable by the notion of ℓ-equivalence, defined by

$$\begin{array}{l} \sigma_1 =_\ell \sigma_2 \triangleq \\ \forall ch \in Ch : \forall \tau_1, \tau_2 \in Tr : \\ (ch \sqsubseteq \ell \land \tau_1{\downarrow}\,\ell_1 = \tau_2{\downarrow}\,\ell_2) \implies \sigma_1(\tau_1, ch) = \sigma_2(\tau_2, ch) \end{array}$$

We use a knowledge-based definition of information-flow security, inspired by [22]. The property that we define is progress-sensitive and suitable for our model from Section III.

Definition 1. *We say that* a pool state $pool \in PSt$ is secure for a level $\ell \in Lev$ *iff*

$$\forall \sigma \in \Sigma : \forall \langle\!\langle pool', mem', \tau', gmon' \rangle\!\rangle \in reach_\sigma(pool) :$$

$$\kappa(\ell, pool, \tau') \supseteq \{\sigma' \in \Sigma \mid \sigma =_\ell \sigma'\}$$

Our security property requires that if an attacker at level ℓ observes an execution starting in *pool* under strategy σ, then his knowledge must be bounded by the set of strategies that are ℓ-equivalent to σ. That is, the attacker cannot learn anything about the behavior of the actual strategy on any channel $ch \not\sqsubseteq \ell$.

B. A Specialized Local Monitor

Our local monitor is parametric in the security policy $SP = (Lev, \sqsubseteq, \sqcup, \bot)$ and in the domain assignment $chlev : Ch \to Lev$. A third parameter is a function $\mathcal{L} : Var \to Lev$ that assigns a *default security level* to each variable. These three parameters must be chosen identically for the local monitors of all threads of a multi-threaded program.

Our local monitor maintains a local copy of the mode state of the guarded thread. As a convention, we use $lmdst$ as a meta-variable for such copies of a thread's mode state.

Typing Environment: Information flow into and out of a variable x is constrained by the local monitor based on the default security level $\mathcal{L}(x)$. However, if the guarded thread has exclusive write-access to x and the thread previously wrote information into x that is less confidential than $\mathcal{L}(x)$ then the local monitor can use that. Moreover, if the guarded thread has exclusive read-access to x then the local monitor may allow the thread to temporarily store information in x that is more confidential than $\mathcal{L}(x)$. The local monitor uses a typing environment Γ to track the actual security level of variables for which the guarded thread has exclusive access in some sense. Formally, a typing environment is a partial function $\Gamma : FloatVar \rightharpoonup Lev$, where $FloatVar \subseteq Var$. The variables whose security level may float might be limited, for instance, because the run-time environment accesses some variables while relying that they store information of a particular security level (e.g., variables that define thread priorities, accessed by a priority-based scheduler). The set of variables whose security level must not float is $NonFloatVar = Var \setminus FloatVar$.

We lift a typing environment $\Gamma : FloatVar \rightharpoonup Lev$ to a total function in $Var \to Lev$ by

$$\Gamma\langle x\rangle = \begin{cases} \Gamma(x) & \text{if } x \in pre(\Gamma) \\ \mathcal{L}(x) & \text{otherwise} \end{cases}$$

We write $\Gamma\langle e\rangle$ for $\bigsqcup_{x \in vars(e)} \Gamma\langle x\rangle$.

Mode-State-Says Notation: To improve readability, we introduce a notation for properties of mode states. We write $mdst \triangleright \mathsf{fact}$ (read "$mdst$ says fact") iff mode state $mdst$ has the property expressed by a fact from the following language

$$\mathsf{mayread}(x) \mid \mathsf{maywrite}(x) \mid \mathsf{exclusiveread}(x) \mid \mathsf{exclusivewrite}(x) \mid \mathsf{othersmightread}(x) \mid \mathsf{othersmightwrite}(x)$$

with $x \in Var$. The semantics of $mdst \triangleright \mathsf{fact}$ are defined by:

$$mdst \triangleright \mathsf{mayread}(x) \triangleq x \notin mdst(\mathsf{G\text{-}NR})$$
$$mdst \triangleright \mathsf{maywrite}(x) \triangleq x \notin mdst(\mathsf{G\text{-}NW})$$
$$mdst \triangleright \mathsf{exclusiveread}(x) \triangleq x \in mdst(\mathsf{A\text{-}NR})$$
$$mdst \triangleright \mathsf{exclusivewrite}(x) \triangleq x \in mdst(\mathsf{A\text{-}NW})$$
$$mdst \triangleright \mathsf{othersmightread}(x) \triangleq x \notin mdst(\mathsf{A\text{-}NR})$$
$$mdst \triangleright \mathsf{othersmightwrite}(x) \triangleq x \notin mdst(\mathsf{A\text{-}NW})$$

For brevity, we write $mdst \triangleright [\mathsf{fact}_1, \ldots, \mathsf{fact}_n]$ instead of $mdst \triangleright \mathsf{fact}_1, \ldots, mdst \triangleright \mathsf{fact}_n$, a list of mode-state-says statements concerning the same mode state. We write $mdst \triangleright \mathsf{mayread}(e)$ instead of $mdst \triangleright \mathsf{mayread}(x_1), \ldots, mdst \triangleright \mathsf{mayread}(x_{n'})$, where $vars(e) = \{x_1, \ldots, x_{n'}\}$.

Local Monitor States: A local monitor state is a tuple $\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$. Typing environment $\Gamma : FloatVar \rightharpoonup Lev$ tracks the actual security level of variables to which the guarded thread has exclusive access, as already described. Mode state $lmdst \in MdSt$ is the local monitor's copy of the mode state of the guarded thread. The pc stack $\overline{pc}$ and branch environment stack $\overline{br}$ summarize, respectively, the control flow decisions to reach the current program point, and the behavior of the local monitor on execution paths not taken. The pc stack is a stack of security levels, and the branch environment stack is a stack of tuples, described below. Each time the thread enters a conditional or loop, a security level that bounds the control flow decision is pushed on the pc stack, and a tuple that approximates the monitor's behavior on the branch not taken is pushed on the branch environment stack. When a conditional or loop is exited, the top element of each stack is popped.

To track information flow via internal timing, progress channels, and monitor interventions, local monitor states include timing level $time : Lev$, termination level $term : Lev$, and blocking level $block : Lev$. Termination level $term$ is an upper bound on information that influenced termination of loops prior to this point in the guarded thread's execution. The termination level only increases during thread execution. Blocking level $block$ is an upper bound on information that influenced whether the monitor blocked or allowed thread execution prior to this point in the execution. The blocking level captures information flow via monitor interventions (or lack of interventions), and only increases during execution. Timing level $time$ is an upper bound on information since the last synchronization that influenced *when* the guarded thread reaches its current state. The timing level is lowered after synchronization barriers, but otherwise only increases during execution. The timing level describes the information that may affect the relative timing of this guarded thread with respect to other threads and is used to prevent internal timing leaks.

When a conditional or loop is exited, the timing, termination, and blocking levels are updated to account for information flows due to execution paths that could have been taken, but weren't. For example, when a loop is exited, the termination level is increased to ensure that it is an upper bound of the information that influenced the loop guard expression, which determines how many times the loop is executed.

Calculus for Local Monitor Transitions: Selected inference rules for local monitor transitions are shown in Fig. 5. All inference rules are presented and explained in Appendix C.

Rule (M-Assign1) is used for an assignment $x := e$ when x is readable by other threads, according to the current mode

$$\textsc{M-Assign1}\quad \frac{\begin{array}{c} \mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)] \\ \ell = \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \ell \sqsubseteq \mathcal{L}(x) \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{a}(x,e)}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Assign2}\quad \frac{\mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)] \qquad \ell = \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{a}(x,e)}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Branch}\quad \frac{\mathit{lmdst} \triangleright \mathsf{mayread}(e) \qquad (\ell_{\mathsf{sb}}, \mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ \mathit{com}_1\ \mathsf{else}\ \mathit{com}_2\ \mathsf{fi}, \Gamma, \mathit{lmdst}, \overline{pc}, \mathit{time}, \mathit{term}, \mathit{block})}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)}_{\mathit{perm}} \langle \Gamma, \mathit{lmdst}, \overline{pc}\cdot\ell_{\mathsf{sb}}, \overline{br}\cdot(\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}), \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Join}\quad \frac{\begin{array}{c} \mathit{time}'' = \mathit{time} \sqcup \mathit{time}' \sqcup \ell \qquad \mathit{term}'' = \mathit{term} \sqcup \mathit{term}' \qquad \Gamma'' = \lambda x. \begin{cases} \Gamma(x) \sqcup \Gamma'(x) & \text{if } x \in \mathit{pre}(\Gamma) \cap \mathit{pre}(\Gamma') \\ \Gamma(x) & \text{if } x \in \mathit{pre}(\Gamma) \setminus \mathit{pre}(\Gamma') \\ \mathsf{undef} & \text{otherwise} \end{cases} \\ \mathit{block}'' = \mathit{block} \sqcup \mathit{block}' \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}\cdot\ell, \overline{br}\cdot(\mathit{time}', \mathit{term}', \mathit{block}', \Gamma'), \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{join}}_{\mathit{perm}} \langle \Gamma'', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}'', \mathit{term}'', \mathit{block}''\rangle}$$

$$\textsc{M-Barrier-Local}\quad \frac{\begin{array}{c} (\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} = \bot \qquad \mathit{lmdst}' = \mathit{update}(\mathit{lmdst}, \delta) \\ \mathit{pre}(\Gamma') = \{x \mid x \in \mathit{FloatVar} \wedge (\mathit{lmdst}' \triangleright \mathsf{exclusiveread}(x) \vee \mathit{lmdst}' \triangleright \mathsf{exclusivewrite}(x))\} \\ (\mathit{lmdst} \triangleright \mathsf{exclusiveread}(x) \wedge \mathit{lmdst}' \triangleright \mathsf{othersmightread}(x)) \implies \Gamma(x) \sqsubseteq \mathcal{L}(x) \\ \mathit{lmdst}' \triangleright \mathsf{exclusivewrite}(x) \implies \Gamma'(x) = \Gamma\langle x\rangle \qquad (\mathit{lmdst} \triangleright \mathsf{othersmightwrite}(x) \wedge \mathit{lmdst}' \triangleright \mathsf{exclusiveread}(x)) \implies \Gamma'(x) = \Gamma\langle x\rangle \\ (\mathit{lmdst} \triangleright \mathsf{exclusivewrite}(x) \wedge \mathit{lmdst}' \triangleright [\mathsf{exclusiveread}(x), \mathsf{othersmightwrite}(x)]) \implies \Gamma'(x) = \Gamma(x) \sqcup \mathcal{L}(x) \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\delta, \mathsf{sync}}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}', \overline{pc}, \overline{br}, \bot, \bot, \bot\rangle}$$

$$\Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle(y) = \begin{cases} \mathsf{undef} & \text{if } y \notin \mathit{pre}(\Gamma) \\ \Gamma(y) & \text{if } y \in \mathit{pre}(\Gamma) \wedge y \neq x \\ \ell & \text{if } y \in \mathit{pre}(\Gamma) \wedge y = x \wedge \mathit{lmdst} \triangleright \mathsf{exclusivewrite}(x) \wedge x \in \mathit{FloatVar} \\ \ell \sqcup \mathcal{L}(x) & \text{if } y \in \mathit{pre}(\Gamma) \wedge y = x \wedge \mathit{lmdst} \triangleright \mathsf{othersmightwrite}(x) \wedge x \in \mathit{FloatVar} \end{cases}$$

Fig. 5. Local monitoring: selected rules

state lmdst. Level ℓ bounds the information that might be revealed by evaluating e at this point in the execution: it is influenced by the level of the variables in e, by the decision to execute this command ($\sqcup\overline{pc}$, the join of the pc stack), the fact that the monitor did not previously block the thread (block), the fact that the thread did not previously diverge (term), and the relative timing of this thread with respect to others (time).[1] Since other threads might read x, we require that ℓ is bounded above by $\mathcal{L}(x)$, the default security level of x. Rule (M-Assign2) is similar, but applies when x cannot be read by other threads, which allows us to treat its level flow-sensitively. In both cases Γ' is computed using operator $\Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle$ (defined in Fig. 5) that returns an updated environment with the type of variable x updated to ℓ depending on mode state lmdst. Both rules require x to be writable and all variables in e to be readable according to lmdst.

Rules (M-Branch) and (M-Join) handle conditionals. Recall that monitor event $\mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)$ and join are emitted, respectively, when a thread enters and exits a conditional $\mathsf{if}\ e\ \mathsf{then}\ \mathit{com}_1\ \mathsf{else}\ \mathit{com}_2\ \mathsf{fi}$. Rule (M-Branch) requires that the local monitor's mode state allows the thread to read the conditional expression: $\mathit{lmdst} \triangleright \mathsf{mayread}(e)$. It uses the static bounds oracle function $\mathsf{SB}(\mathit{com}, \Gamma, \mathit{lmdst}, \overline{pc}, \mathit{time}, \mathit{term}, \mathit{block})$ to approximate the behavior of the monitor on both branches. This is an on-the-fly static analysis (thus making the local monitor hybrid) needed for the soundness of information-flow tracking. Security level ℓ_{sb} returned by the oracle is an upper bound on the decision about which branch to take, and the monitor pushes it on pc stack $\overline{pc}$. The other elements returned describe, respectively, upper bounds on the timing level, termination level, blocking level, and typing environment that the local monitor would have after completing the conditional. The analysis in essence considers all possible executions of the conditional, and approximates the behavior of the guarded thread in these hypothetical executions. Level $\mathit{time}_{\mathsf{sb}}$ is an upper bound on information that affects when the conditional finishes, $\mathit{term}_{\mathsf{sb}}$ is an upper bound on information that affects whether execution of the conditional will terminate or diverge, $\mathit{block}_{\mathsf{sb}}$ is an upper bound on the information that affects whether the monitor will block the thread while executing the conditional, and Γ_{sb} describes upper bounds on the typing

[1]Other threads may modify variables in e concurrently with this thread's execution, and thus the relative timing may influence the result of evaluating e. If the guarded thread has exclusive write access to variables in e, then the second premise could be replaced by $\ell = \Gamma\langle e\rangle \sqcup (\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block}$, i.e., timing level time does not need to be included in the join. For simplicity, we do not provide this additional precision.

environment when the conditional terminates. Note that the oracle is a partial function: if the result is undefined the monitor blocks. For example, the result is undefined if a branch contains a barrier command and the branch condition is not $\bot$, since synchronizations are publicly observable and should not depend on confidential information. A full description of the static bounds oracle (including the oracle's semantic interface and an implementation) is available in Appendices E and F.

Rule (M-Join) pops the top elements of the pc stack and the static branching environment stack, and updates the variable context, timing level, termination level, and blocking level to account for potential information flows on the branch not taken.

Rule (M-Barrier-Local) regulates when a thread may synchronize. The first premise ($(\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} = \bot$) ensures that the decision to reach a barrier is influenced only by public information. The second premise computes the updated mode state lmdst'. The remaining premises ensure that the typing environments before and after the barrier (Γ and Γ' respectively) are appropriate based on the access this and other threads may have to variables before and after the barrier.

The other monitor rules, not presented here, are: (M-Skip) (for skip commands); (M-Input1), (M-Input2), and (M-Output) for input and output commands; (M-Enter), (M-More), and (M-Leave) for loops; and (M-Term) for terminated threads.

Theorem 2. *Our calculus for local monitor transitions properly tracks modes and enforces guarantees.*

Proof sketch: Given a derivation of $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{perm} \mathit{lmon}'$, we show $\mathit{lmon}'.\mathsf{mdst} = \mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta)$. This implies that our calculus properly tracks modes. To prove that our calculus enforces guarantees, we show that every local configuration $\langle[\mathit{com}, \mathit{lmon}, \mathit{mdst}], \mathit{mem}, \tau\rangle$ with a well-formed thread state provides the no-write guarantee for each variable $x \in \mathit{mdst}(\mathsf{G\text{-}NW})$ and the no-read guarantee for each variable $y \in \mathit{mdst}(\mathsf{G\text{-}NR})$ by a case distinction on the local event emitted when this local configuration performs a step. ■

A detailed proof of Theorem 2 appears in Appendix D.

From Theorems 1 and 2, we obtain the following corollary.

Corollary 1. *If a global configuration is well formed and justifies its assumptions then it ensures a sound use of modes.*

Soundness of Information-Flow Control: Given a multi-threaded program $\mathit{com}_1 \cdot \ldots \cdot \mathit{com}_n \in \mathit{Com}^*$, we define the initial pool state for this program by

$$\begin{aligned}\mathit{pool}_{\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n} \triangleq\ & [\mathit{com}_1, \mathit{lmon}_{\mathit{init}}, \mathit{mdst}_{\mathit{init}}] \\ & \cdot \ldots \\ & \cdot [\mathit{com}_n, \mathit{lmon}_{\mathit{init}}, \mathit{mdst}_{\mathit{init}}]\end{aligned}$$

where $\mathit{mdst}_{\mathit{init}}$ is the *inital mode state*, defined by

$$\mathit{mdst}_{\mathit{init}}(\mathsf{G\text{-}NR}) = \{\} \quad \mathit{mdst}_{\mathit{init}}(\mathsf{G\text{-}NW}) = \{\}$$
$$\mathit{mdst}_{\mathit{init}}(\mathsf{A\text{-}NR}) = \{\} \quad \mathit{mdst}_{\mathit{init}}(\mathsf{A\text{-}NW}) = \{\}$$

and $\mathit{lmon}_{\mathit{init}}$ is the *initial local monitor state*, defined by

$$\mathit{lmon}_{\mathit{init}} \triangleq \langle \Gamma_{\mathit{init}}, \mathit{mdst}_{\mathit{init}}, \epsilon, \epsilon, \bot, \bot, \bot \rangle$$

where $\Gamma_{\mathit{init}} : \mathit{FloatVar} \rightharpoonup \mathit{Lev}$ is defined by $\mathit{pre}(\Gamma_{\mathit{init}}) = \{\}$.

Theorem 3. *If $\mathit{com}_1 \cdot \ldots \cdot \mathit{com}_n \in \mathit{Com}^*$ is a multi-threaded program such that each command com_i is in the sub-language meant to be used by the programmer (as defined in Section VI) then $\mathit{pool}_{\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n}$ is secure for every level $\ell \in \mathit{Lev}$.*

Proof sketch: The full proof is in Appendix E. Let global configuration $\mathit{gcnf} = \langle\!\langle \mathit{pool}_{\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n}, \mathit{mem}_{\mathit{init}}, \tau_{\mathit{init}}, \mathit{gmon}_{\mathit{init},n} \rangle\!\rangle$. Since gcnf is well-formed and justifies its assumptions, we can soundly use rely-guarantee reasoning based on Definition 1 and Corollary 1. The rest of the proof is lengthy, but uses established proof techniques. It shows that for any security level ℓ, given an execution of gcnf with strategy σ that produces trace τ, and given an ℓ-equivalent strategy σ', there exists an execution of gcnf with σ' that produces trace τ' such that $\tau \downarrow \ell = \tau' \downarrow \ell$. Thus, the knowledge of an attacker at level ℓ that observes trace τ will include both σ and σ'. ■

When our monitoring framework is instantiated with the information-flow control local monitors, it accepts all the secure executions from Section II (with appropriate annotations to indicate assumptions), and correctly rejects the insecure executions (by a local monitor blocking at an appropriate point in the execution). This work presents the first hybrid progress-sensitive information-flow control monitoring framework for concurrent programs that uses fine-grained rely-guarantee reasoning about shared memory. As such, it is the only sound monitoring framework that accepts all secure execution examples from Section II. The examples, though simple, exhibit typical patterns of concurrent programs, and of programs that manipulate information of differing sensitivity.

VIII. Related Work

Static enforcement: Most existing work on information-flow security in concurrent programs uses static techniques. Volpano and Smith [23] provide a type system that enforces probabilistic noninterference in concurrent programs by providing an atomic construct, and preventing high-security loop guards. Russo and Sabelfeld [24] remove the need for the atomic construct under cooperative scheduling. Sabelfeld and Sands [25] provide a type system that ensures probabilistic noninterference for a wide class of schedulers, that also prevents high-security loop guards, and uses Agat's padding technique to prevent timing leaks [26]. Smith [27] presents a type system that reasons precisely about what information influences the timing of executions, prevents timing leaks, and thus enforces probabilistic noninterference for concurrent programs.

Andrews and Reitman [28] present an axiomatic program logic to reason about information flow in sequential and concurrent programs. They use two special "certification variables" in their logic, $\underline{\mathit{local}}$ and $\underline{\mathit{global}}$, which correspond to the pc level and the termination level respectively.

Boudol and Castellani [29] analyze the program and scheduler, and give a type system such that well-typed schedulers and threads satisfy noninterference, which Barthe and Nieto [30] verify. Barthe et al. [31] add mechanisms during compilation to enable a security-aware scheduler to enforce security.

Sabelfeld [8] considers a concurrent language with semaphores, and provides a type system that enforces security. High loops are not allowed, and padding is used to prevent timing leaks, for both branches and fork commands.

Zdancewic and Myers [18] propose that non-determinism should not be observable by low-security users (including non-determinism arising from scheduling, data races, etc.) and present a type system that enforces low-observational determinism for single memory locations. Their enforcement mechanism allows the pc level to be reset at thread synchronization points, similar to our lowering of the timing level at barrier synchronizations. Huisman et al. [32] note that the security condition may permit more information flow than intended and strengthen the condition. Terauchi [33] further improves this condition and enforces it via a fractional-capability type system, which permits updates of a thread's capabilities upon synchronization. This bears similarities to our updates of modes at synchronization points, but fractional capabilities and modes are different. For instance, in our framework a thread may have exclusive write access to a variable without exclusive read access. In contrast to all three articles, our security condition does not demand low-observable determinism.

Mantel and Sudbrock [34] present a security condition that allows nondeterminism in concurrent programs provided secret information does not influence this nondeterminism. They present a type system that enforces security for a broad class of schedulers: once a thread's timing or termination behavior is influenced by secret information, it may not interact with low-security threads. Muller and Chong [21] also permit low-observable nondeterminism via a type system for an extension of the X10 programming language.

Mantel, Sands, and Sudbrock [13] use rely-guarantee reasoning to support flow-sensitive security types in concurrent programs, thus allowing more precise enforcement of security. Our approach is inspired by theirs, but we exploit and justify rely-guarantee reasoning dynamically rather than statically.

Hybrid and dynamic enforcement: By contrast with static enforcement techniques, our approach is hybrid, combining static and dynamic techniques. This enables more precise enforcement of security, since static techniques must accept or reject a program in its entirety, whereas dynamic and hybrid techniques can accept or reject single executions.

Le Guernic [14] presents the first hybrid monitor for concurrent programs. It is flow sensitive, but uses a single monitor (and single type environment) for the entire thread pool, which restricts concurrency. To handle locks, the monitor uses static analysis to determine when a thread might require a lock, and acquires it before any high branch in that thread, thus ensuring lock acquisition does not depend on secret information. Le Guernic's monitor suppresses insecure output instead of blocking the thread. We block threads rather than modify the semantics of programs by altering or suppressing outputs.

Stefan et al. [35] present a dynamic termination-sensitive information-flow control mechanism. Their mechanism does not rely on a single global monitor but rather uses coarse-grained containers with "floating labels," where the label of the container is increased based on information read by the container, and the label restricts writes and other observable effects. In addition to preventing internal timing leaks, they mitigate external timing leaks using predictive mitigation [36]. We do not address external timing leaks, but enforce security at finer granularity (i.e., per program variable) and with greater precision (through flow-sensitivity).

IX. Conclusion

We have developed a novel framework to monitor concurrent programs, and instantiated this framework to enforce a knowledge-based progress-sensitive noninterference security condition in concurrent programs where threads share memory resources at the granularity of individual memory locations.

The framework uses a single global monitor to ensure that threads can soundly use rely-guarantee reasoning about shared memory. Each thread has its own local monitor that both enforces thread guarantees regarding shared memory, and also tracks and controls information flow within the thread. The global monitor is accessed only when threads synchronize, ensuring that the monitoring framework does not needlessly restrict concurrency. The local monitors are hybrid: they combine dynamic techniques for information-flow control with on-the-fly static program analysis to approximate information flow on untaken execution paths. Local monitor precision is improved by using rely-guarantee reasoning.

Acknowledgments

We thank the anonymous reviewers for their comments and Sylvia Grewe for discussions during early stages of this research project. The second author thanks the programming languages group at Harvard University for the inspiring environment during his sabbatical. This work is supported by the German Research Foundation (DFG) under project RSCP (MA 33326/4-2/3) in the priority program RS^3 (SPP 1496) and by the National Science Foundation under Grant No. 1054172.

References

[1] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *Proceedings of the 28th IEEE Computer Security Foundations Symposium*. IEEE Press, Jul. 2015.

[2] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 263–278.

[3] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proceedings of the 21st ACM Symposium on Operating System Principles*, Oct. 2007.

[4] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[5] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Conference Record of the Twenty-Sixth*

Annual ACM Symposium on Principles of Programming Languages, Jan. 1999, pp. 228–241.

[6] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," 2001–2008, software release. Located at http://www.cs.cornell.edu/jif.

[7] V. Simonet, "The Flow Caml System: documentation and user's manual," Institut National de Recherche en Informatique et en Automatique (INRIA), Technical Report 0282, Jul. 2003.

[8] A. Sabelfeld, "The impact of synchronisation on secure information flow in concurrent programs," in *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, vol. 2244, 2002, pp. 225–239.

[9] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, 2009, pp. 352–365.

[10] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, 2009.

[11] ——, "Permissive dynamic information flow analysis," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2010, pp. 3:1–3:12.

[12] ——, "Multiple facets for dynamic information flow," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2012, pp. 165–178.

[13] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium*, 2011, pp. 218–232.

[14] G. Le Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, 2007, pp. 218–232.

[15] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, Jan. 2006, pp. 79–90.

[16] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proceedings of the 13th European Symposium on Research in Computer Security*, Oct. 2008.

[17] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, Jun. 1997, pp. 156–168.

[18] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, Jun. 2003, pp. 29–43.

[19] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[20] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, Jun. 2006, pp. 190–201.

[21] S. Muller and S. Chong, "Towards a practical secure concurrent language," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, Oct. 2012, pp. 57–74.

[22] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Proceedings of the 28th IEEE Symposium on Security and Privacy*, May 2007, pp. 207–221.

[23] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," in *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, 1998, pp. 34–45.

[24] A. Russo and A. Sabelfeld, "Security for multithreaded programs under cooperative scheduling," in *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, vol. 4378, 2006, pp. 474–480.

[25] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Jul. 2000, pp. 200–214.

[26] J. Agat, "Transforming out timing leaks," in *Conference Record of the Twenty-Seventh Annual ACM Symposium on Principles of Programming Languages*, Jan. 2000, pp. 40–53.

[27] G. Smith, "A new type system for secure information flow," in *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Jun. 2001, pp. 115–125.

[28] G. R. Andrews and R. P. Reitman, "An axiomatic approach to information flow in programs," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 56–76, 1980.

[29] G. Boudol and I. Castellani, "Non-interference for concurrent programs and thread systems," *Theoretical Computer Science*, vol. 281, no. 1, pp. 109–130, Jun. 2002.

[30] G. Barthe and L. P. Nieto, "Formally verifying information flow type systems for concurrent and thread systems," in *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, 2004, pp. 13–22.

[31] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld, "Security of multithreaded programs by compilation," *ACM Transactions on Information and System Security*, vol. 13, no. 3, pp. 21:1–21:32, Jul. 2010.

[32] M. Huisman, P. Worah, and K. Sunesen, "A temporal logic characterisation of observational determinism," in *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, 2006.

[33] T. Terauchi, "A type system for observational determinism," in *Proceedings of the 21st IEEE Computer Security

Foundations Symposium, Jun. 2008, pp. 287–300.

[34] H. Mantel and H. Sudbrock, “Flexible scheduler-independent security,” in *Proceedings of the European Symposium on Research in Computer Security*, 2010, pp. 116–133.

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières, “Addressing covert termination and timing channels in concurrent information flow systems,” in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, June 2012.

[36] D. Zhang, A. Askarov, and A. C. Myers, “Predictive mitigation of timing channels in interactive systems,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[37] A. Banerjee, D. A. Naumann, and S. Rosenberg, “Expressive declassification policies and modular static enforcement,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008, pp. 339–353.

Appendix A
Basic Notions and Notation

Sets: We use the usual operators for intersection, union, and subtraction of sets. In addition, we use $\mathcal{P}(S)$ to denote the powerset of a set S and $\overline{S} = D \setminus S$ to denote the complement of a set S in a given domain D.

Functions: We use $A \to B$ and $A \rightharpoonup B$ to denote the set of all total functions and the set of all partial functions, respectively, with domain A and range B. For a function $f : A \rightharpoonup B$, we use $\mathit{pre}(f)$ to denote the pre-image of f, i.e., $\mathit{pre}(f) = \{a \in A \mid f(a) \in B\}$. We use $f[d \mapsto v]$ to denote the update of a function f at position d, i.e., $f[d \mapsto v](d) = v$ and $f[d \mapsto v](d') = f(d')$ for all $d' \in (\mathit{pre}(f) \setminus \{d\})$. Note that a function update might augment the pre-image of a partial function. We use a similar notation to define functions: Given a set R and $r_1, \ldots, r_n \in R$, we use $[d_1 \mapsto r_1] \ldots [d_n \mapsto r_n]$ to denote the total function $f : \{d_1, \ldots, d_n\} \to R$ that maps d_i to r_i for each $i \in \{1, \ldots, n\}$. Moreover, we use λ-calculus to define functions: We write $\lambda x \in A : e$ for the function that takes as argument an element of domain A, binds this value to the variable x, and returns the value of the expression e.

Predicates: A predicate over a set A is a subset of A. In addition to $a \in p$, we use $p(a)$ to denote that a predicate p over A holds at $a \in A$.

Lists: We refer to partial functions with domain $\mathbb{N}_0$, range A, and a finite pre-image of consecutive numbers starting at 0 also as *lists over* A. That is, a partial function $l : \mathbb{N}_0 \rightharpoonup A$ is a *list over* A if $i \in \mathit{pre}(l)$ for all $i < |\mathit{pre}(l)|$. We use A^* to denote the set of all lists over a set A. As a convention, we use ϵ to denote the empty list, $l{\cdot}a$ to denote the list that results from appending an element $a \in A$ to the end of a list $l \in A^*$, $a{\cdot}l$ to denote the list that results from appending $a \in A$ to the beginning of $l \in A^*$, and $l_1{\cdot}l_2$ to denote the concatenation of two lists $l_1, l_2 \in A^*$. That is, $\mathit{pre}(\epsilon) = \emptyset$, $\mathit{pre}(l{\cdot}a) = \mathit{pre}(l) \cup \{|\mathit{pre}(l)|\}$, $l{\cdot}a = l[|\mathit{pre}(l)| \mapsto a]$, $(a{\cdot}l)(0) = a$, $(a{\cdot}l)(i+1) = l(i)$ for all $i \in \mathit{pre}(l)$, $(l_1{\cdot}l_2)(j) = l_1(j)$ for all $j \in \mathit{pre}(l_1)$, and $(l_1{\cdot}l_2)(k + |\mathit{pre}(l_1)|) = l_2(k)$ for all $k \in \mathit{pre}(l_2)$ hold.

We write $a \Subset l$ to denote that $a \in A$ is contained in the list $l \in A^*$. We write $a \notin l$ if a is not contained in list l.

We define the functional map that applies a function $f : A \to B$ to each element of a list over A by $\mathsf{map}(f, \epsilon) = \epsilon$ and $\mathsf{map}(f, l{\cdot}a) = \mathsf{map}(f, l){\cdot}f(a)$. Moreover, we define the functional filter that removes from a list over A those elements that do not satisfy a predicate $p \subseteq A$ by $\mathsf{filter}(p, \epsilon) = \epsilon$, $\mathsf{filter}(p, l{\cdot}a) = \mathsf{filter}(p, l){\cdot}a$ if $p(a)$, and $\mathsf{filter}(p, l{\cdot}a) = \mathsf{filter}(p, l)$ if $\neg p(a)$.

Appendix B
Command Configuration Rules

The following is a complete list of the inference rules for the transitions between command configurations. Selected rules were shown in Figure 4.

We briefly discuss some of the rules that were not explained previously. The rule for loops inserts more e do com od into the control state. There are two rules that capture the step in a control state of the form more e do com od where the body of the loop is executed and where the loop is left, respectively.

There is one rule for input, and one rule for output. Note that input commands accept arbitrary values as input, but which input is supplied by the environment depends on its strategy σ (due to the premise $v = \sigma(\tau, ch)$).

$$\frac{e, mem \Downarrow v}{(x := e, mem, \tau) \overset{\mathsf{a}(x,e),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau)} \qquad \frac{}{(\mathsf{skip}, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)}$$

$$\frac{}{(\mathsf{stop}; com, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow_\sigma} (com, mem, \tau)} \qquad \frac{}{(\mathsf{stop}, mem, \tau) \overset{\mathsf{term},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{term}, mem, \tau)}$$

$$\frac{(com_1, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow_\sigma} (com_1', mem', \tau') \quad com_1 \neq \mathsf{stop}}{(com_1; com_2, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow_\sigma} (com_1'; com_2, mem', \tau')} \qquad \frac{}{(//\gamma//\ \mathsf{barrier}, mem, \tau) \overset{\mathsf{sync},\gamma}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)}$$

$$\frac{e, mem \Downarrow v \qquad (v \neq 0 \implies i = 1) \qquad (v = 0 \implies i = 2)}{(\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}, mem, \tau) \overset{\mathsf{b}(e,com_1,com_2),\epsilon}{\twoheadrightarrow_\sigma} (com_i; \mathsf{join}, mem, \tau)} \qquad \frac{}{(\mathsf{join}, mem, \tau) \overset{\mathsf{join},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)}$$

$$\frac{}{(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \overset{\mathsf{enter}(e,com),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau)}$$

$$\frac{e, mem \Downarrow v \qquad v \neq 0}{(\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \overset{\mathsf{more}(e,com),\epsilon}{\twoheadrightarrow_\sigma} (com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau)}$$

$$\frac{e, mem \Downarrow v \qquad v = 0}{(\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \overset{\mathsf{leave}(e,com),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)} \qquad \frac{v = \sigma(\tau, ch) \qquad mem' = mem[x \mapsto v] \qquad \tau' = \tau \cdot \mathsf{inp}(ch, v)}{(\mathsf{input}\ ch\ \mathsf{to}\ x, mem, \tau) \overset{\mathsf{input}(x,ch,v),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem', \tau')}$$

$$\frac{e, mem \Downarrow v \qquad \tau' = \tau \cdot \mathsf{out}(ch, v)}{(\mathsf{output}\ e\ \mathsf{to}\ ch, mem, \tau) \overset{\mathsf{output}(ch,e,v),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau')}$$

Appendix C
Local Monitor Rules

Figures 6, 7, and 8 contain the complete inference rules for the local monitor transitions. Selected rules were shown and briefly explained in Figure 5. We give more complete descriptions of the rules here.

$$\textsc{M-Skip}\ \frac{}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon,\mathsf{s}}_{\mathit{perm}} \langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Assign1}\ \frac{\begin{array}{c}\mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)] \\ \ell = \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \ell \sqsubseteq \mathcal{L}(x) \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle\end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon,\mathsf{a}(x,e)}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Assign2}\ \frac{\mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)] \qquad \ell = \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon,\mathsf{a}(x,e)}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Input1}\ \frac{\begin{array}{c}\mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \sqsubseteq \mathit{ch} \qquad \mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{othersmightread}(x)] \\ \ell = \mathit{ch} \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \ell \sqsubseteq \mathcal{L}(x) \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle\end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon,\mathsf{input}(x,ch,v)}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Input2}\ \frac{\begin{array}{c}\mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \sqsubseteq \mathit{ch} \qquad \mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{exclusiveread}(x)] \\ \ell = \mathit{ch} \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle\end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon,\mathsf{input}(x,ch,v)}_{\mathit{perm}} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Output}\ \frac{\mathit{lmdst} \triangleright \mathsf{mayread}(e) \qquad \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \sqsubseteq \mathit{ch}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon,\mathsf{output}(ch,e,v)}_{\mathit{perm}} \langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle(y) = \begin{cases} \mathsf{undef} & \text{if } y \notin \mathit{pre}(\Gamma) \\ \Gamma(y) & \text{if } y \in \mathit{pre}(\Gamma) \wedge y \neq x \\ \ell & \text{if } y \in \mathit{pre}(\Gamma) \wedge y = x \wedge \mathit{lmdst} \triangleright \mathsf{exclusivewrite}(x) \wedge x \in \mathit{FloatVar} \\ \ell \sqcup \mathcal{L}(x) & \text{if } y \in \mathit{pre}(\Gamma) \wedge y = x \wedge \mathit{lmdst} \triangleright \mathsf{othersmightwrite}(x) \wedge x \in \mathit{FloatVar} \end{cases}$$

Fig. 6. Local monitoring: skip, assignment, and I/O rules

Figure 6 shows the rules for skip, assignment and input and output. The rules for assignment were described in Section VII-B. The rule for skip, (M-Skip), leaves the monitor state unchanged. The two rules for inputs are analogous to the rules for assignments. Because inputs can always be read, there is no corresponding mode state check.

The only rule for outputs is analogous to (M-Assign1). There is no corresponding rule for (M-Assign2), because channel levels are fixed.

Figure 7 shows the rules for branches and loops.

Rule (M-Branch) requires that the local monitor's mode state allows the thread to read the conditional expression: $\mathit{lmdst} \triangleright \mathsf{mayread}(e)$. It pushes on to the program counter stack an upper bound of the decision about which branch to take (as returned by the static bounds oracle and pushes the other results of $\mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ \mathit{com}_1\ \mathsf{else}\ \mathit{com}_2\ \mathsf{fi}, \Gamma, \mathit{lmdst}, \overline{pc}, \mathit{time}, \mathit{term}, \mathit{block})$ on to the stack of static branch environments. See below for more details and intuition of the computation of static branch environments and the definition of $\mathsf{SB}(\mathit{com}, \Gamma, \mathit{lmdst}, \overline{pc}, \mathit{time}, \mathit{term}, \mathit{block})$.

Once the end of the conditional is reached (i.e., the join event is emitted), rule (M-Join) pops off the top elements of program counter stack and the static branch environment stack. Recall that static branch environment is a tuple $(\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where $\mathit{time}_{\mathsf{sb}}$, $\mathit{term}_{\mathsf{sb}}$, $\mathit{block}_{\mathsf{sb}}$, and Γ_{sb} are upper bounds on, respectively, the timing level, termination level, blocking level, and typing environment of a local monitor for any execution of the conditional. At the end of the conditional, we raise the current timing level, termination level, blocking level, and typing environment to these upper bounds, to ensure that they account for information flow that may have occurred due to the non-execution of the other branch. Thus, the new termination level is the old termination level joined with the termination level in the static branch environment, the new blocking level is the old blocking level joined with the blocking level in the static branch environment, and the typing

M-Branch
$$\frac{\begin{array}{c} \mathit{lmdst} \rhd \mathsf{mayread}(e) \\ (\ell_{\mathsf{sb}}, \mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ \mathit{com}_1\ \mathsf{else}\ \mathit{com}_2\ \mathsf{fi}, \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \mathit{time}, \mathit{term}, \mathit{block})\end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \overline{\mathit{br}}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)}_{\mathit{perm}} \langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}} \cdot \ell_{\mathsf{sb}}, \overline{\mathit{br}} \cdot (\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}), \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

M-Join
$$\frac{\begin{array}{ccc} \mathit{time}'' = \mathit{time} \sqcup \mathit{time}' \sqcup \ell & \mathit{term}'' = \mathit{term} \sqcup \mathit{term}' & \Gamma'' = \lambda x. \begin{cases} \Gamma(x) \sqcup \Gamma'(x) & \text{if } x \in \mathit{pre}(\Gamma) \cap \mathit{pre}(\Gamma') \\ \Gamma(x) & \text{if } x \in \mathit{pre}(\Gamma) \setminus \mathit{pre}(\Gamma') \\ \mathsf{undef} & \text{otherwise} \end{cases} \\ & \mathit{block}'' = \mathit{block} \sqcup \mathit{block}' & \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}} \cdot \ell, \overline{\mathit{br}} \cdot (\mathit{time}', \mathit{term}', \mathit{block}', \Gamma'), \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{join}}_{\mathit{perm}} \langle \Gamma'', \mathit{lmdst}, \overline{\mathit{pc}}, \overline{\mathit{br}}, \mathit{time}'', \mathit{term}'', \mathit{block}''\rangle}$$

M-Enter
$$\frac{\mathit{lmdst} \rhd \mathsf{mayread}(e) \qquad (\ell_{\mathsf{sb}}, \mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od}, \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \mathit{time}, \mathit{term}, \mathit{block})}{\langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \overline{\mathit{br}}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{enter}(e, \mathit{com})}_{\mathit{perm}} \langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}} \cdot \ell_{\mathsf{sb}}, \overline{\mathit{br}} \cdot (\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}), \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\text{M-More}\ \frac{\mathit{lmdst} \rhd \mathsf{mayread}(e)}{\langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \overline{\mathit{br}}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{more}(e, \mathit{com})}_{\mathit{perm}} \langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \overline{\mathit{br}}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

M-Leave
$$\frac{\begin{array}{ccc} \mathit{lmdst} \rhd \mathsf{mayread}(e) & \mathit{time}'' = \mathit{time} \sqcup \mathit{time}' \sqcup \ell & \Gamma'' = \lambda x. \begin{cases} \Gamma(x) \sqcup \Gamma'(x) & \text{if } x \in \mathit{pre}(\Gamma) \cap \mathit{pre}(\Gamma') \\ \Gamma(x) & \text{if } x \in \mathit{pre}(\Gamma) \setminus \mathit{pre}(\Gamma') \\ \mathsf{undef} & \text{otherwise} \end{cases} \\ \mathit{term}'' = \mathit{term} \sqcup \mathit{term}' & \mathit{block}'' = \mathit{block} \sqcup \mathit{block}' & \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{\mathit{pc}} \cdot \ell, \overline{\mathit{br}} \cdot (\mathit{time}', \mathit{term}', \mathit{block}', \Gamma'), \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow^{\epsilon, \mathsf{leave}(e, \mathit{com})}_{\mathit{perm}} \langle \Gamma'', \mathit{lmdst}, \overline{\mathit{pc}}, \overline{\mathit{br}}, \mathit{time}'', \mathit{term}'', \mathit{block}''\rangle}$$

Fig. 7. Local monitoring: branches and loops

environment raises the level of any variable that is both in the domain of the current typing environment and the typing environment of the static branch environment. Note that the new timing level is the join of the old timing level, the timing level of the static branch environment, and ℓ (i.e., an upper bound on the information that influences the value of the branch condition). Although it can be shown that $\ell \sqsubseteq \mathit{time}_{\mathsf{sb}}$, we fold in ℓ explicitly for clarity.

Note that in (M-Join), the domain of the new typing environment Γ'' is (and must be) the same as the domain of the original typing environment Γ. Consider a conditional expression with a low branch, and only one of the two branches has a barrier in it. Since the domain of the typing environment may change at barriers, the domain of Γ may differ from the domain of the static branch environment at the top of the stack (Γ' in the (M-Join) rule). The definition of Γ'' in (M-Join) is careful to ensure that Γ'' has the same domain as Γ, regardless of the domain of Γ'.

Recall that when a thread enters a loop while e do com od it emits an $\mathsf{enter}(e, \mathit{com})$ event, and the loop becomes more e do com od. Each time the thread enters the loop body (i.e., given more e do com od, expression e evaluates to a non-zero value, and more e do com od will step to com; more e do com od) it emits a $\mathsf{more}(e, \mathit{com})$ event. When the loop terminates (i.e., given more e do com od, expression e evaluates to zero, and more e do com od will step to stop) the thread emits a $\mathsf{leave}(e, \mathit{com})$ event.

Upon receiving an $\mathsf{enter}(e, \mathit{com})$ event, the thread monitor will either block (if execution of the loop may be insecure) or Rule (M-Enter) will push onto the program counter stack an upper bound of the decision about which branch to take (as returned by the static bounds oracle) and pushes the other results of $\mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od}, \Gamma, \mathit{lmdst}, \overline{\mathit{pc}}, \mathit{time}, \mathit{term}, \mathit{block})$ on to the stack of static branch environments. See below for more details and intuition of the computation of static bounds. In addition, the rule requires that the thread's mode state permits it to read expression e.

Upon receiving a $\mathsf{more}(e, \mathit{com})$ event, we simply check that the thread is still permitted to read expression e. We do not modify the stack of program counter levels, the timing level, termination level, blocking level, or stack of static branch environments. Intuitively, this is because the result returned by the static oracle already conservatively approximated an upper bound for the information that may be learned by evaluating the loop guard e in any iteration of the loop (and thus the top element of the program counter stack does not need to be modified) and also produced a static branch environment that conservatively approximates the effects on the timing, termination, blocking and typing context from zero or more iterations of the loop. Note that the timing level, termination level, blocking level and typing context from the static branching environment for the loop will be incorporated into the local monitor state when the loop exits.

$$\textsc{M-Term}\ \frac{\overline{pc} = \epsilon \qquad \overline{br} = \epsilon \qquad term = \bot \qquad block = \bot}{\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle \longrightarrow^{\epsilon,\mathsf{term}} \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle}$$

$$\textsc{M-Barrier-Local}\ \frac{\begin{array}{c}(\sqcup\overline{pc}) \sqcup term \sqcup block = \bot \qquad lmdst' = update(lmdst, \delta)\\ pre(\Gamma') = \{x \mid x \in FloatVar \wedge (lmdst' \triangleright \mathsf{exclusiveread}(x) \vee lmdst' \triangleright \mathsf{exclusivewrite}(x))\}\\ (lmdst \triangleright \mathsf{exclusiveread}(x) \wedge lmdst' \triangleright \mathsf{othersmightread}(x)) \implies \Gamma(x) \sqsubseteq \mathcal{L}(x)\\ lmdst' \triangleright \mathsf{exclusivewrite}(x) \implies \Gamma'(x) = \Gamma\langle x\rangle\\ (lmdst \triangleright \mathsf{othersmightwrite}(x) \wedge lmdst' \triangleright \mathsf{exclusiveread}(x)) \implies \Gamma'(x) = \Gamma\langle x\rangle\\ (lmdst \triangleright \mathsf{exclusivewrite}(x) \wedge lmdst' \triangleright [\mathsf{exclusiveread}(x), \mathsf{othersmightwrite}(x)]) \implies \Gamma'(x) = \Gamma(x) \sqcup \mathcal{L}(x)\end{array}}{\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle \longrightarrow^{\delta,\mathsf{sync}}_{perm} \langle \Gamma', lmdst', \overline{pc}, \overline{br}, \bot, \bot, \bot\rangle}$$

Fig. 8. Local monitoring: Termination and Local Barrier monitoring rules

Figure 8 shows the rules for termination and barrier synchronization.

Rule (M-Term) restricts when a thread may observably terminate. Recall that a thread issues a term event when the thread changes from stop to term (see Appendix B), and that the thread transition rules ensure that term appears only at the top level (i.e., not as a subcommand). Unlike a blocked thread, a terminated thread does not prevent barrier synchronization from occurring, and thus there is a difference between a terminated thread and a blocked thread. So, (M-Term) requires that a thread may terminate only when the termination level $term$ and blocking level $block$ are $\bot$. Otherwise, the fact that the thread is terminated rather than blocked may reveal confidential information. (This is the same reason that (M-Barrier-Local) requires $(\sqcup\overline{pc}) \sqcup term \sqcup block = \bot$.) Note also that since the term should only appear at the top level, we require that the program counter stack $\overline{pc}$ and the stack of static branching environments $\overline{br}$ are empty.

Rule (M-Barrier-Local) regulates when a local thread may synchronize. The first premise ($(\sqcup\overline{pc}) \sqcup term \sqcup block = \bot$) ensures that the decision to reach a barrier is influenced only by public information. The remaining premises ensure that the local typing environments before and after the barrier (Γ and Γ' respectively) are appropriate based on the accesses this and other threads may have to variables before and after the barrier synchronization.

The second premise of (M-Barrier-Local) computes the updated mode state $lmdst'$. The third premise ($pre(\Gamma') = \{x \mid x \in FloatVar \wedge (lmdst' \triangleright \mathsf{exclusiveread}(x) \vee lmdst' \triangleright \mathsf{exclusivewrite}(x))\}$) ensures that the domain of the new local typing environment is exactly the variables for the security level is allowed to float ($x \in FloatVar$) and the thread has either exclusive read or exclusive write access (since if the thread does not have either exclusive read or exclusive write access, then the security level of the variable is fixed).

The fourth premise ($(lmdst \triangleright \mathsf{exclusiveread}(x) \wedge lmdst' \triangleright \mathsf{othersmightread}(x)) \implies \Gamma(x) \sqsubseteq \mathcal{L}(x)$) requires that if this thread is releasing exclusive read access, the security level of information of x before the barrier $\Gamma(x)$ is less than or equal to the default security level for the variable $\mathcal{L}(x)$.

The fifth premise ($lmdst' \triangleright \mathsf{exclusivewrite}(x) \implies \Gamma'(x) = \Gamma\langle x\rangle$) requires that if the thread has exclusive write access to x after the barrier, then the local typing environment must map x to the security level of x before the barrier.

The sixth premise ($(lmdst \triangleright \mathsf{othersmightwrite}(x) \wedge lmdst' \triangleright \mathsf{exclusiveread}(x)) \implies \Gamma'(x) = \Gamma\langle x\rangle$) similarly requires that the local typing environment after the barrier must map x to the security level of x before the barrier if this thread did not have exclusive write access before the barrier, and has exclusive read access after the barrier.

The seventh and final premise ($(lmdst \triangleright \mathsf{exclusivewrite}(x) \wedge lmdst' \triangleright [\mathsf{exclusiveread}(x), \mathsf{othersmightwrite}(x)]) \implies \Gamma'(x) = \Gamma(x) \sqcup \mathcal{L}(x)$) requires that if this thread is giving up exclusive write access to x and has exclusive read access after the barrier then the local typing environment after the barrier must map x to the join of the security level of x before the barrier and the default security level of x.

A. Computation of static bounds

When execution of a command branches (either by an if statement or entering a while loop) the monitor statically analyzes the command, in order to approximate what the monitor would do on any possible execution of the command. The result of this analysis is used by the monitor in two ways: (1) to add a security level to the program counter stack $\overline{pc}$ that captures an upper bound on the decision of which branch to execute, or whether to execute the loop body; and (2) upon exiting the branch or loop to increase the timing level, termination level, and typing context to account for the possible behavior of the monitor on other executions of the command. In Appendices E and F we present further details, including a semantic specification of the analysis, and a static analysis that meets this specification.

Appendix D
Soundness of Rely-Guarantee Style Reasoning

In this section, we provide the detailed proof of Theorem 1 and of Theorem 2.

The following proposition clarifies that $\mathit{gmon\text{-}impose}$, indeed, ensures for each $i \in \mathit{pre}(\mathit{gmon}')$ and each assumption in $\mathit{gmon}'(i)$ that the corresponding guarantee is acquired in $(\mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma))(j)$ for each $j \in \mathit{pre}(\Gamma)$ with $j \neq i$.

Proposition 3. *Let* $\mathit{gmon}, \mathit{gmon}' \in \mathit{GMon}$, $\Gamma : \mathit{pre}(\mathit{gmon}) \rightharpoonup \mathit{Ann}^\star$, $i \in \mathit{pre}(\mathit{gmon})$, *and* $\mathit{mod} \in \mathit{Asm}$, *be arbitrary. Let* $\Delta = \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma)$. *If* $x \in \mathit{gmon}'(i)(\mathit{mod})$ *then* $\mathsf{acq}(\mathit{invert}(\mathit{mod}), x) \in\!\!\!\!+ \Delta(j)$ *and* $\mathsf{rel}(\mathit{invert}(\mathit{mod}), x) \notin \Delta(j)$ *holds for all* $j \in \mathit{pre}(\Gamma) \setminus \{i\}$.

Proof: Assume $\Delta = \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma)$. Let $x \in (\mathit{gmon}'(i))(\mathit{mod})$ and $j \in \mathit{pre}(\Gamma) \setminus \{i\}$ be arbitrary.

Since $\mathit{mod} \in \mathit{Asm}$, either $\mathit{mod} = \mathsf{A\text{-}NW}$ or $\mathit{mod} = \mathsf{A\text{-}NR}$. We only prove the first case, the other is analogous.

If $\mathit{mod} = \mathsf{A\text{-}NW}$ then $x \in \mathit{NW}(j)$ holds for the local variable NW in the body of the function $\mathit{gmon\text{-}impose}$ because $x \in (\mathit{gmon}'(i))(\mathit{mod})$ and $i \in \mathit{pre}(\mathit{gmon}') \setminus \{j\}$. Hence, $\mathsf{acq}(\mathsf{G\text{-}NW}, x) \in\!\!\!\!+ \Delta(j)$ and $\mathsf{rel}(\mathsf{G\text{-}NW}, x) \notin \Delta(j)$ because $\Delta(j) = (\Gamma(j) \restriction \mathit{Asm}) \cdot \mathit{acq}(\mathsf{G\text{-}NW}, \mathit{NW}(j)) \cdot \mathit{acq}(\mathsf{G\text{-}NR}, \mathit{NR}(j)) \cdot \mathit{rel}(\mathsf{G\text{-}NW}, \overline{\mathit{NW}(j)}) \cdot \mathit{rel}(\mathsf{G\text{-}NR}, \overline{\mathit{NR}(j)})$, $x \in \mathit{NW}(j)$, and $x \notin \overline{\mathit{NW}(j)}$. Thus, $\mathsf{acq}(\mathit{invert}(\mathit{mod}), x) \in\!\!\!\!+ \Delta(j)$ and $\mathsf{rel}(\mathit{invert}(\mathit{mod}), x) \notin \Delta(j)$ hold. ■

The following proposition explicates some facts about transitions of the global monitor.

Proposition 4. *Let* $\mathit{gmon}, \mathit{gmon}' \in \mathit{GMon}$ *and* $\Gamma, \Delta : \mathit{pre}(\mathit{gmon}) \rightharpoonup \mathit{Ann}^\star$ *be arbitrary. If* $\mathit{gmon} \longrightarrow^{\Gamma, \Delta} \mathit{gmon}'$ *then*

1) $\mathit{pre}(\mathit{gmon}') = \mathit{pre}(\mathit{gmon})$,
2) $\mathit{pre}(\Delta) = \mathit{pre}(\Gamma)$, *and*
3) $(\Gamma(i)) \restriction \{\mathit{mod}\} = (\Delta(i)) \restriction \{\mathit{mod}\}$ *for each* $i \in \mathit{pre}(\Gamma)$ *and* $\mathit{mod} \in \mathit{Asm}$.

Proof: Assume $\mathit{gmon} \longrightarrow^{\Gamma, \Delta} \mathit{gmon}'$.

1) According to the definition of the function $\mathit{gmon\text{-}update}$, $\mathit{gmon\text{-}update}(\mathit{gmon}, \Gamma)$ is a function with pre-image $\mathit{pre}(\mathit{gmon})$. Hence, $\mathit{pre}(\mathit{gmon}') = \mathit{pre}(\mathit{gmon})$ follows from the first premise of the rule in Figure 3.

2) According to the definition of the function $\mathit{gmon\text{-}impose}$, $\mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma)$ is a function with pre-image $\mathit{pre}(\Gamma)$. Hence, $\mathit{pre}(\Delta) = \mathit{pre}(\Gamma)$ follows from the third premise of the rule in Figure 3.

3) Let $i \in \mathit{pre}(\Gamma)$ and $\mathit{mod} \in \mathit{Asm}$ be arbitrary. From $\mathit{gmon} \longrightarrow^{\Gamma, \Delta} \mathit{gmon}'$ and Figure 3, we obtain $\Delta = \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma)$. Hence, $\Delta(i) \restriction \mathit{Asm} = \Gamma(i) \restriction \mathit{Asm}$ follows from the definition of the function $\mathit{gmon\text{-}impose}$. Since $\{\mathit{mod}\} \subset \mathit{Asm}$, we have $\Delta(i) \restriction \{\mathit{mod}\} = \Gamma(i) \restriction \{\mathit{mod}\}$. ■

The following theorem implies that being well formed is an invariant for global configurations.

Theorem 4. *Let* $\mathit{gcnf} = \langle\!\langle \mathit{pool}, \mathit{mem}, \tau, \mathit{gmon} \rangle\!\rangle$ *and* $\mathit{gcnf}' = \langle\!\langle \mathit{pool}', \mathit{mem}', \tau', \mathit{gmon}' \rangle\!\rangle$ *be two global configurations such that* $\mathit{gcnf} \twoheadrightarrow_\sigma \mathit{gcnf}'$ *is derivable for some* $\sigma \in \mathit{\Sigma}$.

1) *If* gmon *is compatible with* pool *then* gmon' *is compatible with* pool'.
2) *If* $\mathit{pool}(i)$ *is a well-formed thread state for each* $i \in \mathit{pre}(\mathit{pool})$ *and the calculus for local monitor transitions properly tracks modes then* $\mathit{pool}'(j)$ *is a well-formed thread state for each* $j \in \mathit{pre}(\mathit{pool}')$.

Proof: Assume $\mathit{gcnf} \twoheadrightarrow_\sigma \mathit{gcnf}'$ is derivable for $\sigma \in \mathit{\Sigma}$.

Proof of 1st Proposition: Assume gmon is compatible with pool. To prove that gmon' is compatible with pool', we need to show $\mathit{pre}(\mathit{pool}') = \mathit{pre}(\mathit{gmon}')$ and $(\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod}) = (\mathit{gmon}'(i))(\mathit{mod})$ for all $i \in \mathit{pre}(\mathit{pool}')$ and $\mathit{mod} \in \mathit{Asm}$. Let $i \in \mathit{pre}(\mathit{pool}')$ and $\mathit{mod} \in \mathit{Asm}$ be arbitrary.

From $i \in \mathit{pre}(\mathit{pool}')$ and $\mathit{gcnf} \twoheadrightarrow_\sigma \mathit{gcnf}'$, we obtain $i \in \mathit{pre}(\mathit{pool})$. Since gmon is compatible with pool and $i \in \mathit{pre}(\mathit{pool})$, we have $\mathit{pre}(\mathit{pool}) = \mathit{pre}(\mathit{gmon})$ and $(\mathit{pool}(i)).\mathsf{mdst}(\mathit{mod}) = (\mathit{gmon}(i))(\mathit{mod})$.

Depending on which rule in Figure 2 was used to derive $\mathit{gcnf} \twoheadrightarrow_\sigma \mathit{gcnf}'$, we distinguish two cases.

Case 1) If the first rule in Figure 2 was used, we have $\mathit{gmon}' = \mathit{gmon}$ and $\mathit{pool}' = \mathit{pool}[j \mapsto \mathit{thread}']$ for some $j \in \mathit{alive}(\mathit{pool})$ and $\mathit{thread}' \in \mathit{ThSt}$. Consequently, $\mathit{pre}(\mathit{pool}') = \mathit{pre}(\mathit{pool})$ and $\mathit{pre}(\mathit{gmon}) = \mathit{pre}(\mathit{gmon}')$ hold. Hence, $\mathit{pre}(\mathit{pool}') = \mathit{pre}(\mathit{gmon}')$ follows from $\mathit{pre}(\mathit{pool}) = \mathit{pre}(\mathit{gmon})$.

We distinguish whether the ith thread performed the computation step (Case 1a) or some other thread (Case 1b).

Case 1a) If the ith thread performed the step then $\langle \mathit{pool}(i), \mathit{mem}, \tau \rangle \xrightarrow{\epsilon, \epsilon, \epsilon}_\sigma \langle \mathit{thread}', \mathit{mem}', \tau' \rangle$ follows from the second, third, and fourth premise of the first rule in Figure 2. The only rule for deriving this judgment is the rule in Figure 1, and the third premise of this rule (i.e., $\mathit{mdst}' = \mathit{update}(\mathit{mdst}, \delta)$ for $\delta = \epsilon$) ensures that the mode state of the ith thread remains unmodified. Hence, $(\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod}) = (\mathit{pool}(i)).\mathsf{mdst}(\mathit{mod}) = (\mathit{gmon}(i))(\mathit{mod}) = (\mathit{gmon}'(i))(\mathit{mod})$ follows from $\mathit{gmon}' = \mathit{gmon}$.

Case 1b) If the ith thread did not perform the step then $(\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod}) = (\mathit{gmon}'(i))(\mathit{mod})$ follows from $\mathit{pool}' =$

$pool[j \mapsto thread']$ for some $j \neq i$, $(pool(i)).\mathsf{mdst}(mod) = (gmon(i))(mod)$, and $gmon = gmon'$.
Case 2) According to the second rule in Figure 2, we have $gmon \longrightarrow^{\Gamma,\Delta} gmon'$ for some $\Gamma, \Delta : alive(pool) \longrightarrow Ann^{\star}$. Consequently, $pre(gmon') = pre(gmon)$ follows from Item 1 of Proposition 4. Moreover, according to the fourth premise of the second rule in Figure 2, we have $pre(pool') = pre(pool)$. Hence, $pre(pool') = pre(gmon')$ follows from $pre(gmon) = pre(pool)$.

We distinguish whether the ith thread is alive (Case 2a) or not (Case 2b).
Case 2a) If $i \in alive(pool)$ then $i \in pre(\Gamma)$ follows from the second premise of the second rule in Figure 2. From Item 2 of Proposition 4, we obtain $pre(\Delta) = pre(\Gamma)$ and, thus, $i \in pre(\Delta)$. That is Γ and Δ are both defined at i.

Let $mdst = (pool(i)).\mathsf{mdst}$ and $mdst' = (pool'(i)).\mathsf{mdst}$ (hence, $mdst(mod) = (gmon(i))(mod)$ holds). From the last premise of the second rule in Figure 2, we obtain $\langle pool(i), mem, \tau\rangle \xrightarrow{\mathsf{sync},\Gamma(i),\Delta(i)}_{\sigma} \langle pool'(i), mem, \tau\rangle)$. The only rule for deriving this judgment is the rule in Figure 1, and this rule ensures that $mdst' = update(mdst, \Delta(i))$.

From $gmon \longrightarrow^{\Gamma,\Delta} gmon'$, the first premise of the rule in Figure 3, and the definition of $gmon\text{-}update$, we obtain $gmon'(i) = update(gmon(i), (\Gamma(i) \upharpoonright Asm))$. Moreover, $\Delta(i) \upharpoonright \{mod\} = (\Gamma(i) \upharpoonright Asm) \upharpoonright \{mod\}$ follows from $mod \in Asm$ and $\Delta(i) \upharpoonright \{mod\} = \Gamma(i) \upharpoonright \{mod\}$ (Item 3 of Proposition 4). Since $mdst(mod) = (gmon(i))(mod)$ and $\Delta(i) \upharpoonright \{mod\} = (\Gamma(i) \upharpoonright Asm) \upharpoonright \{mod\}$, we can apply Proposition 1 to obtain $(update(mdst, \Delta(i)))(mod) = (update(gmon(i), (\Gamma(i) \upharpoonright Asm)))(mod)$, and, hence, $mdst'(mod) = (gmon'(i))(mod)$.

Consequently, $(pool'(i)).\mathsf{mdst}(mod) = mdst'(mod) = (gmon'(i))(mod)$ holds.
Case 2b) If $i \in terminated(pool)$ then $pool'(i) = pool(i)$ according to the fifth premise of the second rule in Figure 2. Since $\Gamma : alive(pool) \longrightarrow Ann^{\star}$ according to the second premise of the rule, we have $i \notin pre(\Gamma)$. From $gmon \longrightarrow^{\Gamma,\Delta} gmon'$, the first premise of the rule in Figure 3, $i \notin pre(\Gamma)$, and the definition of $gmon\text{-}update$, we obtain $gmon'(i) = gmon(i)$. Consequently, $(pool'(i)).\mathsf{mdst}(mod) = (pool(i)).\mathsf{mdst}(mod) = (gmon(i))(mod) = (gmon'(i))(mod)$ holds.

That is, $gmon'$ is compatible with $pool'$.

Proof of 2nd Proposition: Assume $pool(i)$ is a well-formed thread state for each $i \in pre(pool)$ and that the calculus for local monitor transitions properly tracks modes. We need to show that $pool'(j)$ is a well-formed thread state for each $j \in pre(pool')$. Let $j \in pre(pool')$ be arbitrary.

From $j \in pre(pool')$ and $gcnf \twoheadrightarrow_{\sigma} gcnf'$, we obtain $j \in pre(pool)$. Hence, $pool(j)$ is a well-formed thread state.

Depending on which rule in Figure 2 was used to derive $gcnf \twoheadrightarrow_{\sigma} gcnf'$, we distinguish two cases.
Case 1) If the first rule in Figure 2 was used, we have $\langle pool(k), mem, \tau\rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle pool'(k), mem', \tau'\rangle$ and $pool' = pool[k \mapsto thread']$ for some $k \in alive(pool)$, $\beta \in GEv$, and $\gamma, \delta \in Ann^{\star}$. If $k \neq j$ then $pool'(j) = pool(j)$ holds and, thus, $pool'(j)$ is a well-formed thread state.

We assume $k = j$ from now. Let $[com, lmon, mdst] = pool(j)$ and $[com', lmon', mdst'] = pool'(j)$. From $\langle pool(j), mem, \tau\rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle pool'(j), mem', \tau'\rangle$ and the rule in Figure 1 we obtain $mdst' = update(mdst, \delta)$ and that $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ is derivable for some $\alpha \in Ev$ with $\chi(\alpha) = \beta$. From $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ and our assumption that the calculus for local monitor transitions properly tracks modes, we obtain $lmon'.\mathsf{mdst} = update(lmon.\mathsf{mdst}, \delta)$. We obtain $lmon'.\mathsf{mdst} = mdst'$ from $lmon'.\mathsf{mdst} = update(lmon.\mathsf{mdst}, \delta)$, $mdst' = update(mdst, \delta)$, and $lmon.\mathsf{mdst} = mdst$.
Case 2) If the second rule in Figure 2 was used, we have $pool'(k) = pool(k)$ for all $k \in terminated(pool)$. Hence, if $k \in terminated(pool)$ then $pool'(j) = pool(j)$ holds and, thus, $pool'(j)$ is a well-formed thread state.

We assume $k \in alive(pool)$ from now. From the second rule in Figure 2, we obtain that $\langle pool(j), mem, \tau\rangle \xrightarrow{\mathsf{sync},\gamma,\delta}_{\sigma} \langle pool'(j), mem', \tau'\rangle$ is derivable for some $\gamma, \delta \in Ann^{\star}$. From $\langle pool(j), mem, \tau\rangle \xrightarrow{\mathsf{sync},\gamma,\delta}_{\sigma} \langle pool'(j), mem', \tau'\rangle$ and the rule in Figure 1 we obtain $mdst' = update(mdst, \delta)$ and that $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ is derivable for some $\alpha \in Ev$ with $\chi(\alpha) = \mathsf{sync}$. From the latter fact and from our assumption that the calculus for local monitor transitions properly tracks modes, we obtain $lmon'.\mathsf{mdst} = update(lmon.\mathsf{mdst}, \delta)$. We obtain $lmon'.\mathsf{mdst} = mdst'$ from $lmon'.\mathsf{mdst} = update(lmon.\mathsf{mdst}, \delta)$, $mdst' = update(mdst, \delta)$, and $lmon.\mathsf{mdst} = mdst$.

That is, $pool'(j)$ is a well-formed thread state. ■

The following theorem implies that justifying all assumptions is an invariant for well-formed global configurations.

Theorem 5. *Let $gcnf = \langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle$ and $gcnf' = \langle\!\langle pool', mem', \tau', gmon'\rangle\!\rangle$ be two global configurations such that $gcnf \twoheadrightarrow_{\sigma} gcnf'$ is derivable for some $\sigma \in \Sigma$.*

If all assumptions in $gcnf$ are justified and $gmon$ is compatible with $pool$ then all assumptions in $gcnf'$ are justified.

Proof: Assume all assumptions in $gcnf$ are justified, and $gmon$ is compatible with $pool$.

According to the definition of justified assumptions, we need to show

$$\forall i \in pre(pool') : \forall mod \in Asm : \forall x \in (pool'(i)).\mathsf{mdst}(mod) :$$
$$\forall j \in alive(pool') \setminus \{i\} : x \in (pool'(j)).\mathsf{mdst}(invert(mod))$$

Let $i \in \mathit{pre}(\mathit{pool}')$, $\mathit{mod} \in \mathit{Asm}$, $x \in (\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod})$, and $j \in \mathit{alive}(\mathit{pool}') \setminus \{i\}$ be arbitrary. Since $j \in \mathit{alive}(\mathit{pool}')$ and $\mathit{alive}(\mathit{pool}') \subseteq \mathit{alive}(\mathit{pool})$, we have $j \in \mathit{alive}(\mathit{pool})$. Depending on which rule in Figure 2 was used to derive $\mathit{gcnf} \twoheadrightarrow_\sigma \mathit{gcnf}'$, we distinguish two cases:

Case 1) According to the first rule in Figure 2, mode states of threads remain unchanged when this rule is applied. Hence, we obtain that $(\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod}) = (\mathit{pool}(i)).\mathsf{mdst}(\mathit{mod})$ and $(\mathit{pool}'(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod})) = (\mathit{pool}(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod}))$ hold. Thus, $x \in (\mathit{pool}(i)).\mathsf{mdst}(\mathit{mod})$ follows from $x \in (\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod})$. Since all assumptions in gcnf are justified, $x \in (\mathit{pool}(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod}))$ must hold because $j \in \mathit{alive}(\mathit{pool})$. From $(\mathit{pool}'(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod})) = (\mathit{pool}(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod}))$, we obtain $x \in (\mathit{pool}'(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod}))$.

Case 2) Let $[\mathit{com}_j, \mathit{lmon}_j, \mathit{mdst}_j] = \mathit{pool}(j)$ and $[\mathit{com}'_j, \mathit{lmon}'_j, \mathit{mdst}'_j] = \mathit{pool}'(j)$. According to the second rule in Figure 2, we have $\mathit{gmon} \longrightarrow^{\Gamma,\Delta} \mathit{gmon}'$ for some $\Gamma, \Delta : \mathit{alive}(\mathit{pool}) \longrightarrow \mathit{Ann}^\star$. Hence, we obtain $\Delta = \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma)$ from the rule in Figure 3.

Since $x \in (\mathit{pool}'(i)).\mathsf{mdst}(\mathit{mod})$, we obtain $x \in (\mathit{gmon}'(i))(\mathit{mod})$ from $\mathit{gcnf} \twoheadrightarrow_\sigma \mathit{gcnf}'$, the assumption that gcnf is well formed, and Theorem 4. Since $x \in (\mathit{gmon}'(i))(\mathit{mod})$, $\mathit{mod} \in \mathit{Asm}$, and $\Delta = \mathit{gmon\text{-}impose}(\mathit{gmon}', \Gamma)$, we obtain $\mathsf{acq}(\mathit{invert}(\mathit{mod}), x) \in \Delta(j)$ and $\mathsf{rel}(\mathit{invert}(\mathit{mod}), x) \notin \Delta(j)$ from Proposition 3. Consequently, $x \in (\mathit{update}(\mathit{mdst}_j, \Delta(j)))(\mathit{invert}(\mathit{mod}))$ according to the definition of the function update.

We have $\langle[\mathit{com}_j, \mathit{lmon}_j, \mathit{mdst}_j], \mathit{mem}, \tau\rangle \xrightarrow{\mathsf{sync},\Gamma(j),\Delta(j)}_\sigma \langle[\mathit{com}'_j, \mathit{lmon}'_j, \mathit{mdst}'_j], \mathit{mem}, \tau\rangle$ according to the second rule in Figure 2 because $j \in \mathit{alive}(\mathit{pool})$. The only rule for deriving this judgment is the rule in Figure 1, and the premise of this rule ensures that $\mathit{mdst}'_j = \mathit{update}(\mathit{mdst}_j, \Delta(j))$ holds and, thus, $x \in \mathit{mdst}'_j(\mathit{invert}(\mathit{mod}))$. Consequently, $x \in (\mathit{pool}'(j)).\mathsf{mdst}(\mathit{invert}(\mathit{mod}))$ holds.

That is, all assumptions in gcnf' are justified. ■

The following theorem implies Theorem 1 and, hence, that rely-guarantee-style reasoning is sound under the assumption that the calculus for local monitor transitions properly tracks modes and enforces guarantees.

Theorem 6. *Let $\mathit{gcnf} = \langle\!\langle \mathit{pool}, \mathit{mem}, \tau, \mathit{gmon}\rangle\!\rangle$ and $\mathit{gcnf}' = \langle\!\langle \mathit{pool}', \mathit{mem}', \tau', \mathit{gmon}'\rangle\!\rangle$ be two global configurations such that $\mathit{gcnf}' \in \mathit{greach}_\sigma(\mathit{gcnf})$ for some $\sigma \in \Sigma$.*

If all assumptions in gcnf are justified, if gcnf is well formed, and if the calculus for local monitor transitions properly tracks modes and enforces guarantees then gcnf' justifies its assumptions and provides its guarantees.

Proof: We prove that gcnf' is a well-formed global configuration that justifies its assumptions by induction over the number of steps by which gcnf' is reached from gcnf using Theorems 4 and 5 in the induction step.

Since gcnf' is well formed, $\mathit{pool}'(i)$ is a well-formed thread state for each $i \in \mathit{pre}(\mathit{pool}')$. From our assumption that the calculus for local monitor transitions enforces guarantees, we obtain that for each $i \in \mathit{pre}(\mathit{pool}')$, the thread state $\mathit{pool}'(i)$ provides its guarantees. Consequently, gcnf' provides its guarantees. ■

Proof of Theorem 2: We prove the two propositions of the theorem in Part 1 and 2, respectively.

Part 1. To prove that the calculus propertly tracks modes, we need to show that if $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{\mathit{perm}} \mathit{lmon}'$ is derivable then $\mathit{lmon}'.\mathsf{mdst} = \mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta)$. We assume that $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{\mathit{perm}} \mathit{lmon}'$ is derivable and make a case distinction about the calculus rule applied at the root of this derivation. If (M-Barrier-Local) is applied at the root of the derivation then $\mathit{lmon}'.\mathsf{mdst} = \mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta)$ holds due to the premises of this rule. If some other rule than (M-Barrier-Local) is applied at the root of the derivation then $\delta = \epsilon$ and $\mathit{lmon}'.\mathsf{mdst} = \mathit{lmon}.\mathsf{mdst}$ hold according to the conclusion of each such rule. Hence, $\mathit{lmon}'.\mathsf{mdst} = \mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta)$ follows from $\mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta) = \mathit{lmon}.\mathsf{mdst}$ (holds due to the definition of the function update for $\delta = \epsilon$).

Part 2. Let $\mathit{thread} = [\mathit{com}, \mathit{lmon}, \mathit{mdst}]$ be an arbitrary well-formed thread state. That is, $\mathit{lmon}.\mathsf{mdst} = \mathit{mdst}$ holds. Let $\mathit{mem} \in \mathit{Mem}$ and $\tau \in \mathit{Tr}$ be arbitary. We need to show that the local configuration $\mathit{lcnf} = \langle \mathit{thread}, \mathit{mem}, \tau\rangle$ provides its guarantees. We first show that lcnf provides the no-write guarantee for each variable $x \in \mathit{mdst}(\mathsf{G\text{-}NW})$ (Part 2a) and then show that lcnf also provides the no-read guarantee for each variable $y \in \mathit{mdst}(\mathsf{G\text{-}NR})$ (Part 2b).

Part 2a. We assume that $\mathit{mdst}(\mathsf{G\text{-}NW}) \neq \emptyset$ because, otherwise, the proposition is trivially fulfilled. Let $x \in \mathit{mdst}(\mathsf{G\text{-}NW})$ be arbitrary. Moreover, let $\sigma \in \Sigma$, $\beta \in \mathit{GEv}$, $\gamma, \delta \in \mathit{Ann}^\star$, and $\mathit{lcnf}' \in \mathit{LCnf}$ be arbitrary such that $\mathit{lcnf} \xrightarrow{\beta,\gamma,\delta}_\sigma \mathit{lcnf}'$ is derivable. Finally, let $\langle \mathit{thread}', \mathit{mem}', \tau'\rangle = \mathit{lcnf}'$ and $[\mathit{com}', \mathit{lmon}', \mathit{mdst}'] = \mathit{thread}'$.

From the derivability of $\mathit{lcnf} \xrightarrow{\beta,\gamma,\delta}_\sigma \mathit{lcnf}'$ and the rule in Figure 1, which is the only rule for deriving this judgment, we obtain that there exists a local event $\alpha \in \mathit{Ev}$ such that $(\mathit{com}, \mathit{mem}, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (\mathit{com}', \mathit{mem}', \tau')$ is derivable, $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{\mathit{perm}} \mathit{lmon}'$ is derivable, and $\beta = \chi(\alpha)$ holds. We perform a case distinction on α.

Case 2ai. If $\alpha = \mathsf{a}(x, e)$ for some $e \in \mathit{Exp}$ then the judgment $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{\mathit{perm}} \mathit{lmon}'$ must have been derived with the rule

(M-Assign1) or with the rule (M-Assign2) according to our calculus for local monitor transitions. Both of these rules require $lmon.\mathsf{mdst} \rhd \mathsf{maywrite}(x)$ in their premises. Hence, $x \notin mdst(\mathsf{G\text{-}NW})$ because of $lmon.\mathsf{mdst} = mdst$. This contradicts our assumption $x \in mdst(\mathsf{G\text{-}NW})$ and, hence, Case 2ai is impossible.
Case 2aii. If $\alpha = \mathsf{input}(x, ch, v)$ for some $ch \in Ch$ and $v \in Val$ then the judgment $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ must have been derived with the rule (M-Input1) or with the rule (M-Input2) according to our calculus for local monitor transitions. Both of these rules require $lmon.\mathsf{mdst} \rhd \mathsf{maywrite}(x)$ in their premises. Hence, $x \notin mdst(\mathsf{G\text{-}NW})$ because $lmon.\mathsf{mdst} = mdst$. This contradicts our assumption $x \in mdst(\mathsf{G\text{-}NW})$ and, hence, Case 2aii is impossible.
Case 2aiii. If $\alpha \notin \{\mathsf{a}(y, e), \mathsf{input}(y, ch, v) \mid y = x, e \in Exp, ch \in Ch, v \in Val\}$ then we obtain $mem'(x) = mem(x)$ from the derivability of $(com, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem', \tau')$ and the language semantics (see Appendix B).

That is, $lcnf$ provides the no-write guarantee for each variable $x \in mdst(\mathsf{G\text{-}NW})$.

Part 2b. We assume that $mdst(\mathsf{G\text{-}NR}) \neq \emptyset$ because, otherwise, the proposition is trivially fulfilled. Let $y \in mdst(\mathsf{G\text{-}NR})$ be arbitrary. Moreover, let $\sigma \in \Sigma$, $\beta \in GEv$, $\gamma, \delta \in Ann^\star$, and $lcnf' \in LCnf$ be arbitrary such that $lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf'$ is derivable. Finally, let $\langle thread', mem', \tau'\rangle = lcnf'$ and $[com', lmon', mdst'] = thread'$.

From the derivability of $lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf'$ and the rule in Figure 1, which is the only rule for deriving this judgment, we obtain that there exists a local event $\alpha \in Ev$ such that $(com, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem', \tau')$ is derivable, $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ is derivable, and $\beta = \chi(\alpha)$ holds. We perform a case distinction on α and show for each of the five cases that, for each $v \in Val$, the judgment $\langle thread, mem[y \mapsto v], \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle thread', mem', \tau'\rangle$ or the judgment $\langle thread, mem[y \mapsto v], \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle thread', mem'[y \mapsto v], \tau'\rangle$ is derivable. Let $v \in Val$ be arbitrary.

Case 2bi. If

$$\alpha \in \left\{ \begin{array}{c} \mathsf{a}(x,e), \mathsf{b}(e, com_1, com_2), \mathsf{enter}(e, com_1), \\ \mathsf{more}(e, com_1), \mathsf{leave}(e, com_1), \mathsf{output}(ch, e, v') \end{array} \middle| \begin{array}{c} x \in Var, e \in Exp, y \in vars(e), \\ com_1, com_2 \in Com, ch \in Ch, v' \in Val \end{array} \right\}$$

then $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ must have been derived by one of the following rules: (M-Assign1), (M-Assign2), (M-Branch), (M-Enter), (M-More), (M-Leave), and (M-Output). Each of these rules requires $lmon.\mathsf{mdst} \rhd \mathsf{mayread}(e)$ in its premises. Hence, $y \notin mdst(\mathsf{G\text{-}NR})$ because $lmon.\mathsf{mdst} = mdst$ and $y \in vars(e)$. This contradicts our assumption $y \in mdst(\mathsf{G\text{-}NR})$ and, hence, Case 2bi is impossible.

Case 2bii. If $\alpha = \mathsf{a}(y, e)$ for some $e \in Exp$ with $y \notin vars(e)$ then, according to Appendix B, $mem' = mem[y \mapsto v']$ must hold for some value $v' \in Val$ for which $e, mem \Downarrow v'$ is derivable (argument is by induction on the number of applications of the rule for sequential composition, using the rule for assignments in the base case). Since $y \notin vars(e)$ and $e, mem \Downarrow v'$ is derivable, $e, mem[y \mapsto v] \Downarrow v'$ must also be derivable according to the properties of the function $vars$. Since $\alpha = \mathsf{a}(y, e)$, $e, mem[y \mapsto v] \Downarrow v'$, and $(com, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[y \mapsto v'], \tau')$ is derivable, $(com, mem[y \mapsto v], \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[y \mapsto v'], \tau)$ must also be derivable according to Appendix B (apply rules in the same way as in the first derivation). From $mem' = mem[y \mapsto v']$ and Figure 1, we obtain that $\langle thread, mem[y \mapsto v], \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle thread', mem', \tau'\rangle$ is derivable, which concludes this case.

Case 2biii. If $\alpha = \mathsf{input}(y, ch, v')$ for some $ch \in Ch$ and $v' \in Val$ then $mem' = mem[y \mapsto v']$ must hold according to Appendix B (argument is by induction on the number of applications of the rule for sequential composition, using the rule for input commands in the base case). Since $\alpha = \mathsf{input}(x, ch, v')$ and $(com, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[y \mapsto v'], \tau')$ is derivable, $(com, mem[y \mapsto v], \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[y \mapsto v'], \tau)$ must also be derivable according to Appendix B (apply rules in the same way as in the first derivation). From $mem' = mem[y \mapsto v']$ and Figure 1, we obtain that $\langle thread, mem[y \mapsto v], \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle thread', mem', \tau'\rangle$ is derivable, which concludes this case.

Case 2biv. If $\alpha \in \{\mathsf{a}(x, e), \mathsf{input}(x, ch, v') \mid x \in Var, x \neq y, e \in Exp, y \notin vars(e), ch \in Ch, v' \in Val\}$ then $mem' = mem[x \mapsto v']$ must hold for some $v' \in Val$ according to Appendix B (argument is by induction on the number of applications of the rule for sequential composition). Since $x \neq y$ and $(com, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[x \mapsto v'], \tau')$ is derivable, $(com, mem[y \mapsto v], \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[y \mapsto v][x \mapsto v'], \tau)$ must also be derivable according to Appendix B (apply rules in the same way as in the first derivation). Moreover, $mem[y \mapsto v][x \mapsto v'] = mem[x \mapsto v'][y \mapsto v]$ follows from $x \neq y$. Consequently, $\langle thread, mem[y \mapsto v], \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle thread', mem'[y \mapsto v], \tau'\rangle$ is derivable, which concludes this case.

Case 2bv. If

$$\alpha \in \left\{ \begin{array}{c} \mathsf{s}, \mathsf{b}(e, com_1, com_2), \mathsf{join}, \mathsf{enter}(e, com_1), \mathsf{more}(e, com_1), \\ \mathsf{leave}(e, com_1), \mathsf{output}(ch, e, v), \mathsf{sync}, \mathsf{term} \end{array} \middle| \begin{array}{c} e \in Exp, y \notin vars(e), \\ com_1, com_2 \in Com, ch \in Ch, v \in Val \end{array} \right\}$$

then $mem' = mem$ must hold according to Appendix B. Since $y \notin vars(e)$ and $(com, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem, \tau')$ is derivable, $(com, mem[y \mapsto v], \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_\sigma (com', mem[y \mapsto v], \tau)$ must also be derivable according to Appendix B (apply rules like in the first derivation). Consequently, $\langle thread, mem[y \mapsto v], \tau\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle thread', mem'[y \mapsto v], \tau'\rangle$ is derivable, which concludes this case.

That is, $lcnf$ provides the no-read guarantee for each variable $y \in mdst(\textsf{G-NR})$. $\blacksquare$

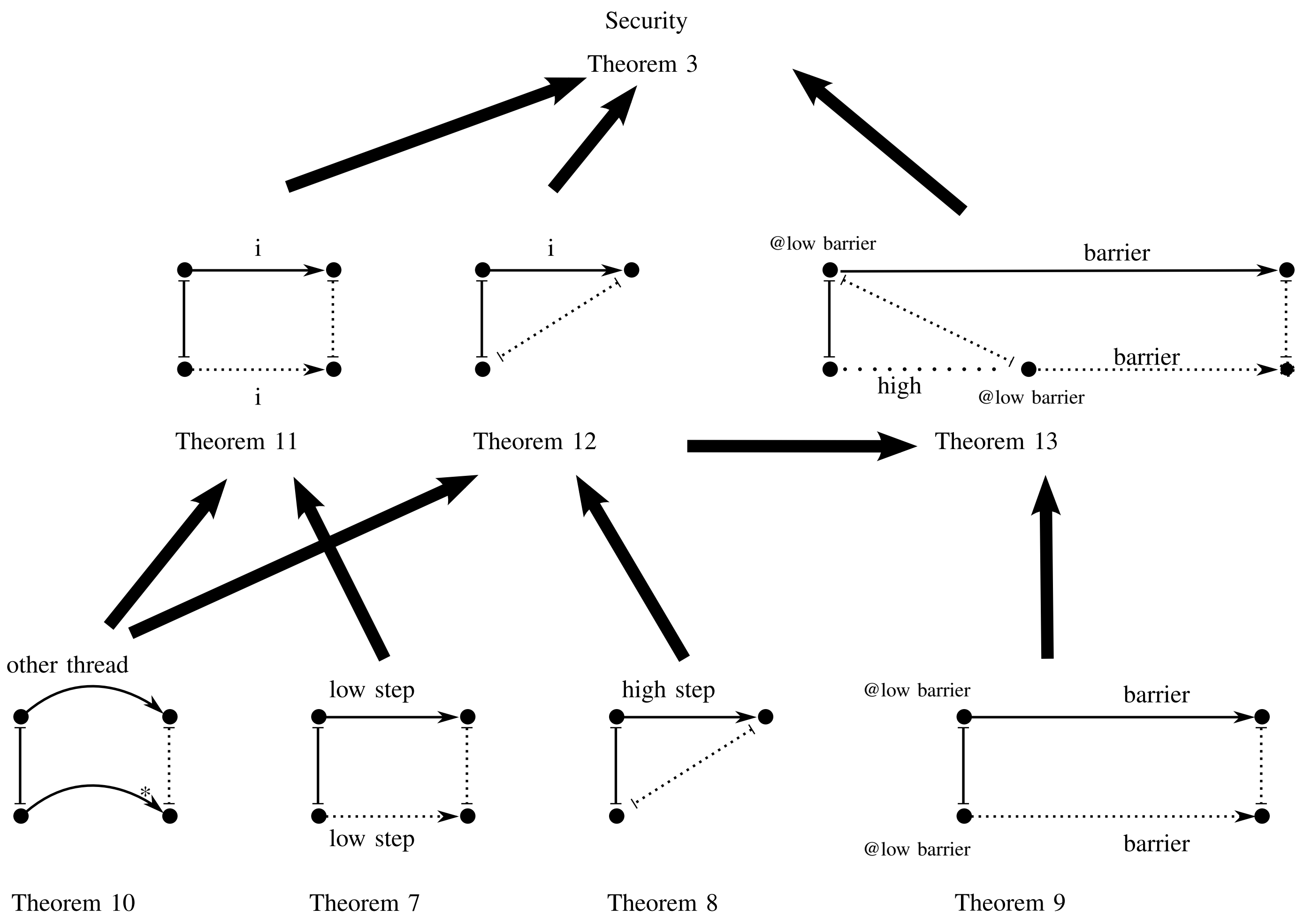

Fig. 9. Diagram of key theorems for proof of soundness of monitoring

Appendix E
Soundness of Monitoring

In this section, we prove Theorem 3, which states that the local monitors, combined with the global monitor, enforce security.

Figure 9 provides an overview of the proof. Starting from two identical global configurations $gcnf_1^0$ and $gcnf_2^0$, and two ℓ-equivalent strategies σ_1 and σ_2, we show that for each step that the first configuration can take (using σ_1), the second one can take zero or more steps (using σ_2) to a ℓ-equivalent global configuration. Theorem 11 says that if the first configuration takes a "low" step, then the second configuration can also take a low step. Theorem 12 says that if the first configuration takes a "high" step, then the second configuration does not need to take any step. Theorem 13 says that if the first configuration takes a barrier step, then the second configuration takes zero or more high steps to reach the barrier, whereupon it can take a barrier step. Theorems 7, 8, and 9 describe the ℓ-equivalence of individual threads, and Theorem 10 states that ℓ-equivalence of threads is preserved when the memory and trace changes due to other threads taking a step. Theorems 7, 8, 9, and 10 are used in the proofs of Theorems 11, 12, and 13, as indicated in Figure 9.

The proof of Theorem 13 is more complex than the others, and is contained in Section E-F. In it, we define a stronger notion of ℓ-equivalence between thread states: time-insensitive-ℓ-equivalence, which in essence requires the components of equivalent thread states to be identical if the pc stack, termination level, and blocking level are all below ℓ (i.e., it ignores the timing level). This time-insensitive-ℓ-equivalence allows us to show that if one thread exits a while loop or conditional command with the pc stack, termination level and blocking level all below ℓ, then the other thread will also exit the same while loop or conditional, even though the timing level may be above ℓ.

A. Static bounds

In Section C-A, we introduced the idea that the local monitor must examine conditional commands and loops in order to approximate the behavior of the monitor on the code not executed.

More specifically, the local monitor invokes a static oracle, passing in some of the local monitor state, and the oracle returns a tuple: $\mathsf{SB}(com, \Gamma, lmdst, \overline{pc}, time, term, block) = (\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$. Security level ℓ_{sb} will be pushed on to the program counter stack, and will remain on the stack until the branch or loop command is exited. Thus, ℓ_{sb} should be an upper bound on the information that determines which branch to take (for branch commands), or how many times to execute the loop body (for loop commands). The tuple $(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a static branching environment, and will be pushed onto the stack of static branching environments, to be incorporated into the local monitor's timing level, termination level, blocking level, and typing context when the monitor exits the branch or loop command.

Intuitively, the static branching environment is meant to summarize the behavior of the local monitor during *interesting possible executions of com, other than the execution that actually occurs*. Other executions of com are interesting only if they may reveal secret information, i.e., only if $\ell_{\mathsf{sb}} \neq \bot$. If $\ell_{\mathsf{sb}} = \bot$, then other possible executions of com are not interesting, and we do not have any restrictions on static branching environment $(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.

If, however, $\ell_{\mathsf{sb}} \neq \bot$ then there may be other executions of com where different control flow decisions occur based on secret information bounded by ℓ_{sb}. In that case, static branching environment $(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ summarizes these other possible executions. Assume that we have a monitored execution of command com, and just before executing com the local monitor configuration is $\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \rangle$, and the local monitor's configuration after successfully executing com is $\langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block' \rangle$.

Then the static branching environment timing level $time_{\mathsf{sb}}$ is an upper bound on the local monitor's timing level at the end of com (i.e., $time' \sqsubseteq time_{\mathsf{sb}}$). (Moreover, for loops, $time_{\mathsf{sb}}$ is an upper bound of the local monitor's timing level at the start of every loop iteration.)

Similarly, the static branching environment termination level $term_{\mathsf{sb}}$, and the blocking level $block_{\mathsf{sb}}$ are upper bounds on, respectively, the local monitor's termination level and the local monitor's blocking level at the end of com (i.e., $term' \sqsubseteq term_{\mathsf{sb}}$ and $block' \sqsubseteq block_{\mathsf{sb}}$). (Moreover, for loops, $term_{\mathsf{sb}}$ and $block_{\mathsf{sb}}$ are upper bounds of the local monitor's termination and blocking levels at the start of every loop iteration.)

Also, the static branching environment typing environment Γ_{sb} has the same domain as the local monitor's typing environment at the end of com (i.e., $pre(\Gamma') = pre(\Gamma_{\mathsf{sb}})$), and for each variable x, Γ_{sb} provides an upper bound of the local monitor's level for variable x (i.e., $\forall x \in Var.\ \Gamma'\langle x \rangle \sqsubseteq \Gamma_{\mathsf{sb}}\langle x \rangle$). (Moreover, for loops, this holds true for the local monitor's typing environment at the start of every loop iteration.)

There is one additional constraint on the soundness of the static monitor. If the command com contains any barrier synchronization, then $\ell_{\mathsf{sb}} = \bot$. That is, the decision to execute a barrier synchronization must depend only on public (i.e., level $\bot$) information.

We can phrase the informal requirements on the static oracle more formally, as follows. If we have a thread pool where the ith thread is com and com is the branching or looping command, and Γ, $lmdst$, $time$, $term$, and $block$ are the relevant parts of the current state of the local monitor of the ith thread, and the monitored execution of com terminates (i.e., the ith thread becomes stop) with monitor state $time'$, $term'$, $block'$, Γ' and $\mathsf{SB}(com, \Gamma, lmdst, \overline{pc}, time, term, block) = (\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}\Gamma_{\mathsf{sb}})$, and $\ell_{\mathsf{sb}} \neq \bot$ then $time_{\mathsf{sb}}$, $term_{\mathsf{sb}}$, $block_{\mathsf{sb}}$ and Γ_{sb} are upper bounds of $time'$, $term'$, $block'$, and Γ' respectively. Moreover, for loops, these are upper bounds on the monitor state every time the loop guard is executed.

Note that this holds true regardless of the current memory, trace, or other threads in the thread pool.

We define soundness of the static oracle by describing the conditions under which the tuple $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a *conservative approximation* of the branching, timing, termination, blocking, and floating behavior of a local monitor for command com, with local monitor configuration $\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \rangle$.

Definition 2 (Conservative approximation of branching, timing, termination, blocking, and floating behavior)**.** *Let $com \in Com$, Γ be an environment, $lmdst \in MdSt$, and time, term, and block be security levels, such that*

- *For all variables x, $lmdst \triangleright \mathsf{othersmightread}(x) \implies \Gamma\langle x \rangle \sqsubseteq \mathcal{L}(x)$*
- *For all variables x, $lmdst \triangleright \mathsf{othersmightwrite}(x) \implies \mathcal{L}(x) \sqsubseteq \Gamma\langle x \rangle$*

Let com_{start} and com_{end} be defined as either $com_{start} = com$ and $com_{end} = \mathsf{stop}$ or $com_{start} = com; com'$ and $com_{end} = \mathsf{stop}; com'$ for some arbitrary command com'.

We say that the tuple $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a conservative approximation *of the branching, timing, termination, blocking, and floating behavior for com, Γ, lmdst, $\overline{pc}$, $\overline{br}$, time, term, block if all of the following hold.*

1) $term_{\mathsf{sb}} \sqsubseteq time_{\mathsf{sb}}$
2) *For all variables x, $lmdst \triangleright \mathsf{othersmightread}(x) \implies \Gamma_{\mathsf{sb}}\langle x \rangle \sqsubseteq \mathcal{L}(x)$*
3) *For all variables x, $lmdst \triangleright \mathsf{othersmightwrite}(x) \implies \mathcal{L}(x) \sqsubseteq \Gamma_{\mathsf{sb}}\langle x \rangle$*

4) *For all pool* $\in$ *PSt, i* $\in$ *pre(pool),* $\sigma \in \Sigma$*, mem* $\in$ *Mem,* $\tau \in$ *Tr, gmon* $\in$ *GMon, and pool'* $\in$ *PSt, mem'* $\in$ *Mem,* $\tau' \in$ *Tr, gmon'* $\in$ *GMon and* Γ'*, lmdst', $\overline{pc}'$, $\overline{br}'$, time', term', and block' such that*
   - $pool(i) = [com_{start}, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
   - $\langle\langle pool, mem, \tau, gmon\rangle\rangle \twoheadrightarrow^*_\sigma \langle\langle pool', mem', \tau', gmon'\rangle\rangle$
   - $pool'(i) = [com_{end}, \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle, mdst']$

   we have:
   - *Either* $\ell_{\mathsf{sb}} = \bot$ *or (*$time' \sqsubseteq time_{\mathsf{sb}}$ *and* $term' \sqsubseteq term_{\mathsf{sb}}$ *and* $block' \sqsubseteq block_{\mathsf{sb}}$ *and* $pre(\Gamma') = pre(\Gamma_{\mathsf{sb}})$ *and* $\forall x \in Var.\ \Gamma'\langle x\rangle \sqsubseteq \Gamma_{\mathsf{sb}}\langle x\rangle$*)*
5) *For all pool* $\in$ *PSt, i* $\in$ *pre(pool),* $\sigma \in \Sigma$*, mem* $\in$ *Mem,* $\tau \in$ *Tr, gmon* $\in$ *GMon such that*
   - $pool(i) = [com_{start}, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
   - *execution of* $pool(i)$ *may fail to terminate normally because either* $pool(i)$ *diverges before reaching command* com_{end} *or the monitor for* $pool(i)$ *blocks before reaching command* com_{end}

   we have:
   - $\ell_{\mathsf{sb}} \sqsubseteq term_{\mathsf{sb}}$ *or* $\ell_{\mathsf{sb}} \sqsubseteq block_{\mathsf{sb}}$
6) ***If*** $\ell_{\mathsf{sb}} \neq \bot$ ***then*** *sync-free*(com).
7) ***If*** $com = \mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}$ ***then*** $\Gamma\langle e\rangle \sqcup time \sqsubseteq \ell_{\mathsf{sb}}$.
8) ***If*** $com = \mathsf{while}\ e\ \mathsf{do}\ com_1\ \mathsf{od}$ ***then*** *for all pool* $\in$ *PSt, i* $\in$ *pre(pool),* $\sigma \in \Sigma$*, mem* $\in$ *Mem,* $\tau \in$ *Tr, gmon* $\in$ *GMon, and pool'* $\in$ *PSt, mem'* $\in$ *Mem,* $\tau' \in$ *Tr, gmon'* $\in$ *GMon, and* Γ'*, lmdst', $\overline{pc}'$, $\overline{br}'$, time', term', and block' such that*
   - $pool(i) = [com_{start}, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
   - $\langle\langle pool, mem, \tau, gmon\rangle\rangle \twoheadrightarrow^*_\sigma \langle\langle pool', mem', \tau', gmon'\rangle\rangle$
   - $pool'(i) = [com_{more}, \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle, mdst']$ *(where* $com_{more} = \mathsf{more}\ e\ \mathsf{do}\ com_1\ \mathsf{od}$ *if* $com_{start} = com$ *and* $com_{more} = (\mathsf{more}\ e\ \mathsf{do}\ com_1\ \mathsf{od}); com'$ *if* $com_{start} = com; com'$*).*

   we have all of the following:
   - $\Gamma'\langle e\rangle \sqcup time' \sqsubseteq \ell_{\mathsf{sb}}$.
   - *Either* $\ell_{\mathsf{sb}} = \bot$ *or (*$time' \sqsubseteq time_{\mathsf{sb}}$ *and* $term' \sqsubseteq term_{\mathsf{sb}}$ *and* $block' \sqsubseteq block_{\mathsf{sb}}$ *and* $pre(\Gamma') = pre(\Gamma_{\mathsf{sb}})$ *and* $\forall x \in Var.\ \Gamma'\langle x\rangle \sqsubseteq \Gamma_{\mathsf{sb}}\langle x\rangle$*)*

Note that in the clauses that require that ℓ_{sb} is an upper bound of the branching or looping decision, we are more precise when the current thread has exclusive write access to all variables in e, i.e., $lmdst \triangleright \mathsf{exclusivewrite}(e)$. Specifically, if $lmdst \triangleright \mathsf{exclusivewrite}(e)$ then other threads cannot concurrently change the value that e will evaluate to, and thus $\Gamma\langle e\rangle$ is an upper bound on the information that may be learned by evaluating e here (i.e., the timing of the evaluation of e relative to other threads—bounded above by the timing level *time*—is not relevant). Therefore, in that case, ℓ_{sb} must be an upper bound only of $\Gamma\langle e\rangle$. Otherwise, if $lmdst \triangleright \mathsf{othersmightwrite}(e)$, then ℓ_{sb} must be an upper bound of $\Gamma\langle e\rangle \sqcup time$, since the timing of when the expression is evaluated (relative to concurrent updates to variables in e) may influence that value of the expression.

The static bounds oracle is defined in Appendix F, and a sketch of the proof of soundness.

B. Local configuration results

We first define some properties about single executions of programs. These properties describe invariants and formalize the intended meaning of monitor state.

1) Properties of local monitors: Each time execution enters a conditional or a loop, a special "control state" term is added to the program's continuation, either a join command (for conditionals) or a $\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ command (for loops). We define set $\mathsf{Continuation_n}$ to be the subset of commands that have exactly n occurrences of $\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ or join as subcommands, and (for $n > 0$), start with a $\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ or join command. This definition will be useful, both to specify that the depth of a program counter level stack is equal to the number of $\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and join commands in the continuation, and also to identify the continuation that corresponds to the continuation after a particular loop or conditional.

$$\begin{aligned}
\mathsf{Continuation_0} &= \{com \mid com \text{ does not contain a subcommand}\\
&\qquad\qquad \text{of the form } \mathsf{join} \text{ or } \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}\}\\
\mathsf{Continuation_{n+1}} &= \{\mathsf{join}; com; com' \mid com \in \mathsf{Continuation_0} \text{ and } com' \in \mathsf{Continuation_n}\}\\
&\cup \{\mathsf{join}; com' \mid com' \in \mathsf{Continuation_n}\}\\
&\cup \{\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}; com'; com'' \mid com' \in \mathsf{Continuation_0} \text{ and } com'' \in \mathsf{Continuation_n}\}\\
&\cup \{\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}; com' \mid com' \in \mathsf{Continuation_n}\}
\end{aligned}$$

Property 1 (Invariants of local monitors). *Let $pool_0 \in PSt$, $\sigma \in \Sigma$, $gmon \in GMon$ and $\langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle \in reach_\sigma(pool_0)$. If $pool(i) = [com, \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle, mdst]$ for some $i \in pre(pool)$ then all of the following hold.*

1) $|\overline{br}| = |\overline{pc}|$
2) *Let $n = |\overline{br}|$. Either $com = com'; com''$ where $com' \in \mathsf{Continuation_0}$ and $com'' \in \mathsf{Continuation_n}$ or $com \in \mathsf{Continuation_n}$.*
3) $term \sqsubseteq time$
4) $pre(\Gamma) = \{x \mid x \in FloatVar \wedge (lmdst \triangleright \mathsf{exclusiveread}(x) \vee lmdst \triangleright \mathsf{exclusivewrite}(x))\}$
5) *For all variables x, $lmdst \triangleright \mathsf{othersmightread}(x) \implies \Gamma\langle x\rangle \sqsubseteq \mathcal{L}(x)$*
6) *For all variables x, $lmdst \triangleright \mathsf{othersmightwrite}(x) \implies \mathcal{L}(x) \sqsubseteq \Gamma\langle x\rangle$*
7) $lmdst = mdst$.
8) *If $com = \mathsf{term}$ then $\overline{pc} = \epsilon$, $\overline{br} = \epsilon$, $term = \bot$, and $block = \bot$.*
9) *For all $k \in |\overline{br}|$, the tuple $(\overline{pc}(k), time(k), term(k), block(k), \Gamma(k))$ (where $\overline{br}(k) = (time(k), term(k), block(k), \Gamma(k))$) is a conservative approximation (Definition 2) of the branching, timing, termination, and floating behavior for some com_k, Γ_k, $lmdst$, $\overline{pc}$, $\overline{br}$, time, term, block such that there is a global configuration $\langle\!\langle pool'_k, mem'_k, \tau'_k, gmon'_k\rangle\!\rangle$ with*

$$pool'_k(i) = [com'_k, \langle \Gamma_k, lmdst_k, \overline{pc}_k, \overline{br}_k, time_k, term_k, block_k\rangle, mdst_k]$$

and com_k is the redex of com'_k and $\langle\!\langle pool'_k, mem'_k, \tau'_k, gmon'_k\rangle\!\rangle \in reach_\sigma(pool_0)$ and $\langle\!\langle pool'_k, mem'_k, \tau'_k, gmon'_k\rangle\!\rangle \twoheadrightarrow^*_\sigma \langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle$.

Proof: We proceed by induction on the steps of the global configuration. In the base case, every local configuration for the global configuration $pool_0$ is of the form

$$[com, \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle, mdst]$$

where

$$\begin{aligned}
time &= \bot\\
term &= \bot\\
block &= \bot\\
\overline{pc} &\text{ is empty}\\
\overline{br} &\text{ is empty}\\
com &\in \mathsf{Continuation_0}\\
pre(\Gamma) &= \{x \mid x \in FloatVar \wedge (lmdst \triangleright \mathsf{exclusiveread}(x) \vee lmdst \triangleright \mathsf{exclusivewrite}(x))\}\\
\forall x \in pre(\Gamma).\Gamma(x) &= \mathcal{L}(x)\\
lmdst &= mdst
\end{aligned}$$

All of the invariants hold.

For the inductive case, assume the invariants hold true of all local configurations in a global configuration

$$gcnf = \langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle,$$

and consider

$$gcnf_1 \twoheadrightarrow_\sigma \langle\!\langle pool', mem', \tau', gmon'\rangle\!\rangle$$

There are two possible inference rules that allow a global configuration to take a step. In one of them, all local configurations take a synchronization step.

$$\langle pool(i), mem, \tau\rangle \xrightarrow{\mathsf{sync}, \gamma, \Delta(i)}_\sigma \langle pool'(i), mem, \tau\rangle$$

In the other inference rule that allows a global configuration to take a step, one local configuration takes a step:

$$\langle pool(i), mem, \tau\rangle \xrightarrow{\epsilon, \gamma, \epsilon}_\sigma \langle pool'(i), mem', \tau'\rangle$$

We cover both of these cases at the same time by performing an induction on the local monitor step associated with the local configuration step. Consider

$$\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle \longrightarrow^{\delta,\alpha}_{perm} \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle$$

Assume that the invariants hold for $pool(i)$, and we need to show they hold for $pool'(i)$.

- **M-Skip**, **M-Output**, **M-Term**, **M-More**. Here $\delta = \epsilon$, $\Gamma' = \Gamma$ and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}$ and $\overline{br}' = \overline{br}$ and $time' = time$ and $term' = term$ and $block' = block$.
  All invariants except for Condition 8 hold trivially, since they held for $pool(i)$. For Condition 8, if the monitor rule was **M-Term**, then from the premises of **M-Term**, we have $\overline{pc} = \epsilon$, $\overline{br} = \epsilon$, $term = \bot$, and $block = \bot$, as required.
- **M-Assign1** Here $\delta = \epsilon$, $\Gamma' = \Gamma\langle x \mapsto_{lmdst} \ell\rangle$ where
$$\ell = \Gamma\langle e\rangle \sqcup time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block$$
  and
$$lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)]$$
  and $\ell \sqsubseteq \mathcal{L}(x)$ and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}$ and $\overline{br}' = \overline{br}$ and $time' = time$ and $term' = term$ and $block' = block$.
  All invariants hold either by simple inspection or because they held for $pool(i)$.
- **M-Assign2** Here $\delta = \epsilon$, $\Gamma' = \Gamma\langle x \mapsto_{lmdst} \ell\rangle$ where
$$\ell = \Gamma\langle e\rangle \sqcup time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block$$
  and
$$lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)]$$
  and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}$ and $\overline{br}' = \overline{br}$ and $time' = time$ and $term' = term$ and $block' = block$.
  All invariants hold either by simple inspection or because they held for $pool(i)$.
- **M-Input1** Here $\delta = \epsilon$, $\Gamma' = \Gamma\langle x \mapsto_{lmdst} \ell\rangle$ where
$$\ell = ch \sqcup time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block$$
  and
$$lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{othersmightread}(x)]$$
  and $\ell \sqsubseteq \mathcal{L}(x)$ and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}$ and $\overline{br}' = \overline{br}$ and $time' = time$ and $term' = term$ and $block' = block$.
  All invariants hold either by simple inspection or because they held for $pool(i)$.
- **M-Input2** Here $\delta = \epsilon$, $\Gamma' = \Gamma\langle x \mapsto_{lmdst} \ell\rangle$ where
$$\ell = ch \sqcup time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block$$
  and
$$lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{exclusiveread}(x)]$$
  and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}$ and $\overline{br}' = \overline{br}$ and $time' = time$ and $term' = term$ and $block' = block$.
  All invariants hold either by simple inspection or because they held for $pool(i)$.
- **M-Barrier-Local** Here $\overline{pc}' = \overline{pc}$ and $\overline{br}' = \overline{br}$ and $time' = \bot$ and $term' = term = \bot$ and $block' = block = \bot$ and $(\sqcup\overline{pc}) = \bot$.
  Condition 1 ($|\overline{br}'| = |\overline{pc}'|$) holds because $|\overline{br}| = |\overline{pc}|$.
  Condition 2 holds because if $pool(i)$.com $\in \mathsf{Continuation_n}$ then $pool'(i)$.com $\in \mathsf{Continuation_n}$.
  Condition 3 holds because $term' = \bot \sqsubseteq \bot = time'$.
  Condition 4 holds because $pre(\Gamma') = \{x \mid x \in FloatVar \wedge (lmdst' \triangleright \mathsf{exclusiveread}(x) \vee lmdst' \triangleright \mathsf{exclusivewrite}(x))\}$ is true by the premise of the M-Barrier-Local rule.
  Condition 5 For all variables x, $lmdst' \triangleright \mathsf{othersmightread}(x) \implies \Gamma'\langle x\rangle \sqsubseteq \mathcal{L}(x)$ is true by the premises of the M-Barrier-Local rule.
  Condition 6 For all variables x, $lmdst' \triangleright \mathsf{othersmightwrite}(x) \implies \mathcal{L}(x) \sqsubseteq \Gamma'\langle x\rangle$ is true by the premises of the M-Barrier-Local rule.
  Condition 7 $lmdst' = mdst'$ since $lmdst' = update(lmdst, \delta)$ and $mdst' = update(mdst, \delta)$. But by the inductive hypothesis we have $lmdst = mdst$ so $lmdst' = mdst'$ as required.
  Condition 8 is trivially true, since $com' \neq \mathsf{term}$.
  Condition 9 is true because it held for $pool(i)$, and this global configuration is reachable from the previous global configuration.
- **M-Branch** Here $\delta = \epsilon$, $\Gamma' = \Gamma$ and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}{\cdot}\ell_{\mathsf{sb}}$ and $\overline{br}' = \overline{br}{\cdot}(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where
$$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}, \Gamma, lmdst, \overline{pc}, time, term, block)$$
  and $time' = time$ and $term' = term$ and $block' = block$.

Except for conditions 2 and 9, all invariants hold trivially, since they held for $pool(i)$.
For condition 2, by inversion on the command, we see that a conditional statement $\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}$ was reduced to either $com_1; \mathsf{join}$ or $com_2; \mathsf{join}$, and thus the new command for the local configuration has one more join statement than the previous command, and the size of the program counter stack has also increased by one.
For condition 9, by Lemma 14, $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a conservative approximation of the timing, termination, blocking, and floating behavior for com, Γ, $lmdst$, $\overline{pc}$, $\overline{br}$, $time$, $term$, and $block$ where com is the command of the local configuration before the step, as required.

- **M-Join** Here $\delta = \epsilon$, $lmdst' = lmdst$ and $\overline{pc}'{\cdot}\ell_{\mathsf{sb}} = \overline{pc}$ and $\overline{br}'{\cdot}(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \overline{br}$ and $time' = time \sqcup time_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}}$ and $term' = term \sqcup term_{\mathsf{sb}}$ and $block' = block \sqcup block_{\mathsf{sb}}$ and
$$\Gamma' = \lambda x.\begin{cases}\Gamma(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma) \cap pre(\Gamma_{\mathsf{sb}})\\ \Gamma(x) & \text{if } x \in pre(\Gamma) \setminus pre(\Gamma_{\mathsf{sb}})\\ \mathsf{undef} & \text{otherwise}\end{cases}$$
Since both the pc stack and the branch information stack decrease in size by one element, condition 1 is satisfied.
By inversion on the command step, we see that the command has one less join in it than previously, and the size of the pc stack has reduced by one, so condition 2 is satisfied.
Since $term \sqsubseteq time$ and $term_{\mathsf{sb}} \sqsubseteq time_{\mathsf{sb}}$ (by Definition 2), we have $term' \sqsubseteq time'$, satisfying condition 3.
Condition 4 is satisfied since $pre(\Gamma) = pre(\Gamma')$.
Conditions 5 and 6 hold due to the definition of conservative approximation (Definition 2).
Conditions 7 and 8 are trivially satisfied, and condition 9 follows immediately since it held for $pool(i)$.
- **M-Enter** Here $\delta = \epsilon$, $\Gamma' = \Gamma$ and $lmdst' = lmdst$ and $\overline{pc}' = \overline{pc}{\cdot}\ell_{\mathsf{sb}}$ and $\overline{br}' = \overline{br}{\cdot}(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where
$$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com'\ \mathsf{od}, \Gamma, lmdst, \overline{pc}, time, term, block)$$
and $time' = time$ and $term' = term$ and $block' = block$.
Except for conditions 2 and 9, all invariants hold trivially, since they held for $pool(i)$.
For condition 2, by inversion on the command, we see that a while loop $\mathsf{while}\ e\ \mathsf{do}\ com'\ \mathsf{od}$ was reduced to $\mathsf{more}\ e\ \mathsf{do}\ com'\ \mathsf{od}$, and thus the new command for the local configuration has one more more statement than the previous command, and the size of the program counter stack has also increased by one.
For condition 9, by Lemma 14, $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a conservative approximation of the timing, termination, blocking, and floating behavior for com, Γ, $lmdst$, $\overline{pc}$, $\overline{br}$, $time$, $term$, and $block$ where com is the command of the local configuration before the step, as required.
- **M-Leave** Here $\delta = \epsilon$, $lmdst' = lmdst$ and $\overline{pc}'{\cdot}\ell_{\mathsf{sb}} = \overline{pc}$ and $\overline{br}'{\cdot}(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \overline{br}$ and $time \sqcup time_{\mathsf{sb}} \sqsubseteq time'$ and $term' = term \sqcup term_{\mathsf{sb}}$ and $block' = block \sqcup block_{\mathsf{sb}}$ and
$$\Gamma' = \lambda x.\begin{cases}\Gamma(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma) \cap pre(\Gamma_{\mathsf{sb}})\\ \Gamma(x) & \text{if } x \in pre(\Gamma) \setminus pre(\Gamma_{\mathsf{sb}})\\ \mathsf{undef} & \text{otherwise}\end{cases}$$
Since both the pc stack and the branch information stack decrease in size by one element, condition 1 is satisfied.
By inversion on the command step, we see that the command has one less more in it than previously, and the size of the pc stack has reduced by one, so condition 2 is satisfied.
Since $term \sqsubseteq time$ and $term_{\mathsf{sb}} \sqsubseteq time_{\mathsf{sb}}$ (by Definition 2), we have $term' \sqsubseteq time'$, satisfying condition 3.
Condition 4 is satisfied since $pre(\Gamma) = pre(\Gamma')$.
Conditions 5 and 6 hold due to the definition of conservative approximation (Definition 2).
Conditions 7 and 8 are trivially satisfied, and condition 9 follows immediately since it held for $pool(i)$.

■

Property 2 (Properties of local monitor steps)**.** *Let $pool \in PSt$, $\sigma \in \Sigma$, and $\langle\!\langle pool_1, mem_1, \tau_1, gmon_1\rangle\!\rangle \in reach_\sigma(pool)$. If $pool_1(i) = [com, mc_1, mdst_1]$ for some $i \in pre(pool)$ and*
$$\langle [com_1, mc_1, mdst_1], mem_1, \tau_1\rangle \xrightarrow{\beta,\gamma,\delta}_\sigma \langle [com_2, mc_2, mdst_2], mem_2, \tau_2\rangle$$
where $mc_i = \langle \Gamma_i, lmdst_i, \overline{pc}_i, \overline{br}_i, time_i, term_i, block_i\rangle$ then

1) *For all variables x, if $time_1 \sqcup (\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1 \not\sqsubseteq \Gamma_2\langle x\rangle$ then $mem_1(x) = mem_2(x)$ and $\Gamma_1(x) = \Gamma_2(x)$. (i.e., $time_1 \sqcup (\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1$ is a lower bound on the memory side effects, and the local typing context cannot change. Note that we use the local typing environment after the step: Γ_2. This simplifies the statement of this lemma, since we get to ignore whether or not x floats.)*

2) *For all channels ch if $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \not\sqsubseteq ch$ then $\tau_1 \upharpoonright ch = \tau_2 \upharpoonright ch$ (i.e., $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1$ is a lower bound on the I/O effects.)*
3) *$\mathit{term}_1 \sqsubseteq \mathit{term}_2$ (i.e., termination level increases monotonically)*
4) *$\mathit{block}_1 \sqsubseteq \mathit{block}_2$ (i.e., blocking level increases monotonically)*
5) *If $\beta \neq \mathsf{sync}$ then $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1$ (i.e., the "time-inclusive pc" can go down only at synchronization steps)*
6) *If $\beta \neq \mathsf{sync}$ then $\mathit{time}_1 \sqsubseteq \mathit{time}_2$ (i.e., timing level increases monotonically, except at synchronizations)*
7) *If $\beta \neq \mathsf{sync}$ then $\mathit{lmdst}_1 = \mathit{lmdst}_2$ (i.e., mode state changes only at synchronizations)*
8) *If $(\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \neq \bot$ then $\beta \neq \mathsf{sync}$ and $\mathit{com}_2 \neq \mathsf{term}$ (i.e., non-bottom termination, blocking, and pc prevents synchronization or thread termination.)*
9) *If $\overline{pc}_1 \neq \epsilon$ then $\mathit{com}_2 \neq \mathsf{term}$ (i.e., non-empty pc stack prevents thread termination.)*

Proof: We consider the local monitor step associated with the local configuration step $\langle[\mathit{com}_1, \mathit{mc}_1, \mathit{mdst}_1], \mathit{mem}_1, \tau_1\rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle[\mathit{com}_2, \mathit{mc}_2, \mathit{mdst}_2], \mathit{mem}_2, \tau_2\rangle$. That is, we consider the step

$$\langle \mathit{\Gamma}_1, \mathit{lmdst}_1, \overline{pc}_1, \overline{br}_1, \mathit{time}_1, \mathit{term}_1, \mathit{block}_1\rangle \longrightarrow^{\epsilon,\alpha}_{\mathit{perm}} \langle \mathit{\Gamma}_2, \mathit{lmdst}_2, \overline{pc}_2, \overline{br}_2, \mathit{time}_2, \mathit{term}_2, \mathit{block}_2\rangle$$

- **M-Skip** Here $\mathit{\Gamma}_2 = \mathit{\Gamma}_1$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
  Note that by inversion on the command step, we also have that the memory is not changed and $\mathit{com}_2 \neq \mathsf{term}$. All of the conditions are satisfied trivially.
- **M-Assign1** Here $\mathit{\Gamma}_2 = \mathit{\Gamma}_1\langle x \mapsto_{\mathit{lmdst}_1} \ell\rangle$ where

  $$\ell = \mathit{\Gamma}\langle e\rangle \sqcup \mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1$$

  and

  $$\mathit{lmdst}_1 \rhd [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)]$$

  and $\ell \sqsubseteq \mathcal{L}(x)$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
  Note that by inversion on the command step, we have that for all variables y if $y \neq x$ then $\mathit{mem}_1(y) = \mathit{mem}_2(y)$ and $\mathit{\Gamma}_1(y) = \mathit{\Gamma}_2(y)$, and $\mathit{com}_2 \neq \mathsf{term}$.
  All of the conditions except condition 1 are satisfied trivially.
  Consider $\mathit{\Gamma}_2\langle x\rangle$. If $x \notin \mathit{pre}(\mathit{\Gamma}_2)$ then $\mathit{\Gamma}_2\langle x\rangle = \mathcal{L}(x)$, and, since $\ell \sqsubseteq \mathcal{L}(x)$ we have $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \mathit{\Gamma}_2\langle x\rangle$. If $x \in \mathit{pre}(\mathit{\Gamma}_2)$ then $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \ell \sqsubseteq \mathit{\Gamma}_2\langle x\rangle$. Thus, condition 1 is satisfied.
- **M-Assign2** Here $\mathit{\Gamma}_2 = \mathit{\Gamma}_1\langle x \mapsto_{\mathit{lmdst}_1} \ell\rangle$ where

  $$\ell = \mathit{\Gamma}_1\langle e\rangle \sqcup \mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1$$

  and

  $$\mathit{lmdst}_1 \rhd [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)]$$

  and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
  Note that by inversion on the command step, we have that for all variables y if $y \neq x$ then $\mathit{mem}_1(y) = \mathit{mem}_2(y)$ and $\mathit{\Gamma}_1(y) = \mathit{\Gamma}_2(y)$, and $\mathit{com}_2 \neq \mathsf{term}$.
  All of the conditions except condition 1 are satisfied trivially.
  Consider $\mathit{\Gamma}_2\langle x\rangle$. Since $\mathit{lmdst} \rhd [\mathsf{exclusiveread}(x)]$, we must have $x \in \mathit{pre}(\mathit{\Gamma}_2)$. Thus, $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \ell \sqsubseteq \mathit{\Gamma}_2\langle x\rangle$, and so condition 1 is satisfied.
- **M-Input1** Here $\mathit{\Gamma}_2 = \mathit{\Gamma}_1\langle x \mapsto_{\mathit{lmdst}_1} \ell\rangle$ where

  $$\ell = ch \sqcup \mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1$$

  and

  $$\mathit{lmdst}_1 \rhd [\mathsf{maywrite}(x), \mathsf{othersmightread}(x)]$$

  and $\ell \sqsubseteq \mathcal{L}(x)$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
  Note that by inversion on the command step, we have that for all variables y if $y \neq x$ then $\mathit{mem}_1(y) = \mathit{mem}_2(y)$ and $\mathit{\Gamma}_1(y) = \mathit{\Gamma}_2(y)$, and $\mathit{com}_2 \neq \mathsf{term}$, and the trace had an event for channel ch added.
  All of the conditions except conditions 1 and 2 are satisfied trivially.

Consider $\Gamma_2\langle x\rangle$. If $x \notin \mathit{pre}(\Gamma_2)$ then $\Gamma_2\langle x\rangle = \mathcal{L}(x)$, and, since $\ell \sqsubseteq \mathcal{L}(x)$ we have $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \Gamma_2\langle x\rangle$. If $x \in \mathit{pre}(\Gamma_2)$ then $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \ell \sqsubseteq \Gamma_2\langle x\rangle$. Thus, condition 1 is satisfied.
Also, $\mathit{time}\sqcup(\sqcup\overline{pc})\sqcup\mathit{term}\sqcup\mathit{block} \sqsubseteq \mathit{ch}$, and so condition 2 is satisfied.

- **M-Input2** Here $\Gamma_2 = \Gamma_1\langle x \mapsto_{\mathit{lmdst}_1} \ell\rangle$ where

$$\ell = \mathit{ch} \sqcup \mathit{time}_1\sqcup(\sqcup\overline{pc}_1)\sqcup\mathit{term}_1\sqcup\mathit{block}_1$$

and

$$\mathit{lmdst}_1 \triangleright [\mathsf{maywrite}(x), \mathsf{exclusiveread}(x)]$$

and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
Note that by inversion on the command step, we have that for all variables y if $y \neq x$ then $\mathit{mem}_1(y) = \mathit{mem}_2(y)$ and $\Gamma_1(y) = \Gamma_2(y)$, and $\mathit{com}_2 \neq \mathsf{term}$, and the trace had an event for channel ch added.
All of the conditions except conditions 1 and 2 are satisfied trivially.
Consider $\Gamma_2\langle x\rangle$. Since $\mathit{lmdst} \triangleright [\mathsf{exclusiveread}(x)]$, we must have $x \in \mathit{pre}(\Gamma_2)$. Thus, $\mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \sqsubseteq \ell \sqsubseteq \Gamma_2\langle x\rangle$, and so condition 1 is satisfied.
Also, $\mathit{time}\sqcup(\sqcup\overline{pc})\sqcup\mathit{term}\sqcup\mathit{block} \sqsubseteq \mathit{ch}$, and so condition 2 is satisfied.
- **M-Output** Here $\Gamma_2 = \Gamma_1$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
Note that by inversion on the command step, we have that the memory is unchanged, and $\mathit{com}_2 \neq \mathsf{term}$, and the trace had an event for channel ch added.
All of the conditions except condition 2 are satisfied trivially, and since $\mathit{time}\sqcup(\sqcup\overline{pc})\sqcup\mathit{term}\sqcup\mathit{block} \sqsubseteq \mathit{ch}$, condition 2 is also satisfied.
- **M-Term** Here $\Gamma_2 = \Gamma_1$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1 = \epsilon$ and $\overline{br}_2 = \overline{br}_1 = \epsilon$ and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1 = \bot$ and $\mathit{block}_2 = \mathit{block}_1 = \bot$ and, by inversion on the command step, $\mathit{com}_2 = \mathsf{term}$.
All conditions are trivially satisfied.
- **M-Barrier-Local** Here $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $\mathit{time}_2 = \bot$ and $\mathit{term}_2 = \mathit{term}_1 = \bot$ and $\mathit{block}_2 = \mathit{block}_1 = \bot$ and $(\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 = \bot$.
By inversion on the command step, we see that the memory is unchanged and so is the event trace. Moreover, $\beta = \mathsf{sync}$ and $\mathit{com}_2 \neq \mathsf{term}$. Thus, all conditions are satisfied.
- **M-Branch** Here $\Gamma_2 = \Gamma_1$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1{\cdot}\ell_{\mathsf{sb}}$ and $\overline{br}_2 = \overline{br}_1{\cdot}(\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where

$$(\ell_{\mathsf{sb}}, \mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ \mathit{com}_1\ \mathsf{else}\ \mathit{com}_2\ \mathsf{fi}, \Gamma_1, \mathit{lmdst}_1, \overline{pc}_1, \mathit{time}_1, \mathit{term}_1, \mathit{block}_1)$$

and $\mathit{time}_2 = \mathit{time}_1$ and $\mathit{term}_2 = \mathit{term}_1$ and $\mathit{block}_2 = \mathit{block}_1$.
By inversion on the command step, we see that the memory and event trace are unchanged, and that $\beta \neq \mathsf{sync}$ and $\mathit{com}_2 \neq \mathsf{term}$.
All conditions are thus trivially satisfied.
- **M-Join** Here $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2{\cdot}\ell_{\mathsf{sb}} = \overline{pc}_1$ and $\overline{br}_2{\cdot}(\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \overline{br}_1$ and $\mathit{time}_2 = \mathit{time}_1 \sqcup \mathit{time}_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}}$ and $\mathit{term}_2 = \mathit{term}_1 \sqcup \mathit{term}_{\mathsf{sb}}$ and $\mathit{block}_2 = \mathit{block}_1 \sqcup \mathit{block}_{\mathsf{sb}}$ and

$$\Gamma_2 = \lambda x.\begin{cases} \Gamma_1(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in \mathit{pre}(\Gamma_1) \cap \mathit{pre}(\Gamma_{\mathsf{sb}}) \\ \Gamma_1(x) & \text{if } x \in \mathit{pre}(\Gamma_1) \setminus \mathit{pre}(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

By inversion on the command step, we see that the memory and event trace are unchanged, and that $\beta \neq \mathsf{sync}$ and $\mathit{com}_2 \neq \mathsf{term}$. All conditions except condition 5 are trivially satisfied.
For condition 5, note that we have

$$\begin{aligned} \mathit{time}_1 \sqcup (\sqcup\overline{pc}_1) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 &= \mathit{time}_1 \sqcup (\sqcup\overline{pc}_2) \sqcup \ell_{\mathsf{sb}} \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \\ &\sqsubseteq \mathit{time}_1 \sqcup \ell_{\mathsf{sb}} \sqcup \mathit{time}_{\mathsf{sb}} \sqcup (\sqcup\overline{pc}_2) \sqcup \mathit{term}_1 \sqcup \mathit{block}_1 \\ &\sqsubseteq \mathit{time}_1 \sqcup \ell_{\mathsf{sb}} \sqcup \mathit{time}_{\mathsf{sb}} \sqcup (\sqcup\overline{pc}_2) \sqcup \mathit{term}_1 \sqcup \mathit{term}_{\mathsf{sb}} \sqcup \mathit{block}_1 \sqcup \mathit{block}_{\mathsf{sb}} \\ &= \mathit{time}_2 \sqcup (\sqcup\overline{pc}_2) \sqcup \mathit{term}_2 \sqcup \mathit{block}_2 \end{aligned}$$

Thus all conditions are satisfied.
- **M-Enter** $\Gamma_2 = \Gamma_1$ and $\mathit{lmdst}_2 = \mathit{lmdst}_1$ and $\overline{pc}_2 = \overline{pc}_1{\cdot}\ell_{\mathsf{sb}}$ and $\overline{br}_2 = \overline{br}_1{\cdot}(\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where

$$(\ell_{\mathsf{sb}}, \mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ \mathit{com}_2\ \mathsf{od}, \Gamma_1, \mathit{lmdst}_1, \overline{pc}_1, \mathit{time}_1, \mathit{term}_1, \mathit{block}_1)$$

and $time_2 = time_1$ and $term_2 = term_1$ and $block_2 = block_1$.
By inversion on the command step, we see that the memory and event trace are unchanged, and that $\beta \neq \mathsf{sync}$ and $com_2 \neq \mathsf{term}$.
All conditions are thus trivially satisfied.

- **M-More** Here $\Gamma_2 = \Gamma_1$ and $lmdst_2 = lmdst_1$ and $\overline{pc}_2 = \overline{pc}_1$ and $\overline{br}_2 = \overline{br}_1$ and $time_2 = time_1$ and $term_2 = term_1$ and $block_2 = block_1$.
By inversion on the command step, we see that the memory and event trace are unchanged, and that $\beta \neq \mathsf{sync}$ and $com_2 \neq \mathsf{term}$.
All conditions are thus trivially satisfied.
- **M-Leave** Here $lmdst_2 = lmdst_1$ and $\overline{pc}_2 \cdot \ell_{\mathsf{sb}} = \overline{pc}_1$ and $\overline{br}_2 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \overline{br}_1$ and $time_1 \sqcup time_{\mathsf{sb}} \sqsubseteq time_2$ and $term_2 = term_1 \sqcup term_{\mathsf{sb}}$ and $block_2 = block_1 \sqcup block_{\mathsf{sb}}$ and
$$\Gamma_2 = \lambda x. \begin{cases} \Gamma_1(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_1) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_1(x) & \text{if } x \in pre(\Gamma_1) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$
By inversion on the command step, we see that the memory and event trace are unchanged, and that $\beta \neq \mathsf{sync}$ and $com_2 \neq \mathsf{term}$.
All conditions are thus trivially satisfied.

■

C. ℓ-equivalence relations

We define ℓ-equivalence relations over local configurations and over global configurations. Intuitively, two (local, global) configurations are ℓ-equivalent if they agree on information that is at level ℓ or below.

Definition 3 (ℓ-equivalence of local configurations)**.** *Two local configurations*

$$\langle [com_1, mc_1, mdst_1], mem_1, \tau_1 \rangle \textit{ and } \langle [com_2, mc_2, mdst_2], mem_2, \tau_2 \rangle$$

are ℓ-equivalent if and only if all the following conditions hold. Assume that for $i = 1, 2$ we have

$$mc_i = \langle \Gamma_i, lmdst_i, \overline{pc}_i, \overline{br}_i, time_i, term_i, block_i \rangle$$

1) *$\tau_1 \downarrow \ell = \tau_2 \downarrow \ell$.*
2) *$lmdst_1 = lmdst_2$.*
3) *$mdst_1 = mdst_2$.*
4) *$pre(\Gamma_1) = pre(\Gamma_2)$.*
5) *For all variables x, if $x \in pre(\Gamma_1)$ and $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\Gamma_1\langle x \rangle \sqcup time_1 \sqcup (\sqcup \overline{pc}_1) \sqsubseteq \ell$ and $\Gamma_2\langle x \rangle \sqcup time_2 \sqcup (\sqcup \overline{pc}_2) \sqsubseteq \ell$ then $mem_1(x) = mem_2(x)$ and $\Gamma_1\langle x \rangle = \Gamma_2\langle x \rangle$.*
6) *For all variables x, if $x \notin pre(\Gamma_1)$ and $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\mathcal{L}(x) \sqcup time_1 \sqcup (\sqcup \overline{pc}_1) \sqsubseteq \ell$ and $\mathcal{L}(x) \sqcup time_2 \sqcup (\sqcup \overline{pc}_2) \sqsubseteq \ell$ then $mem_1(x) = mem_2(x)$.*
7) *If*
$$time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$
or
$$time_2 \sqcup (\sqcup \overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$
then $time_1 = time_2$ and $term_1 = term_2$ and $block_1 = block_2$ and $\overline{pc}_1 = \overline{pc}_2$ and $\overline{br}_1 = \overline{br}_2$ and $com_1 = com_2$ and for all x, $\Gamma_1\langle x \rangle = \Gamma_2\langle x \rangle$.
8) *For all $j \in 1..max(|\overline{pc}_1|, |\overline{pc}_2|)$, if*
$$time_1 \sqcup (\overline{pc}_1(0) \sqcup \ldots \sqcup \overline{pc}_1(j-1)) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$
or
$$time_2 \sqcup (\overline{pc}_2(0) \sqcup \ldots \sqcup \overline{pc}_2(j-1)) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$
then all of the following holds
   - *$|\overline{pc}_1| \geq j$.*
   - *$|\overline{pc}_2| \geq j$.*
   - *For all $k \in 0..j$ we have $\overline{pc}_1(k) = \overline{pc}_2(k)$ and $\overline{br}_1(k) = \overline{br}_2(k)$.*

- *There exists* $com_{cnt} \in \mathsf{Continuation_j}$ *such that:*
  - *Either* $com_1 = com_{cnt}$ *or* $com_1 = com'_1; com_{cnt}$ *for some* com'_1*; and*
  - *Either* $com_2 = com_{cnt}$ *or* $com_2 = com'_2; com_{cnt}$ *for some* com'_2*.*

  (This says that the pc level stack and the branch environments are identical up to and including the first high branch, and the two configurations agree on the low continuation.)

Note that the definition of of ℓ-equivalence of local configurations is symmetric. That is, if $lcnf_1$ and $lcnf_2$ are ℓ-equivalent, then $lcnf_2$ and $lcnf_1$ are ℓ-equivalent.

We now define ℓ-equivalence for global configurations, using ℓ-equivalence for local configurations as part of the definition.

Definition 4 (ℓ-equivalence of global configurations)**.** *Two global configurations* $\langle\!\langle pool_1, mem_1, \tau_1, gmon_1\rangle\!\rangle$ *and* $\langle\!\langle pool_2, mem_2, \tau_2, gmon_2\rangle\!\rangle$ *are ℓ-equivalent if and only if all the following conditions hold.*

- $pre(pool_1) = pre(pool_2)$.
- $\forall i \in pre(pool_1) : \langle pool_1(i), mem_1, \tau_1\rangle$ *is ℓ-equivalent to* $\langle pool_2(i), mem_2, \tau_2\rangle$.
- $\tau_1 \downarrow \ell = \tau_2 \downarrow \ell$.
- $gmon_1 = gmon_2$

D. ℓ-equivalent local configuration results

We define lemmas and theorems that describe the behavior of ℓ-equivalent local configurations when one configuration takes a step.

Lemma 1 (Low-equivalent expressions)**.** *Let*

$$lcnf_1 = \langle [com_1, \langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1\rangle, mdst_1], mem_1, \tau_1\rangle$$

and

$$lcnf_2 = \langle [com_2, \langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle, mdst_2], mem_2, \tau_2\rangle$$

be ℓ-equivalent local configurations.

For any expression e such that $lmdst_1 \triangleright \mathsf{mayread}(e)$*, if* $\Gamma_1\langle e\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ *and* $\Gamma_2\langle e\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$ *and* $e, mem_1 \Downarrow v_1$ *and* $e, mem_2 \Downarrow v_2$ *then* $v_1 = v_2$*.*

Proof: By induction on the structure of e. The interesting case is for a variable x. Since $lmdst_1 \triangleright \mathsf{mayread}(e)$ we have $lmdst_2 \triangleright \mathsf{mayread}(x)$. Since $\Gamma_1\langle e\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ we have $\Gamma_1\langle x\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and from $\Gamma_2\langle e\rangle \sqsubseteq \ell$ we have $\Gamma_2\langle x\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$. So by conditions 5 and 6 we have $mem_1(x) = mem_2(x)$ as required. ■

Theorem 7 (Thread Low Step)**.** *Let σ_1 and σ_2 be ℓ-equivalent strategies and let*

$$lcnf_1 = \langle [com_1, \langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1\rangle, mdst_1], mem_1, \tau_1\rangle$$

and

$$lcnf_2 = \langle [com_2, \langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle, mdst_2], mem_2, \tau_2\rangle$$

be ℓ-equivalent local configurations such that

$$time_1 \sqcup (\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$

and

$$time_1 \sqcup (\sqcup\overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$

and $lcnf_1 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} lcnf'_1$*.*

Then $lcnf_2 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} lcnf'_2$ *and* $lcnf'_1$ *and* $lcnf'_2$ *are ℓ-equivalent.*

Proof: From ℓ-equivalence of the two configurations (condition 7) we have that $term_1 = term_2$ and $block_1 = block_2$ and $\overline{pc}_1 = \overline{pc}_2$ and $\overline{br}_1 = \overline{br}_2$ and $com_1 = com_2$ and for all x, $\Gamma_1\langle x\rangle = \Gamma_2\langle x\rangle$.

Since $lcnf_1 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} lcnf'_1$, we have

$$(com_1, mem_1, \tau_1) \overset{\alpha,\epsilon}{\twoheadrightarrow}_{\sigma_1} (com'_1, mem'_1, \tau'_1)$$

and

$$\langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_1, lmdst'_1, \overline{pc}'_1, \overline{br}'_1, time'_1, term'_1, block'_1\rangle$$

We proceed by induction on the judgment $(com_1, mem_1, \tau_1) \overset{\alpha,\epsilon}{\twoheadrightarrow}_{\sigma_1} (com'_1, mem'_1, \tau'_1)$.

- $(x := e, mem, \tau) \overset{\mathsf{a}(x,e),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau)$
Here, $com_1 \equiv x := e$ and $com'_1 \equiv \mathsf{stop}$ and $mem'_1 = mem_1[x \mapsto v_1]$ and $\tau'_1 = \tau_1$ and $e, mem_1 \Downarrow v_1$.
By inversion on the local monitor rules, there are two possible rules that may apply: M-Assign1 and M-Assign2. In both of these cases, we have $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1$ and $\overline{br}_1 = \overline{br}'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$ and $\Gamma'_1 = \Gamma_1\langle x \mapsto_{lmdst_1} \ell'\rangle$ where $\ell' = \Gamma\langle e\rangle \sqcup time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1$.
We can construct
$$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$
and
$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
where $com_2 \equiv x := e$ and $com'_2 \equiv \mathsf{stop}$ and $mem'_2 = mem_2[x \mapsto v_2]$ and $\tau'_2 = \tau_2$ and $e, mem_2 \Downarrow v_2$ and $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$ and $\Gamma'_2 = \Gamma_2\langle x \mapsto_{lmdst_2} \ell'\rangle$ where $\ell' = \Gamma_2\langle e\rangle \sqcup time_2 \sqcup (\sqcup \overline{pc}_2) \sqcup term_2 \sqcup block_2$.
All of the conditions for ℓ-equivalence of $lcnf'_1$ and $lcnf'_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, except for conditions 5 and 6.
If $\Gamma_1\langle e\rangle \sqsubseteq \ell$ and $time_1 \sqsubseteq \ell$, then $\Gamma'_1\langle x\rangle \sqsubseteq \ell$ and $\Gamma'_1\langle x\rangle = \Gamma'_2\langle x\rangle$ and by Lemma 1 we have $v_1 = v_2$, so conditions 5 and 6 are satisfied.
If $\Gamma_1\langle e\rangle \not\sqsubseteq \ell$ or $time_1 \not\sqsubseteq \ell$, then $\Gamma'_1\langle x\rangle \not\sqsubseteq \ell$ and so conditions 5 and 6 are satisfied.
- $(\mathsf{skip}, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)$
Here, $com_1 \equiv \mathsf{skip}$ and $com'_1 \equiv \mathsf{stop}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-Skip must have been used, and we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1$ and $\overline{br}_1 = \overline{br}'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$.
We can construct
$$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$
and
$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
where $com_2 \equiv \mathsf{skip}$ and $com'_2 \equiv \mathsf{stop}$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$ and $\Gamma'_2 = \Gamma_2$.
The result follows trivially from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{stop}; com, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow_\sigma} (com, mem, \tau)$
Here, $com_1 \equiv \mathsf{stop}; com'_1$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-Skip must have been used, we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1$ and $\overline{br}_1 = \overline{br}'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$.
We can construct
$$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$
and
$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
where $com_2 \equiv \mathsf{stop}; com'_2$ and $mem'_2 = mem_1$ and $\tau'_2 = \tau_1$ and $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$ and $\Gamma'_2 = \Gamma_2$.
The result follows trivially from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(com_a; com_b, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow_\sigma} (com'_a; com_b, mem', \tau')$
Here we have $(com_a, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow_\sigma} (com'_a, mem', \tau')$ and the result holds by the inductive hypothesis.
- $(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, mem, \tau) \overset{\mathsf{b}(e,com_a,com_b),\epsilon}{\twoheadrightarrow_\sigma} (com_a; \mathsf{join}, mem, \tau)$
Here, $com_1 \equiv \mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}$ and $com'_1 \equiv com_a; \mathsf{join}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-Branch must have been used, and we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$. Also $\overline{pc}'_1 = \overline{pc}_1 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}'_1 = \overline{br}_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where
$$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, \Gamma_1, lmdst_1, \overline{pc}_1, time_1, term_1, block_1).$$
We can construct
$$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$

and

$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$

where $com_2 \equiv \mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}$ and either $com'_2 \equiv com_a;\mathsf{join}$ or $com'_2 \equiv com_b;\mathsf{join}$ (depending on the result of evaluation e) and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $\Gamma_2 = \Gamma'_2$ and $lmdst_2 = lmdst'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$. Also $\overline{pc}'_2 = \overline{pc}_2 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}'_2 = \overline{br}_2 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ since

$$\mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, \Gamma_1, lmdst_1, \overline{pc}_1, time_1, term_1, block_1) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, \Gamma_2, lmdst_2, \overline{pc}_2, time_2, term_2, block_2).$$

We consider two (mutually exclusive and exhaustive) cases, based on whether $\ell_{\mathsf{sb}} \sqsubseteq \ell$.

- $\ell_{\mathsf{sb}} \sqsubseteq \ell$. Since $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a conservative approximation to if e then com_a else com_b fi, etc., (Definition 2 and Lemma 14), we have that $\Gamma_1\langle e\rangle \sqcup time_1 \sqcup (\sqcup \overline{pc}_1) = \Gamma_2\langle e\rangle \sqcup time_2 \sqsubseteq \ell_{\mathsf{sb}} \sqsubseteq \ell$, so by Lemma 1 we have $e, mem_1 \Downarrow v$ and $e, mem_2 \Downarrow v$ for some v. That is, both executions take the same branch of the conditional, i.e., $com'_1 = com'_2 = com_a;\mathsf{join}$.
  The result follows from this fact plus the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$. Here, we may not have $com'_1 = com'_2$, since the two executions may differ on which branch of the conditional they will take. However, since the pc of the configuration is now "high" (i.e., $(\sqcup \overline{pc}'_1) \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}'_2) \not\sqsubseteq \ell$). The result follows easily from this and the ℓ-equivalence of $lcnf_1$ and $lcnf_2$, including Condition 8.

• $(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, mem, \tau) \overset{\mathsf{b}(e,com_a,com_b),\epsilon}{\twoheadrightarrow_\sigma} (com_b;\mathsf{join}, mem, \tau)$
This case is symmetric to the case above.

• $(\mathsf{join}, mem, \tau) \overset{\mathsf{join},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)$ Here, $com_1 \equiv \mathsf{join}$ and $com'_1 \equiv \mathsf{stop}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-Join must have been used, and we have $lmdst'_1 = lmdst_1$ and $\overline{pc}_1 = \overline{pc}'_1 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_1 = \overline{br}'_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time'_1 = time_1 \sqcup time_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}}$ and $term'_1 = term_1 \sqcup term_{\mathsf{sb}}$ and $block'_1 = block_1 \sqcup block_{\mathsf{sb}}$ and

$$\Gamma'_1 = \lambda x. \begin{cases} \Gamma_1(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_1) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_1(x) & \text{if } x \in pre(\Gamma_1) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Note that for all variables x we have $\Gamma_1(x) \sqsubseteq \Gamma'_1(x)$.
We can construct

$$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$

and

$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$

where $com_2 \equiv com_1 \equiv \mathsf{join}$ and $com'_2 \equiv com_1 \equiv \mathsf{stop}$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}_1 = \overline{pc}'_1 \cdot \ell_{\mathsf{sb}} = \overline{pc}'_2 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_2 = \overline{br}_1 = \overline{br}'_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \overline{br}'_2 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time'_2 = time_2 \sqcup time_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}} = time'_1$ and $term'_2 = term_2 \sqcup term_{\mathsf{sb}} = term'_1$ and $block'_2 = block_2 \sqcup block_{\mathsf{sb}} = block'_1$ and

$$\Gamma'_2 = \lambda x. \begin{cases} \Gamma_2(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_2) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_2(x) & \text{if } x \in pre(\Gamma_2) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

All of the conditions for ℓ-equivalence of $lcnf'_1$ and $lcnf'_2$ follow easily.

• $(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \overset{\mathsf{enter}(e,com),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau)$ Here, $com_1 \equiv \mathsf{while}\ e\ \mathsf{do}\ \ \mathsf{od}$ and $com'_1 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-Enter must have been used, and we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$. Also $\overline{pc}'_1 = \overline{pc}_1 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}'_1 = \overline{br}_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where

$$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma_1, lmdst_1, \overline{pc}_1, time_1, term_1, block_1).$$

We can construct

$$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$

and

$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$

where $com_2 \equiv com_1 \equiv \mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com'_2 \equiv com'_1 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $\Gamma_2 = \Gamma'_2$ and $lmdst_2 = lmdst'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$. Also $\overline{pc}'_2 = \overline{pc}_2 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}'_2 = \overline{br}_2 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ since

$$\mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma_1, lmdst_1, \overline{pc}_1, time_1, term_1, block_1) = \\ \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma_2, lmdst_2, \overline{pc}_2, time_2, term_2, block_2).$$

The result follows easily from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.

- $(\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \xrightarrow{\mathsf{more}(e,com),\epsilon}_{\sigma} (com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau)$
Here, $com_1 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com'_1 \equiv com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-More must have been used, and we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1$ and $\overline{br}_1 = \overline{br}'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$.
From Property 1, we have that for $k = |\overline{br}_1|$, the tuple $(\overline{pc}(k), time(k), term(k), block(k), \Gamma(k))$ (where $\overline{br}(k) = (time(k), term(k), block(k), \Gamma(k))$) is a conservative approximation (Definition 2) of the branching, timing, termination, and floating behavior for $\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}$ (and the local monitor state as at the time the while loop was entered).
Thus, by Definition 2, we have that $\Gamma_1\langle e\rangle \sqcup time_1 \sqsubseteq \overline{pc}(k) \sqsubseteq \ell$. So, from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, we have that $\Gamma_2\langle e\rangle \sqcup time_2 \sqsubseteq \overline{pc}(k) \sqsubseteq \ell$. Thus by Lemma 1 we have $e, mem_1 \Downarrow v$ and $e, mem_2 \Downarrow v$ for some v. That is, in both executions, the loop condition evaluates to true.
So we can construct
$$(com_2, mem_2, \tau_2) \xrightarrow{\alpha,\epsilon}_{\sigma_2} (com'_2, mem'_2, \tau'_2)$$
and
$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
where $com_2 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com'_2 \equiv com'_1 \equiv com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $\Gamma_2 = \Gamma'_2$ and $lmdst_2 = lmdst'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$.
The result follows easily from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \xrightarrow{\mathsf{leave}(e,com),\epsilon}_{\sigma} (\mathsf{stop}, mem, \tau)$
Here, $com_1 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com'_1 \equiv \mathsf{stop}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
By inversion on the local monitor step, rule M-Leave must have been used, and we have $lmdst'_1 = lmdst_1$ and $\overline{pc}_1 = \overline{pc}'_1 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_1 = \overline{br}'_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time'_1 = time_1 \sqcup time_{\mathsf{sb}}$ and $term'_1 = term_1 \sqcup term_{\mathsf{sb}}$ and $block'_1 = block_1 \sqcup block_{\mathsf{sb}}$ and
$$\Gamma'_1 = \lambda x. \begin{cases} \Gamma_1(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_1) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_1(x) & \text{if } x \in pre(\Gamma_1) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$
Note that for all variables x we have $\Gamma_1(x) \sqsubseteq \Gamma'_1(x)$.
From Property 1, we have that for $k = |\overline{br}_1|$, the tuple $(\overline{pc}(k), time(k), term(k), block(k), \Gamma(k))$ (where $\overline{br}(k) = (time(k), term(k), block(k), \Gamma(k))$) is a conservative approximation (Definition 2) of the branching, timing, termination, and floating behavior for $\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}$ (and the local monitor state as at the time the while loop was entered).
Thus, by Definition 2, we have that $\Gamma_1\langle e\rangle \sqcup time_1 \sqsubseteq \overline{pc}(k) \sqsubseteq \ell$. So, from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, we have that $\Gamma_2\langle e\rangle \sqcup time_2 \sqsubseteq \overline{pc}(k) \sqsubseteq \ell$. Thus by Lemma 1 we have $e, mem_1 \Downarrow v$ and $e, mem_2 \Downarrow v$ for some v. That is, in both executions, the loop condition evaluates to false.
So we can construct
$$(com_2, mem_2, \tau_2) \xrightarrow{\alpha,\epsilon}_{\sigma_2} (com'_2, mem'_2, \tau'_2)$$
and
$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
where $com_2 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com'_2 \equiv com'_1 \equiv \mathsf{stop}$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $lmdst'_2 = lmdst_2$ and $\overline{pc}_2 = \overline{pc}'_2 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_2 = \overline{br}'_2 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time'_2 = time_2 \sqcup time_{\mathsf{sb}}$ and $term'_2 = term_2 \sqcup term_{\mathsf{sb}}$ and $block'_2 = block_2 \sqcup block_{\mathsf{sb}}$ and
$$\Gamma'_2 = \lambda x. \begin{cases} \Gamma_2(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_2) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_2(x) & \text{if } x \in pre(\Gamma_2) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$
The result follows easily from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.

- $(\mathsf{input}\ ch\ \mathsf{to}\ x, mem, \tau) \overset{\mathsf{input}(x,ch,v),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau\cdot\mathsf{inp}(ch, v))$ Here, $com_1 \equiv \mathsf{input}\ ch\ \mathsf{to}\ x$ and $com'_1 \equiv \mathsf{stop}$ and $mem'_1 = mem_1[x \mapsto v_1]$ and $\tau'_1 = \tau_1\cdot\mathsf{inp}(ch, v_1)$ where $v_1 = \sigma_1(\tau_1, ch)$.
  By inversion on the local monitor rules, there are two possible rules that may apply: M-Input1 and M-Input2, (which differ on whether $lmdst_1 \triangleright \mathsf{exclusiveread}(x)$ or $lmdst_1 \triangleright \mathsf{othersmightread}(x)$). Regardless of which of the two monitor rules is used, we have $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1$ and $\overline{br}_1 = \overline{br}'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$ and $\Gamma'_1 = \Gamma\langle x \mapsto_{lmdst_1} \ell'\rangle$ where $\ell' = ch \sqcup \Gamma\langle e\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1$.
  We can construct
  $$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$
  and
  $$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
  where $com_2 \equiv \mathsf{input}\ ch\ \mathsf{to}\ x$ and $com'_2 \equiv \mathsf{stop}$ and $mem'_2 = mem_2[x \mapsto v_2]$ and $\tau'_2 = \tau_2\cdot\mathsf{inp}(ch, v_2)$ where $v_2 = \sigma_2(\tau_2, ch)$. Also, we have $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$ and $\Gamma'_2 = \Gamma_2\langle x \mapsto_{lmdst_2} \ell'\rangle$ where $\ell' = ch \sqcup \Gamma\langle e\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqcup term_2 \sqcup block_2$.
  All of the conditions for ℓ-equivalence of $lcnf'_1$ and $lcnf'_2$ follow easily from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, except for conditions 1 and 5 and 6. There are two (mutually exclusive and exhaustive) cases to consider.
  - If $ch \sqsubseteq \ell$ then $v_1 = v_2$ since σ_1 and σ_2 are ℓ-equivalent strategies so $v_1 = \sigma_1(\tau_1, ch) = \sigma_2(\tau_2, ch) = v_2$. So conditions 1, 5, and 6 are satisfied.
  - If $ch \not\sqsubseteq \ell$ then $\Gamma'_1\langle x\rangle \not\sqsubseteq \ell$ and so conditions 1, 5, and 6 are satisfied.
- $(\mathsf{output}\ e\ \mathsf{to}\ ch, mem, \tau) \overset{\mathsf{output}(ch,e,v),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau\cdot\mathsf{out}(ch, v))$
  Here, $com_1 \equiv \mathsf{output}\ e\ \mathsf{to}\ ch$ and $com'_1 \equiv \mathsf{stop}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1\cdot\mathsf{out}(ch, v_1)$ where $e, mem_1 \Downarrow v_1$.
  By inversion on the local monitor step, we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1$ and $\overline{br}_1 = \overline{br}'_1$ and $time_1 = time'_1$ and $term_1 = term'_1$ and $block_1 = block'_1$. We also know that $lmdst \triangleright \mathsf{mayread}(e)$ and $\Gamma_1\langle e\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq ch$
  We can construct
  $$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$
  and
  $$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
  where $\tau'_2 = \tau_2\cdot\mathsf{out}(ch, v_2)$ where $e, mem_2 \Downarrow v_2$ and $\Gamma_2 = \Gamma'_2$ and $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$ and $time_2 = time'_2$ and $term_2 = term'_2$ and $block_2 = block'_2$.
  All of the conditions for ℓ-equivalence of $lcnf'_1$ and $lcnf'_2$ follow easily from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, except for condition 1. There are two (mutually exclusive and exhaustive) cases to consider.
  - If $ch \sqsubseteq \ell$ then $\Gamma_1\langle e\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1 = \Gamma_2\langle e\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq ch \sqsubseteq \ell$, and so by Lemma 1 we have $v_1 = v_2$. Thus $\tau'_1 \!\downarrow \ell = \tau_1 \!\downarrow \ell\cdot\mathsf{out}(ch, v_1) = \tau_2 \!\downarrow \ell\cdot\mathsf{out}(ch, v_2)\tau'_2 \!\downarrow \ell$, and condition 1 holds.
  - If $ch \not\sqsubseteq \ell$ then $\tau'_1 \!\downarrow \ell = \tau_1 \!\downarrow \ell = \tau_2 \!\downarrow \ell \tau'_2 \!\downarrow \ell$, and condition 1 holds.
- $(//\gamma//\ \mathsf{barrier}, mem, \tau) \overset{\mathsf{sync},\gamma}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)$
  This case is impossible, since rule (M-Barrier-Local) is the only rule that could be used to allow the local monitor to take a step (due to the event sync), but this is not a synchronization event: $\mathsf{sync} \neq \epsilon$.
- $(\mathsf{stop}, mem, \tau) \overset{\mathsf{term},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{term}, mem, \tau)$
  Here, $com_1 \equiv \mathsf{stop}$ and $com'_1 \equiv \mathsf{term}$ and $mem'_1 = mem_1$ and $\tau'_1 = \tau_1$.
  By inversion on the local monitor step, rule M-Term must have been used, and we have $\Gamma_1 = \Gamma'_1$ and $lmdst_1 = lmdst'_1$ and $\overline{pc}_1 = \overline{pc}'_1 = \epsilon$ and $\overline{br}_1 = \overline{br}'_1 = \epsilon$ and $time_1 = time'_1$ and $term_1 = term'_1 = \bot$ and $block_1 = block'_1 = \bot$.
  We can construct
  $$(com_2, mem_2, \tau_2) \overset{\alpha,\epsilon}{\twoheadrightarrow_{\sigma_2}} (com'_2, mem'_2, \tau'_2)$$
  and
  $$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$
  where $com_2 \equiv \mathsf{stop}$ and $com'_2 \equiv \mathsf{term}$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $lmdst_2 = lmdst'_2$ and $\overline{pc}_2 = \overline{pc}'_2 = \epsilon$ and $\overline{br}_2 = \overline{br}'_2 = \epsilon$ and $time_2 = time'_2$ and $term_2 = term'_2 = \bot$ and $block_2 = block'_2 = \bot$ and $\Gamma'_2 = \Gamma_2$.
  The result follows trivially from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.

■

Theorem 8 (Thread High Step)**.** *Let σ_1 and σ_2 be ℓ-equivalent strategies and let*

$$lcnf_1 = \langle [com_1, \langle \mathit{\Gamma}_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1 \rangle, mdst_1], mem_1, \tau_1 \rangle$$

and

$$lcnf_2 = \langle [com_2, \langle \mathit{\Gamma}_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2 \rangle, mdst_2], mem_2, \tau_2 \rangle$$

be ℓ-equivalent local configurations such that

$$time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \not\sqsubseteq \ell$$

and $lcnf_1 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} lcnf_1'$ where

$$lcnf_1' = \langle [com_1', \langle \mathit{\Gamma}_1', lmdst_1', \overline{pc}_1', \overline{br}_1', time_1', term_1', block_1' \rangle, mdst_1'], mem_1', \tau_1' \rangle$$

and

$$time_1' \sqcup (\sqcup \overline{pc}_1') \sqcup term_1' \sqcup block_1' \not\sqsubseteq \ell.$$

Then $lcnf_1'$ and $lcnf_2$ are ℓ-equivalent.

Proof: Since $lcnf_1 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} lcnf_1'$, we have

$$(com_1, mem_1, \tau_1) \overset{\alpha,\epsilon}{\twoheadrightarrow}_{\sigma_1} (com_1', mem_1', \tau_1')$$

and

$$\langle \mathit{\Gamma}_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1 \rangle \longrightarrow^{\epsilon,\alpha}_{perm} \langle \mathit{\Gamma}_1', lmdst_1', \overline{pc}_1', \overline{br}_1', time_1', term_1', block_1' \rangle$$

We proceed by induction on the judgment $(com_1, mem_1, \tau_1) \overset{\alpha,\epsilon}{\twoheadrightarrow}_{\sigma_1} (com_1', mem_1', \tau_1')$.

- $(x := e, mem, \tau) \overset{\mathsf{a}(x,e),\epsilon}{\twoheadrightarrow}_{\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau)$
  Here, $com_1 \equiv x := e$ and $com_1' \equiv \mathsf{stop}$ and $mem_1' = mem_1[x \mapsto v]$ and $\tau_1' = \tau_1$ and $e, mem \Downarrow v$.
  By inversion on the local monitor rules, there are two possible rules that may apply: M-Assign1 and M-Assign2. In both of these cases, we have $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$, and $\mathit{\Gamma}_1' = \mathit{\Gamma}_1\langle x \mapsto_{lmdst} \ell' \rangle$ where $\ell' = \mathit{\Gamma}_1\langle e \rangle \sqcup time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1$. Note that $\ell' \not\sqsubseteq \ell$, since by assumption we have $time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \not\sqsubseteq \ell$.
  All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, except for conditions 5 and 6.
  We consider three (mutually exclusive and exhaustive) cases, based on variable x.
  - $x \notin pre(\mathit{\Gamma})_1$. Then we have $\mathit{\Gamma}_1' = \mathit{\Gamma}_1$, and conditions 5 and 6 follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
  - $x \in pre(\mathit{\Gamma})_1 \wedge lmdst_1 \rhd \mathsf{exclusivewrite}(x) \wedge x \in \mathit{FloatVar}$ then $\mathit{\Gamma}_1'\langle x \rangle = \ell'$. Since x is a floating variable, condition 6 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of $lcnf_1$ and $lcnf_2$. Since $\ell' \not\sqsubseteq \ell$, $\mathit{\Gamma}_1'\langle x \rangle \not\sqsubseteq \ell$, condition 5 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
  - $x \in pre(\mathit{\Gamma})_1 \wedge lmdst_1 \rhd \mathsf{othersmightwrite}(x) \wedge x \in \mathit{FloatVar}$ then $\mathit{\Gamma}_1'(x) = \ell' \sqcup \mathcal{L}(x)$. Since x is a floating variable, condition 6 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of $lcnf_1$ and $lcnf_2$. Since $\ell' \not\sqsubseteq \ell$, $\mathit{\Gamma}_1'\langle x \rangle = \ell' \sqcup \mathcal{L}(x) \not\sqsubseteq \ell$, condition 5 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{skip}, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow}_{\sigma} (\mathsf{stop}, mem, \tau)$
  Here, $com_1 \equiv \mathsf{skip}$ and $com_1' \equiv \mathsf{stop}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
  By inversion on the local monitor step, we have $\mathit{\Gamma}_1 = \mathit{\Gamma}_1'$ and $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$.
  The result follows trivially from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{stop}; com, mem, \tau) \overset{\mathsf{s},\epsilon}{\twoheadrightarrow}_{\sigma} (com, mem, \tau)$
  Here, $com_1 \equiv \mathsf{stop}; com_1'$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
  By inversion on the local monitor step, we have $\mathit{\Gamma}_1 = \mathit{\Gamma}_1'$ and $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$.
  The result follows trivially from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(com_a; com_b, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_{\sigma} (com_a'; com_b, mem', \tau')$
  Here we have $(com_a, mem, \tau) \overset{\alpha,\gamma}{\twoheadrightarrow}_{\sigma} (com_a', mem', \tau')$ and the result holds by the inductive hypothesis.
- $(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, mem, \tau) \overset{\mathsf{b}(e,com_a,com_b),\epsilon}{\twoheadrightarrow}_{\sigma} (com_a; \mathsf{join}, mem, \tau)$
  Here, $com_1 \equiv \mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}$ and $com_1' \equiv com_a; \mathsf{join}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.

By inversion on the local monitor step, rule M-Branch must have been used, and we have $\Gamma_1 = \Gamma_1'$ and $lmdst_1 = lmdst_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$. Also $\overline{pc}_1' = \overline{pc}_1 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_1' = \overline{br}_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where

$$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, \Gamma_1, lmdst_1, \overline{pc}_1, time_1, term_1, block_1).$$

All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.

- $(\mathsf{if}\ e\ \mathsf{then}\ com_a\ \mathsf{else}\ com_b\ \mathsf{fi}, mem, \tau) \xrightarrow{\mathsf{b}(e,com_a,com_b),\epsilon}_{\sigma} (com_b; \mathsf{join}, mem, \tau)$
This case is symmetric to the case above.
- $(\mathsf{join}, mem, \tau) \xrightarrow{\mathsf{join},\epsilon}_{\sigma} (\mathsf{stop}, mem, \tau)$ Here, $com_1 \equiv \mathsf{join}$ and $com_1' \equiv \mathsf{stop}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
By inversion on the local monitor step, rule M-Join must have been used, and we have $lmdst_1' = lmdst_1$ and $\overline{pc}_1 = \overline{pc}_1' \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_1 = \overline{br}_1' \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time_1' = time_1 \sqcup time_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}}$ and $term_1' = term_1 \sqcup term_{\mathsf{sb}}$ and $block_1' = block_1 \sqcup block_{\mathsf{sb}}$ and
$$\Gamma_1' = \lambda x. \begin{cases} \Gamma_1(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_1) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_1(x) & \text{if } x \in pre(\Gamma_1) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$
Note that for all variables x we have $\Gamma_1(x) \sqsubseteq \Gamma_1'(x)$.
All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \xrightarrow{\mathsf{enter}(e,com),\epsilon}_{\sigma} (\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau)$ Here, $com_1 \equiv \mathsf{while}\ e\ \mathsf{do}\ \ \mathsf{od}$ and $com_1' \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
By inversion on the local monitor step, rule M-Enter must have been used, and we have $\Gamma_1 = \Gamma_1'$ and $lmdst_1 = lmdst_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$. Also $\overline{pc}_1' = \overline{pc}_1 \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_1' = \overline{br}_1 \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where
$$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma_1, lmdst_1, \overline{pc}_1, time_1, term_1, block_1).$$
All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \xrightarrow{\mathsf{more}(e,com),\epsilon}_{\sigma} (com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau)$
Here, $com_1 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com_1' \equiv com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
By inversion on the local monitor step, rule M-More must have been used, and we have $\Gamma_1 = \Gamma_1'$ and $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$.
All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}, mem, \tau) \xrightarrow{\mathsf{leave}(e,com),\epsilon}_{\sigma} (\mathsf{stop}, mem, \tau)$
Here, $com_1 \equiv \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $com_1' \equiv \mathsf{stop}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
By inversion on the local monitor step, rule M-Leave must have been used, and we have $lmdst_1' = lmdst_1$ and $\overline{pc}_1 = \overline{pc}_1' \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_1 = \overline{br}_1' \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time_1' = time_1 \sqcup time_{\mathsf{sb}}$ and $term_1' = term_1 \sqcup term_{\mathsf{sb}}$ and $block_1' = block_1 \sqcup block_{\mathsf{sb}}$ and
$$\Gamma_1' = \lambda x. \begin{cases} \Gamma_1(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_1) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_1(x) & \text{if } x \in pre(\Gamma_1) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$
Note that for all variables x we have $\Gamma_1(x) \sqsubseteq \Gamma_1'(x)$.
All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- $(\mathsf{input}\ ch\ \mathsf{to}\ x, mem, \tau) \xrightarrow{\mathsf{input}(x,ch,v),\epsilon}_{\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau \cdot \mathsf{inp}(ch, v))$ Here, $com_1 \equiv \mathsf{input}\ ch\ \mathsf{to}\ x$ and $com_1' \equiv \mathsf{stop}$ and $mem_1' = mem_1[x \mapsto v]$ and $\tau_1' = \tau_1 \cdot \mathsf{inp}(ch, v)$.
There are two possible monitor rules that are applicable: (M-Input1) and (M-Input2) (which differ on whether $lmdst_1 \rhd \mathsf{exclusiveread}(x)$ or $lmdst_1 \rhd \mathsf{othersmightread}(x)$). Regardless of which of the two monitor rules is used, we have $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$ and $\Gamma_1' = \Gamma\langle x \mapsto_{lmdst} \ell' \rangle$ where $\ell' = ch \sqcup \Gamma_1\langle e \rangle \sqcup time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1$. Note that $\ell' \not\sqsubseteq \ell$, and since (from the premises of (M-Input1) and (M-Input2)) $time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq ch$, we also have $ch \not\sqsubseteq \ell$, and so $\tau_1 \downarrow \ell = \tau_1' \downarrow \ell$ (which satisfies condition 1 of ℓ-equivalence).
All of the conditions for ℓ-equivalence of $lcnf_1'$ and $lcnf_2$ follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$, except for conditions 5 and 6.
We consider three (mutually exclusive and exhaustive) cases, based on variable x.
  - $x \notin pre(\Gamma)_1$. Then we have $\Gamma_1' = \Gamma_1$, and conditions 5 and 6 follow trivially from ℓ-equivalence of $lcnf_1$ and $lcnf_2$.

- $x \in pre(\Gamma)_1 \wedge lmdst_1 \triangleright \mathsf{exclusivewrite}(x) \wedge x \in \mathit{FloatVar}$ then $\Gamma_1'\langle x\rangle = \ell'$. Since x is a floating variable, condition 6 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of lcnf_1 and lcnf_2. Since $\ell' \not\sqsubseteq \ell$, $\Gamma_1'\langle x\rangle \not\sqsubseteq \ell$, so condition 5 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of lcnf_1 and lcnf_2.
- $x \in pre(\Gamma)_1 \wedge lmdst_1 \triangleright \mathsf{othersmightwrite}(x) \wedge x \in \mathit{FloatVar}$ then $\Gamma_1'(x) = \ell' \sqcup \mathcal{L}(x)$. Since x is a floating variable, condition 6 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of lcnf_1 and lcnf_2. Since $\ell' \not\sqsubseteq \ell$, $\Gamma_1'\langle x\rangle = \ell' \sqcup \mathcal{L}(x) \not\sqsubseteq \ell$, so condition 5 is satisfied for x, and trivially satisfied for all other variables from ℓ-equivalence of lcnf_1 and lcnf_2.

- $(\mathsf{output}\ e\ \mathsf{to}\ ch, mem, \tau) \overset{\mathsf{output}(ch,e,v),\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau\cdot\mathsf{out}(ch,v))$
  Here, $com_1 \equiv \mathsf{output}\ e\ \mathsf{to}\ ch$ and $com_1' \equiv \mathsf{stop}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1\cdot\mathsf{out}(ch,v)$ where $e, mem_1 \Downarrow v$.
  By inversion on the local monitor step, we have $\Gamma_1 = \Gamma_1'$ and $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$.
  We also have (from the premises of (M-Output) that $time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block \sqsubseteq ch$, and thus $ch \not\sqsubseteq \ell$, and so $\tau_1 \downarrow \ell = \tau_1' \downarrow \ell$
  All other conditions for ℓ-equivalence of lcnf_1' and lcnf_2 follow trivially from ℓ-equivalence of lcnf_1 and lcnf_2.
- $(//\gamma//\ \mathsf{barrier}, mem, \tau) \overset{\mathsf{sync},\gamma}{\twoheadrightarrow_\sigma} (\mathsf{stop}, mem, \tau)$
  This case is impossible, since rule (M-Barrier-Local) is the only rule that could be used to allow the local monitor to take a step (due to the event sync), but this is not a synchronization event: $\mathsf{sync} \neq \epsilon$.
- $(\mathsf{stop}, mem, \tau) \overset{\mathsf{term},\epsilon}{\twoheadrightarrow_\sigma} (\mathsf{term}, mem, \tau)$
  Here, $com_1 \equiv \mathsf{stop}$ and $com_1' \equiv \mathsf{term}$ and $mem_1' = mem_1$ and $\tau_1' = \tau_1$.
  By inversion on the local monitor step, we have $\Gamma_1 = \Gamma_1'$ and $lmdst_1 = lmdst_1'$ and $\overline{pc}_1 = \overline{pc}_1'$ and $\overline{br}_1 = \overline{br}_1'$ and $time_1 = time_1'$ and $term_1 = term_1'$ and $block_1 = block_1'$.
  The result follows trivially from the ℓ-equivalence of lcnf_1 and lcnf_2.

■

Note that by the properties of local monitor steps (Property 2, condition 5), we have that the time inclusive PC ($time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block$) increases monotonically except at synchronization steps. This means that given a sequence of local configuration steps (i.e., the execution of a thread), once we apply the high-step theorem (Theorem 8) we cannot apply the low-step theorem (Theorem 7) until the thread reaches a barrier.

Theorem 9 (Thread barrier step)**.** *Let σ_1 and σ_2 be ℓ-equivalent strategies and let*

$$\mathit{lcnf}_1 = \langle [com_1, \langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1\rangle, mdst_1], mem_1, \tau_1\rangle$$

and

$$\mathit{lcnf}_2 = \langle [com_2, \langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle, mdst_2], mem_2, \tau_2\rangle$$

be ℓ-equivalent local configurations such that $\mathit{lcnf}_1 \xrightarrow{\mathsf{sync},\gamma,\delta}_{\sigma_1} \mathit{lcnf}_1'$ and lcnf_2 is at a barrier (i.e., the next command to reduce is a barrier command), and moreover, the following two conditions hold.

1) *If $(\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$ or $(\sqcup\overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$ then $time_1 = time_2$ and $term_1 = term_2$ and $block_1 = block_2$ and $\overline{pc}_1 = \overline{pc}_2$ and $\overline{br}_1 = \overline{br}_2$ and $com_1 = com_2$ and for all x, $\Gamma_1\langle x\rangle = \Gamma_2\langle x\rangle$.*
2) *For all $j \in 1..max(|\overline{pc}_1|, |\overline{pc}_2|)$, if*

$$(\overline{pc}_1(0) \sqcup \ldots \sqcup \overline{pc}_1(j-1)) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$

or

$$(\overline{pc}_2(0) \sqcup \ldots \sqcup \overline{pc}_2(j-1)) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$

then all of the following holds
   - *$|\overline{pc}_1| \geq j$.*
   - *$|\overline{pc}_2| \geq j$.*
   - *For all $k \in 0..j$ we have $\overline{pc}_1(k) = \overline{pc}_2(k)$ and $\overline{br}_1(k) = \overline{br}_2(k)$.*
   - *There exists $com_{cnt} \in \mathsf{Continuation_j}$ such that:*
     - *Either $com_1 = com_{cnt}$ or $com_1 = com_1'; com_{cnt}$ for some com_1'; and*
     - *Either $com_2 = com_{cnt}$ or $com_2 = com_2'; com_{cnt}$ for some com_2'.*

Then

$$\mathit{lcnf}_2 \xrightarrow{\mathsf{sync},\gamma,\delta}_{\sigma_2} \mathit{lcnf}_2'$$

and lcnf_1' and lcnf_2' are ℓ-equivalent.

Proof:

Since $lcnf_1 \xrightarrow{\mathsf{sync},\gamma,\delta}_{\sigma_1} lcnf'_1$, the monitor rule (M-Barrier-Local) must have been used, so we have $(\sqcup\overline{pc}_1) \sqcup term_1 \sqcup block_1 = \bot$ and $lmdst'_1 = update(lmdst_1, \delta)$ and $pre(\Gamma'_1) = \{x \mid x \in FloatVar \wedge (lmdst'_1 \triangleright \mathsf{exclusiveread}(x) \vee lmdst'_1 \triangleright \mathsf{exclusivewrite}(x))\}$ and $(lmdst_1 \triangleright \mathsf{exclusiveread}(x) \wedge lmdst'_1 \triangleright \mathsf{othersmightread}(x)) \implies \Gamma_1(x) \sqsubseteq \mathcal{L}(x)$ and $lmdst'_1 \triangleright \mathsf{exclusivewrite}(x) \implies \Gamma'_1(x) = \Gamma_1\langle x\rangle$ and $(lmdst_1 \triangleright \mathsf{othersmightwrite}(x) \wedge lmdst'_1 \triangleright \mathsf{exclusiveread}(x)) \implies \Gamma'_1(x) = \Gamma_1\langle x\rangle$ and $(lmdst_1 \triangleright \mathsf{exclusivewrite}(x) \wedge lmdst'_1 \triangleright [\mathsf{exclusiveread}(x), \mathsf{othersmightwrite}(x)]) \implies \Gamma'_1(x) = \Gamma_1(x) \sqcup \mathcal{L}(x)$.

Thus, we have $time_1 = time_2$ and $term_1 = term_2$ and $block_1 = block_2$ and $\overline{pc}_1 = \overline{pc}_2$ and $\overline{br}_1 = \overline{br}_2$ and $com_1 = com_2$ and for all x, $\Gamma_1\langle x\rangle = \Gamma_2\langle x\rangle$.

We can therefore construct

$$(com_2, mem_2, \tau_2) \overset{\mathsf{sync},\gamma}{\twoheadrightarrow}_{\sigma_2} (com'_2, mem'_2, \tau'_2)$$

and

$$\langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle \longrightarrow^{\delta,\mathsf{sync}}_{perm} \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle$$

where $com_2 \equiv com_1$ and $com'_2 \equiv com'_1$ and $mem'_2 = mem_2$ and $\tau'_2 = \tau_2$ and $\overline{pc}_2 = \overline{pc}'_2$ and $\overline{br}_2 = \overline{br}'_2$ and $time_2 = time'_2$ and $term_2 = term'_2 = \bot$ and $block_2 = block'_2 = \bot$ and $lmdst'_2 = update(lmdst_2, \delta)$ and $\Gamma'_2 = \Gamma'_1$.

By setting $\Gamma'_2 = \Gamma'_1$, we ensure that the preconditions for (M-Barrier-Local) are satisfied, namely: $pre(\Gamma'_2) = \{x \mid x \in FloatVar \wedge (lmdst'_2 \triangleright \mathsf{exclusiveread}(x) \vee lmdst'_2 \triangleright \mathsf{exclusivewrite}(x))\}$ and $(lmdst_2 \triangleright \mathsf{exclusiveread}(x) \wedge lmdst'_2 \triangleright \mathsf{othersmightread}(x)) \implies \Gamma_2(x) \sqsubseteq \mathcal{L}(x)$ and $lmdst'_2 \triangleright \mathsf{exclusivewrite}(x) \implies \Gamma'_2(x) = \Gamma_2\langle x\rangle$ and $(lmdst_2 \triangleright \mathsf{othersmightwrite}(x) \wedge lmdst'_2 \triangleright \mathsf{exclusiveread}(x)) \implies \Gamma'_2(x) = \Gamma_2\langle x\rangle$ and $(lmdst_2 \triangleright \mathsf{exclusivewrite}(x) \wedge lmdst'_2 \triangleright [\mathsf{exclusiveread}(x), \mathsf{othersmightwrite}(x)]) \implies \Gamma'_2(x) = \Gamma_2(x) \sqcup \mathcal{L}(x)$.

We consider each condition of ℓ-equivalence of local configurations (Definition 3) to ensure that they all hold.

- Condition 1 ($\tau'_1 {\downarrow} \ell = \tau'_2 {\downarrow} \ell$) holds trivially from the ℓ-equivalence of $lcnf_1$ and $lcnf_2$.
- Condition 2 ($lmdst'_1 = lmdst'_2$) holds because $lmdst'_1 = update(lmdst_1, \delta) = update(lmdst_2, \delta) = lmdst'_2$.
- Condition 3 ($mdst'_1 = mdst'_2$) holds because $lmdst'_1 = lmdst'_2$ and, by Property 1, $LMmst'_1 = mdst'_1$ and $LMmst'_2 = mdst'_2$.
- Condition 4 ($pre(\Gamma'_1) = pre(\Gamma'_2)$) holds because

$$\begin{aligned} pre(\Gamma'_1) &= \{x \mid x \in FloatVar \wedge (lmdst'_1 \triangleright \mathsf{exclusiveread}(x) \vee lmdst'_1 \triangleright \mathsf{exclusivewrite}(x))\} \\ &= \{x \mid x \in FloatVar \wedge (lmdst'_2 \triangleright \mathsf{exclusiveread}(x) \vee lmdst'_2 \triangleright \mathsf{exclusivewrite}(x))\} \\ &= pre(\Gamma'_2) \end{aligned}$$

- Condition 5 requires that for all variables x, if $x \in pre(\Gamma'_1)$ and $lmdst'_1 \triangleright \mathsf{mayread}(x)$ and $\Gamma'_1\langle x\rangle \sqcup time'_1 \sqcup (\sqcup\overline{pc}'_1) \sqsubseteq \ell$ and $\Gamma'_2\langle x\rangle \sqcup time'_2 \sqcup (\sqcup\overline{pc}'_2) \sqsubseteq \ell$ then $mem'_1(x) = mem'_2(x)$ and $\Gamma'_1\langle x\rangle = \Gamma'_2\langle x\rangle$.
  Let $x \in pre(\Gamma'_1)$ and $lmdst'_1 \triangleright \mathsf{mayread}(x)$ and $\Gamma'_1\langle x\rangle \sqcup time'_1 \sqcup (\sqcup\overline{pc}'_1) \sqsubseteq \ell$ and $\Gamma'_2\langle x\rangle \sqcup time'_2 \sqcup (\sqcup\overline{pc}'_2) \sqsubseteq \ell$.
  The preconditions for (M-Barrier-Local) ensure that $\Gamma_1\langle x\rangle \sqsubseteq \Gamma'_1\langle x\rangle$ and $\Gamma_2\langle x\rangle \sqsubseteq \Gamma'_2\langle x\rangle$. Thus we have $\Gamma_1\langle x\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\Gamma_2\langle x\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$. So by ℓ-equivalence of $lcnf_1$ and $lcnf_2$, we have $mem'_1(x) = mem_1(x) = mem_2(x) = mem'_2(x)$.
  Since $\Gamma'_1 = \Gamma'_2$ we have $\Gamma'_1\langle x\rangle = \Gamma'_2\langle x\rangle$.
- Condition 6 requires that for all variables x, if $x \notin pre(\Gamma_1)$ and $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\mathcal{L}(x) \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\mathcal{L}(x) \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$ then $mem_1(x) = mem_2(x)$.
  Let $x \notin pre(\Gamma_1)$ and $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\mathcal{L}(x) \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\mathcal{L}(x) \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$.
  The preconditions for (M-Barrier-Local) ensure that $\Gamma_1\langle x\rangle \sqsubseteq \Gamma'_1\langle x\rangle$ and $\Gamma_2\langle x\rangle \sqsubseteq \Gamma'_2\langle x\rangle$. Thus we have $\Gamma_1\langle x\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\Gamma_2\langle x\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$. So by ℓ-equivalence of $lcnf_1$ and $lcnf_2$, we have $mem'_1(x) = mem_1(x) = mem_2(x) = mem'_2(x)$.
- Condition 7 requires that if

$$time'_1 \sqcup (\sqcup\overline{pc}'_1) \sqcup term'_1 \sqcup block'_1 \sqsubseteq \ell$$

  or

$$time'_2 \sqcup (\sqcup\overline{pc}'_2) \sqcup term'_2 \sqcup block'_2 \sqsubseteq \ell$$

  then $time'_1 = time'_2$ and $term'_1 = term'_2$ and $block'_1 = block'_2$ and $\overline{pc}'_1 = \overline{pc}'_2$ and $\overline{br}'_1 = \overline{br}'_2$ and $com'_1 = com'_2$ and for all x, $\Gamma'_1\langle x\rangle = \Gamma'_2\langle x\rangle$.
  This follows immediately from ℓ-equivalence of $lcnf_1$ and $lcnf_2$ and since $\Gamma'_1 = \Gamma'_2$.
- Condition 8 follows immediately by assumption.

■

We define a useful lemma that says that if we have two ℓ-equivalent local configurations, and one of them takes a step, and modifies variable x, which it thinks is level ℓ or below, and the other local configuration takes zero or one steps to a ℓ-equivalent configuration, then the value of x will be the same in the two final configurations.

Lemma 2. *Let σ_1 and σ_2 be ℓ-equivalent strategies and let*

$$lcnf_1 = \langle [com_1, \langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1 \rangle, mdst_1], mem_1, \tau_1 \rangle$$

and

$$lcnf_2 = \langle [com_2, \langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2 \rangle, mdst_2], mem_2, \tau_2 \rangle$$

be ℓ-equivalent local configurations such that $lcnf_1 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} lcnf'_1$ where

$$lcnf'_1 = \langle [com'_1, \langle \Gamma'_1, lmdst'_1, \overline{pc}'_1, \overline{br}'_1, time'_1, term'_1, block'_1 \rangle, mdst'_1], mem'_1, \tau'_1 \rangle$$

and either

$$lcnf_2 = lcnf'_2$$

or

$$lcnf_2 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} lcnf'_2$$

and

$$lcnf'_2 = \langle [com'_2, \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2 \rangle, mdst'_2], mem'_2, \tau'_2 \rangle$$

and $lcnf'_1$ and $lcnf'_2$ are ℓ-equivalent. (That is, $lcnf_2$ takes zero or one steps to reach local configuration $lcnf'_2$ that is ℓ-equivalent to $lcnf'_1$.)

For any variable x, if $mem_1(x) \neq mem'_1(x)$ and $\Gamma'_1\langle x \rangle \sqsubseteq \ell$ then

$$lcnf_2 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} lcnf'_2$$

and $mem'_1(x) = mem'_2(x)$.

Proof: Let x be a variable such that $mem_1(x) \neq mem'_1(x)$ and $\Gamma'_1\langle x \rangle \sqsubseteq \ell$. By Property 2, we have

$$time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \Gamma'_1\langle x \rangle \sqsubseteq \ell$$

By ℓ-equivalence of $lcnf_1$ and $lcnf_2$ (Definition 3) we have

$$time_2 \sqcup (\sqcup \overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$

By inspection of the operational semantics for local configurations, we see that the only way the value of variable x can change is by an assignment to x or an input to x from some channel. These correspond to local monitor rules (M-Assign1), (M-Assign2), (M-Input1), and (M-Input2). In all cases, we have that $time_1 = time'_1$, $\overline{pc}_1 = \overline{pc}'_1$, $term_1 = term'_1$, and $block_1 = block'_1$, and so

$$time'_1 \sqcup (\sqcup \overline{pc}'_1) \sqcup term'_1 \sqcup block'_1 \sqsubseteq \ell.$$

By ℓ-equivalence of $lcnf_1$ and $lcnf_2$ (Definition 3, condition 7) we have $com_1 = com_2$ and $com'_1 = com'_2$, and so $lcnf_2 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} lcnf'_2$.

Also, by inspection of rules (M-Assign1), (M-Assign2), (M-Input1), and (M-Input2), we see that $\Gamma_1\langle e \rangle \sqsubseteq \Gamma'_1\langle x \rangle \sqsubseteq \ell$ and $mdst_1 \triangleright \mathsf{mayread}(e)$. Moreover, by condition 7 of Definition 3 applied to the variables in e, we have $\Gamma_2\langle e \rangle \sqsubseteq \ell$. Therefore by ℓ-equivalence of $lcnf_1$ and $lcnf_2$ and Lemma 1, we have $e, mem_1 \Downarrow v$ and $e, mem_2 \Downarrow v$ where $v = mem'_1(x)$, and thus $mem'_1(x) = mem'_2(x)$ as required. ■

Theorem 10 (Preservation of local equivalence)**.** *Let $\langle\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\rangle$ and $\langle\langle pool_2, mem_2, \tau_2, gmon_2 \rangle\rangle$ be ℓ-equivalent global configurations, and let σ_1 and σ_2 be ℓ-equivalent strategies. Assume*

$$\langle pool_1(i), mem_1, \tau_1 \rangle \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} \langle thread'_{1,i}, mem'_1, \tau'_1 \rangle$$

and we have $thread'_{2,i}$, mem'_2, and $\tau'_{2,i}$ such that $\langle thread'_{1_i}, mem'_1, \tau'_1 \rangle$ and $\langle thread'_{2,i}, mem'_2, \tau'_2 \rangle$ are ℓ-equivalent. and either $thread'_{2,i} = pool_2(i)$, $mem'_2 = mem_2$, and $\tau'_{2,i} = \tau_2$ or

$$\langle pool_2(i), mem_2, \tau_2 \rangle \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} \langle thread'_{2,i}, mem'_2, \tau'_{2,i} \rangle.$$

Then for all $j \in pre(pool_1) \setminus \{i\}$ we have $\langle pool_1(j), mem'_1, \tau'_1 \rangle$ and $\langle pool_2(j), mem'_2, \tau'_2 \rangle$ are ℓ-equivalent.

Proof: First note that $\tau'_1 \downarrow \ell = \tau'_2 \downarrow \ell$, since $\langle thread'_{1_i}, mem'_1, \tau'_1 \rangle$ and $\langle thread'_{2,i}, mem'_2, \tau'_2 \rangle$ are ℓ-equivalent.

Let $j \in pre(pool_1) \setminus \{i\}$. By ℓ-equivalence of $\langle\!\langle pool_1, mem_1, \tau_1, gmon_1\rangle\!\rangle$ and $\langle\!\langle pool_2, mem_2, \tau_2, gmon_2\rangle\!\rangle$ we have that local configurations $\langle pool_1(j), mem_1, \tau_1\rangle$ and $\langle pool_2(j), mem_2, \tau_2\rangle$ are ℓ-equivalent, and we need to show that $\langle pool_1(j), mem'_1, \tau'_1\rangle$ and $\langle pool_2(j), mem'_2, \tau'_2\rangle$ are ℓ-equivalent (Definition 3).

Let

$$pool_1(j) = [com_1, \langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1\rangle, mdst_1]$$

and

$$pool_2(j) = [com_2, \langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle, mdst_2]$$

and

$$pool'_1(i) = [com^i_1, \langle \Gamma^i_1, lmdst^i_1, \overline{pc}^i_1, \overline{br}^i_1, time^i_1, term^i_1, block^i_1\rangle, mdst^i_1]$$

and

$$pool'_2(i) = [com^i_2, \langle \Gamma^i_2, lmdst^i_2, \overline{pc}^i_2, \overline{br}^i_2, time^i_2, term^i_2, block^i_2\rangle, mdst^i_2]$$

Since τ'_1 and τ'_2 are ℓ-equivalent, and $\langle pool_1(j), mem_1, \tau_1\rangle$ and $\langle pool_2(j), mem_2, \tau_2\rangle$ are ℓ-equivalent, we only need to check that conditions of local-configuration-ℓ-equivalence that involve the memory are still satisfied. Specifically, we need to show conditions 5 and 6 hold.

- Condition 5 requires that if $x \in pre(\Gamma_1)$ and $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\Gamma_1\langle x\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\Gamma_2\langle x\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$ then $mem_1(x) = mem_2(x)$ and $\Gamma_1\langle x\rangle = \Gamma_2\langle x\rangle$.
- Condition 6 requires that for all variables x, if $x \notin pre(\Gamma_1)$ and $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\mathcal{L}(x) \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\mathcal{L}(x) \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$ then $mem_1(x) = mem_2(x)$.

We will address both of these cases at once. Let x be a variable such that $lmdst_1 \triangleright \mathsf{mayread}(x)$ and $\Gamma_1\langle x\rangle \sqcup time_1 \sqcup (\sqcup\overline{pc}_1) \sqsubseteq \ell$ and $\Gamma_2\langle x\rangle \sqcup time_2 \sqcup (\sqcup\overline{pc}_2) \sqsubseteq \ell$.

If it is not the case that $lmdst^i_1 \triangleright \mathsf{maywrite}(x)$ (and thus, not the case that $lmdst^i_1 \triangleright \mathsf{maywrite}(x)$), then the ith threads could not have written to x, and so, $mem_1(x) = mem'_1(x)$ and $mem_2(x) = mem'_2(x)$. Moreover, since $\langle pool_1(j), mem_1, \tau_1\rangle$ and $\langle pool_2(j), mem_2, \tau_2\rangle$ are ℓ-equivalent, we have $mem_1(x) = mem_2(x)$, and thus $mem'_1(x) = mem'_2(x)$ as required.

Otherwise, it is the case that $lmdst^i_1 \triangleright \mathsf{maywrite}(x)$ (and thus, $lmdst^i_1 \triangleright \mathsf{maywrite}(x)$). From the sound use of mode states, this means that $lmdst_1 \triangleright \mathsf{othersmightwrite}(x)$ and $lmdst^i_1 \triangleright \mathsf{othersmightread}(x)$. By Property 1, we have $\mathcal{L}(x) \sqsubseteq \Gamma_1\langle x\rangle$ and $\Gamma^i_1\langle x\rangle \sqsubseteq \mathcal{L}(x)$. Thus, we have $\Gamma^i_1\langle x\rangle \sqsubseteq \ell$. If thread i did not write x, then by a similar argument above, we have $mem'_1(x) = mem'_2(x)$ as required. If thread i did write x, then by Lemma 2, we have $mem'_1(x) = mem'_2(x)$ as required. ■

E. ℓ-equivalent global configuration results

We present two of the theorems related to showing that given two ℓ-equivalent global configurations, if one of them takes a step, then the other can take zero or more steps to a ℓ-equivalent global configuration. We actually require more than just two ℓ-equivalent global configurations: we require that those two global configurations were produced by executions that started from the same initial pool state, and that these two executions are "well aligned", i.e., the global configurations in the executions can be lined up into ℓ-equivalent pairs.

Definition 5. *Let $gcnf_1$ and $gcnf_2$ be two ℓ-equivalent global configurations, and let σ_1 and σ_2 be two ℓ-equivalent strategies. We say that $gcnf_1$ and $gcnf_2$ are* well aligned *for σ_1 and σ_2 (or, simply,* well aligned *if the strategies are clear from context) if there exists an initial pool state $pool_0$, k_1 and k_2 such that*

$$gcnf^0_1 = gcnf^0_2 = \langle\!\langle pool_0, mem_{init}, \tau_{init}, gmon_{init,|pre(pool_0)|}\rangle\!\rangle$$

and

$$\textit{for all } i \in 0..(k_1 - 1).\ gcnf^i_1 \twoheadrightarrow_{\sigma_1} gcnf^{i+1}_1$$

and

$$gcnf_1 = gcnf^{k_1}_1$$

and

$$\textit{for all } j \in 0..(k_2 - 1).\ gcnf^j_2 \twoheadrightarrow_{\sigma_2} gcnf^{j+1}_2$$

and

$$gcnf_2 = gcnf^{k_2}_2$$

and there is a relation $R \subseteq \{0, \ldots, k_1\} \times \{0, \ldots, k_2\}$ such that all of the following hold.

- *if $(i, j) \in R$ then $gcnf^i_1$ is ℓ-equivalent to $gcnf^j_2$.*
- $(0, 0) \in R$.

- *for all* $i \in \{0, \ldots, k_1\}$ *there exists* $j \in \{0, \ldots, k_2\}$ *such that* $(i, j) \in R$.
- *for all* $j \in \{0, \ldots, k_2\}$ *there exists* $i \in \{0, \ldots, k_1\}$ *such that* $(i, j) \in R$.
- *for all* $(i, j) \in R$ *and* $(i', j') \in R$*, if* $i < i'$ *then* $j \leq j'$*; and, symmetrically, if* $j < j'$ *then* $i \leq i'$.

Note that this definition implies that $\mathit{gcnf}_1 \in \mathit{reach}_{\sigma_1}(\mathit{pool}_0)$ and $\mathit{gcnf}_2 \in \mathit{reach}_{\sigma_2}(\mathit{pool}_0)$ but also requires that the executions to reach these global configurations can be lined up appropriately into ℓ-equivalent configurations. The relation R is a *correspondence* [37], which shows how the configurations in the two executions line up.

The proof of the soundness of our monitor relies on being given an execution using strategy σ_1 and constructing a well-aligned execution from σ_2, where σ_1 and σ_2 are ℓ-equivalent strategies. This essentially implies that an observer of the channel ℓ cannot distinguish the two executions.

Theorem 11 (Non-barrier global low step)**.** *Let* gcnf_1 *and* gcnf_2 *be* ℓ*-equivalent global configurations, and let* σ_1 *and* σ_2 *be* ℓ*-equivalent strategies. Suppose that* gcnf_1 *and* gcnf_2 *are well aligned. Suppose* $\mathit{gcnf}_1 \twoheadrightarrow_{\sigma_1} \mathit{gcnf}_1'$ *by the ith thread taking a step using the first rule of Figure 2, where*

$$\mathit{time}_i \sqcup (\sqcup \overline{\mathit{pc}}_i) \sqcup \mathit{term}_i \sqcup \mathit{block}_i \sqsubseteq \ell$$

and

$$\mathit{pool}_1(i) = [\mathit{com}_i, \langle \mathit{\Gamma}_i, \mathit{lmdst}_i, \overline{\mathit{pc}}_i, \overline{\mathit{br}}_i, \mathit{time}_i, \mathit{term}_i, \mathit{block}_i \rangle, \mathit{mdst}_i]$$

and

$$\mathit{gcnf}_1 = \langle\!\langle \mathit{pool}_1, \mathit{mem}_1, \tau_1, \mathit{gmon}_1 \rangle\!\rangle.$$

Then there exists gcnf_2' *such that* $\mathit{gcnf}_2 \twoheadrightarrow_{\sigma_2} \mathit{gcnf}_2'$ *and* gcnf_1' *and* gcnf_2' *are* ℓ*-equivalent and well aligned.*

Proof: Since gcnf_1 and $\mathit{gcnf}_2 = \langle\!\langle \mathit{pool}_2, \mathit{mem}_2, \tau_2, \mathit{gmon}_2 \rangle\!\rangle$ are ℓ-equivalent, we have that local configurations $\langle \mathit{pool}_1(i), \mathit{mem}_1, \tau_1 \rangle$ and $\langle \mathit{pool}_2(i), \mathit{mem}_2, \tau_2 \rangle$ are ℓ-equivalent. The result follows immediately by Theorem 7 and Theorem 10. ■

Theorem 12 (Non-barrier global high step)**.** *Let* gcnf_1 *and* gcnf_2 *be* ℓ*-equivalent global configurations, and let* σ_1 *and* σ_2 *be* ℓ*-equivalent strategies. Suppose that* gcnf_1 *and* gcnf_2 *are well aligned. Suppose* $\mathit{gcnf}_1 \twoheadrightarrow_{\sigma_1} \mathit{gcnf}_1'$ *by the ith thread taking a step using the first rule of Figure 2, where*

$$\mathit{time}_i \sqcup (\sqcup \overline{\mathit{pc}}_i) \sqcup \mathit{term}_i \sqcup \mathit{block}_i \not\sqsubseteq \ell$$

and

$$\mathit{pool}_1(i) = [\mathit{com}_i, \langle \mathit{\Gamma}_i, \mathit{lmdst}_i, \overline{\mathit{pc}}_i, \overline{\mathit{br}}_i, \mathit{time}_i, \mathit{term}_i, \mathit{block}_i \rangle, \mathit{mdst}_i]$$

and

$$\mathit{gcnf}_1 = \langle\!\langle \mathit{pool}_1, \mathit{mem}_1, \tau_1, \mathit{gmon}_1 \rangle\!\rangle.$$

Then gcnf_1' *and* gcnf_2 *are* ℓ*-equivalent.*

Proof: Since gcnf_1 and $\mathit{gcnf}_2 = \langle\!\langle \mathit{pool}_2, \mathit{mem}_2, \tau_2, \mathit{gmon}_2 \rangle\!\rangle$ are ℓ-equivalent, we have that local configurations $\langle \mathit{pool}_1(i), \mathit{mem}_1, \tau_1 \rangle$ and $\langle \mathit{pool}_2(i), \mathit{mem}_2, \tau_2 \rangle$ are ℓ-equivalent.

Let

$$\mathit{pool}_1'(i) = [\mathit{com}_i', \langle \mathit{\Gamma}_i', \mathit{lmdst}_i', \overline{\mathit{pc}}_i', \overline{\mathit{br}}_i', \mathit{time}_i', \mathit{term}_i', \mathit{block}_i' \rangle, \mathit{mdst}_i']$$

where

$$\mathit{gcnf}_1' = \langle\!\langle \mathit{pool}_1', \mathit{mem}_1', \tau_1', \mathit{gmon}_1' \rangle\!\rangle.$$

By Property 2, since this is not a synchronization step, we have that

$$\mathit{time}_i \sqcup (\sqcup \overline{\mathit{pc}}_i) \sqcup \mathit{term}_i \sqcup \mathit{block}_i \sqsubseteq \mathit{time}_i' \sqcup (\sqcup \overline{\mathit{pc}}_i') \sqcup \mathit{term}_i' \sqcup \mathit{block}_i'$$

and so

$$\mathit{time}_i' \sqcup (\sqcup \overline{\mathit{pc}}_i') \sqcup \mathit{term}_i' \sqcup \mathit{block}_i' \not\sqsubseteq \ell$$

The result follows immediately by Theorem 8 and Theorem 10. ■

F. Global barrier results

The global barrier theorem (Theorem 13) states, essentially, that if we have two ℓ-equivalent global configurations, and one of them performs a barrier synchronization, then the other global configuration can also reach the global barrier, and successfully perform the synchronization.

In order to prove this, we first define a notion of low equivalence, very similar to ℓ-equivalence of local configurations, but ignoring the timing level. We call this *time-insensitive-ℓ-equivalence*. Some key differences from ℓ-equivalence include: the strong guarantees apply when the pc excluding the timing is low; we don't care about equality of memory; we require that the time-inclusive pc is high (simplifies some of the cases).

Definition 6 (Time-insensitive-ℓ-equivalence of thread states)**.** *Two thread states*

$$[com_1, mc_1, mdst_1] \text{ and } [com_2, mc_2, mdst_2]$$

are time-insensitive-ℓ-equivalent if and only if all the following conditions hold. Assume that for $i = 1, 2$ we have

$$mc_i = \langle \Gamma_i, lmdst_i, \overline{pc}_i, \overline{br}_i, time_i, term_i, block_i \rangle$$

1) $time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \not\sqsubseteq \ell$ *and* $time_2 \sqcup (\sqcup \overline{pc}_2) \sqcup term_2 \sqcup block_2 \not\sqsubseteq \ell$.
2) *If*

$$(\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$

or

$$(\sqcup \overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$

then $time_1 = time_2$ *and* $term_1 = term_2$ *and* $block_1 = block_2$ *and* $\overline{pc}_1 = \overline{pc}_2$ *and* $\overline{br}_1 = \overline{br}_2$ *and* $com_1 = com_2$ *and for all x,* $\Gamma_1\langle x \rangle = \Gamma_2 \langle x \rangle$.
3) *For all $j \in 1..max(|\overline{pc}_1|, |\overline{pc}_2|)$, if*

$$(\overline{pc}_1(0) \sqcup \ldots \sqcup \overline{pc}_1(j-1)) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$

or

$$(\overline{pc}_2(0) \sqcup \ldots \sqcup \overline{pc}_2(j-1)) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$

then all of the following holds
   - $|\overline{pc}_1| \geq j$.
   - $|\overline{pc}_2| \geq j$.
   - *For all $k \in 0..j$ we have $\overline{pc}_1(k) = \overline{pc}_2(k)$ and $\overline{br}_1(k) = \overline{br}_2(k)$.*
   - *There exists $com_{cnt} \in \mathsf{Continuation_j}$ such that:*
     - *Either $com_1 = com_{cnt}$ or $com_1 = com'_1; com_{cnt}$ for some com'_1; and*
     - *Either $com_2 = com_{cnt}$ or $com_2 = com'_2; com_{cnt}$ for some com'_2.*

   (This says that the pc level stack and the branch environments are identical up to and including the first high branch, and the two configurations agree on the low continuation.)

We use this definition of time-insensitive-ℓ-equivalence to define the following key lemma for proving the global barrier theorem.

Lemma 3 (Time-insensitive-ℓ-equivalent execution)**.** *Let $gcnf_1 = \langle\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\rangle$ and $gcnf_2 = \langle\langle pool_2, mem_2, \tau_2, gmon_2 \rangle\rangle$ be ℓ-equivalent global configurations, and let σ_1 and σ_2 be ℓ-equivalent strategies. Suppose that $gcnf_1$ and $gcnf_2$ are well aligned.*

Let $gcnf'_1 = \langle\langle pool'_1, mem'_1, \tau'_1, gmon'_1 \rangle\rangle$.

Let $gcnf_1 \twoheadrightarrow^*_{\sigma_1} gcnf'_1$ without executing a synchronization step.

Suppose $i \in pre(pool_1)$ and $pool_1(i)$ and $pool_2(i)$ are time-insensitive-ℓ-equivalent, and

$$pool'_1(i) = [com'_i, \langle \Gamma'_i, lmdst'_i, \overline{pc}'_i, \overline{br}'_i, time'_i, term'_i, block'_i \rangle, mdst'_i]$$

where

$$(\sqcup \overline{pc}'_i) \sqcup term'_i \sqcup block'_i \sqsubseteq \ell.$$

Then for all (fair) executions from $gcnf_2$, there exists a $gcnf'_2 = \langle\langle pool'_2, mem'_2, \tau'_2, gmon'_2 \rangle\rangle$ in the execution (reachable without going through a synchronization step) such that $pool'_1(i)$ and $pool'_2(i)$ are time-insensitive-ℓ-equivalent.

In order to prove Lemma 3, we first state and prove several supporting lemmas (Lemma 4, Lemma 5, and Lemma 6), which are analogous to, respectively, a high-step lemma, a low-step lemma, and a high-to-low-step lemma, but for time-insensitive-ℓ-equivalence instead of ℓ-equivalence. Another key difference is that while the ℓ-equivalence theorems simply needed to find *some* steps that satisfied the requirements, these lemmas must hold for all fair executions. This is because at this point in the proof, the second execution has already been chosen (by the ℓ-equivalence theorems) and we need to show that these executions satisfy some additional properties.

Lemma 4 (Time-insensitive-ℓ-equivalence high step)**.** *Let*

$$gcnf_1 = \langle\!\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\!\rangle$$

and

$$gcnf_2 = \langle\!\langle pool_2, mem_2, \tau_2, gmon_2 \rangle\!\rangle$$

be ℓ-equivalent global configurations, and let σ_1 and σ_2 be ℓ-equivalent strategies. Suppose that $gcnf_1$ and $gcnf_2$ are well aligned. $i \in pre(pool_1)$ and assume $pool_1(i)$ and $pool_2(i)$ are time-insensitive-ℓ-equivalent.

Let $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf_1'$ be a non-synchronization step. Let $gcnf_1' = \langle\!\langle pool_1', mem_1', \tau_1', gmon_1' \rangle\!\rangle$ and

$$pool_1(i) = [com_{1,i}, \langle \Gamma_{1,i}, lmdst_{1,i}, \overline{pc}_{1,i}, \overline{br}_{1,i}, time_{1,i}, term_{1,i}, block_{1,i} \rangle, mdst_{1,i}]$$

and

$$pool_1'(i) = [com_{1,i}', \langle \Gamma_{1,i}', lmdst_{1,i}', \overline{pc}_{1,i}', \overline{br}_{1,i}', time_{1,i}', term_{1,i}', block_{1,i}' \rangle, mdst_{1,i}']$$

If $pool_1(i) \neq pool_1'(i)$ and

$$(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$$

and

$$(\sqcup \overline{pc}_{1,i}') \sqcup term_{1,i}' \sqcup block_{1,i}' \not\sqsubseteq \ell$$

then $pool_1'(i)$ is time-insensitive-ℓ-equivalent to $pool_2(i)$.

Proof: First note that $(\sqcup \overline{pc}_{2,i}) \sqcup term_{2,i} \sqcup block_{2,i} \not\sqsubseteq \ell$, since $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$ and $pool_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2(i)$,

We show that each condition holds.

- Condition 1. Since $(\sqcup \overline{pc}_{1,i}') \sqcup term_{1,i}' \sqcup block_{1,i}' \not\sqsubseteq \ell$ we have $time_{1,i}' \sqcup (\sqcup \overline{pc}_{1,i}') \sqcup term_{1,i}' \sqcup block_{1,i}' \not\sqsubseteq \ell$. Also, $time_{2,i} \sqcup (\sqcup \overline{pc}_{2,i}) \sqcup term_{2,i} \sqcup block_{2,i} \not\sqsubseteq \ell$ since $pool_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2(i)$.
- Condition 2. We have $(\sqcup \overline{pc}_{2,i}) \sqcup term_{2,i} \sqcup block_{2,i} \not\sqsubseteq \ell$. Also, by assumption we have $(\sqcup \overline{pc}_{1,i}') \sqcup term_{1,i}' \sqcup block_{1,i}' \not\sqsubseteq \ell$. Thus the precondition is false, and this condition is trivially satisfied.
- Condition 3. By cases on the monitor rule used in the derivation of judgment

$$\langle pool_1(i), mem_1, \tau_1 \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma_1} \langle pool'(i), mem_1', \tau_1' \rangle$$

  - (M-Skip), (M-Assign1), (M-Assign2), (M-Input1), (M-Input2), (M-Output), and (M-Term). In all of these rules, most of the monitor state remains the same ($\overline{pc}_1 = \overline{pc}_1'$, $\overline{br}_1 = \overline{br}_1'$, $term_1 = term_1'$, and $block_1 = block_1'$), which ensures that most of the requirements of Condition 3 are met. Note also that none of these commands reduce a join or a more, and so if there previous existed a com_{cnt} that satisfied the requirements, there will continue to exist such a com_{cnt} that satisfies the requirements.
  - (M-Branch), (M-Enter). These commands increase the pc and branch environment stack by one. But since $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$, the precondition is not satisfied for this new branch.
  - (M-Join), (M-Leave). These commands pop an element off the pc and branch environment stack, and increase the termination and blocking levels. Thus, the required conditions follow from the time-insensitive-ℓ-equivalence of $pool_1(i)$ and $pool_2(i)$.
  - (M-Barrier-Local) is impossible, since it was a non-synchronization step.

■

Lemma 5 (Time-insensitive-ℓ-equivalence low step)**.** *Let $gcnf_1 = \langle\!\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\!\rangle$ be a global configuration and let $gcnf_2^0$, $gcnf_2^1$, $gcnf_2^2$, $gcnf_2^3$, … be a (finite or infinite) sequence of global configurations such that for all $j < n$ we have $gcnf_2^j \twoheadrightarrow_{\sigma_2} gcnf_2^{j+1}$ (without taking a synchronization step), and the execution is fair (i.e., for any thread that can take a non-synchronization step, it eventually does) and $gcnf_1$ is ℓ-equivalent to $gcnf_2^j$ for all $gcnf_2^j$ in the sequence. Moreover, suppose that $gcnf_1$ and $gcnf_2^0$ are well aligned.*

Let $gcnf_2^j = \langle\!\langle pool_2^j, mem_2^j, \tau_2^j, gmon_2^j \rangle\!\rangle$.

Let $i \in pre(pool_1)$ and assume $pool_1(i)$ and $pool_2^0(i)$ are time-insensitive-ℓ-equivalent.

Let $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf'_1$ be a non-synchronization step. Let $gcnf'_1 = \langle\!\langle pool'_1, mem'_1, \tau'_1, gmon'_1 \rangle\!\rangle$ and

$$pool_1(i) = [com_{1,i}, \langle \mathit{\Gamma}_{1,i}, lmdst_{1,i}, \overline{pc}_{1,i}, \overline{br}_{1,i}, time_{1,i}, term_{1,i}, block_{1,i} \rangle, mdst_{1,i}]$$

and

$$pool'_1(i) = [com'_{1,i}, \langle \mathit{\Gamma}'_{1,i}, lmdst'_{1,i}, \overline{pc}'_{1,i}, \overline{br}'_{1,i}, time'_{1,i}, term'_{1,i}, block'_{1,i} \rangle, mdst'_{1,i}]$$

If $pool_1(i) \neq pool'_1(i)$ and

$$(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \sqsubseteq \ell$$

then there exists a k such that for all $j \in 1..k$, $pool_2^j(i) = pool_2^0(i)$ and $pool_2^k(i) \neq pool_2^{k+1}(i)$ and $pool'_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2^{k+1}(i)$ (i.e., the next step that thread i takes is to an equivalent thread state).

Proof: Assume $pool_1(i) \neq pool'_1(i)$ and $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \sqsubseteq \ell$.

Since $pool_1(i)$ and $pool_2^0(i)$ are time-insensitive-ℓ-equivalent, by Condition 2 of Definition 6 we have $time_{1,i} = time_{2,i}^k$ and $term_{1,i} = term_{2,i}^k$ and $block_{1,i} = block_{2,i}^k$ and $\overline{pc}_{1,i} = \overline{pc}_{2,i}^k$ and $\overline{br}_{1,i} = \overline{br}_{2,i}^k$ and $com_{1,i} = com_{2,i}^k$ and for all x, $\mathit{\Gamma}_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^k\langle x \rangle$.

Moreover, since the step $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf'_1$ was not a synchronization step, we have

$$\langle pool_1(i), mem_1, \tau_1 \rangle \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} \langle pool'_1(i), mem'_1, \tau'_1 \rangle$$

and thus, by inversion on the inference rule in Figure 1,

$$(com_{1,i}, mem_1, \tau_1) \overset{\alpha,\epsilon}{\rightarrowtriangle}_{\sigma_1} (com'_{1,i}, mem'_1, \tau'_1)$$

and

$$\langle \mathit{\Gamma}_{1,i}, lmdst_{1,i}, \overline{pc}_{1,i}, \overline{br}_{1,i}, time_{1,i}, term_{1,i}, block_{1,i} \rangle \longrightarrow_{perm}^{\epsilon,\alpha} \langle \mathit{\Gamma}'_{1,i}, lmdst'_{1,i}, \overline{pc}'_{1,i}, \overline{br}'_{1,i}, time'_{1,i}, term'_{1,i}, block'_{1,i} \rangle$$

Since the execution from $gcnf_2^0$ is fair, the ith thread must be scheduled eventually. Assume that this occurs after k steps, i.e., $pool_2^k(i) \neq pool_2^{k+1}(i)$.

We proceed by induction on the command $com_{2,i}^k$. We show that we can construct a derivation for

$$\langle pool_2^k(i), mem_2^k, \tau_2^k \rangle \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} \langle pool_2^{k+1}(i), mem_2^{k+1}, \tau_2^{k+1} \rangle$$

such that $pool'_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2^{k+1}(i)$

- $com_{2,i}^k = \mathsf{skip}$. Here, $com_{2,i}^{k+1} = com'_{1,i} = \mathsf{stop}$ and $time_{1,i} = time_{2,i}^k = time'_{1,i} = time_{2,i}^{k+1}$ and $term_{1,i} = term_{2,i}^k = term'_{1,i} = term_{2,i}^{k+1}$ and $block_{1,i} = block_{2,i}^k = block'_{1,i} = block_{2,i}^{k+1}$ and $\overline{pc}_{1,i} = \overline{pc}_{2,i}^k = \overline{pc}'_{1,i} = \overline{pc}_{2,i}^{k+1}$ and $\overline{br}_{1,i} = \overline{br}_{2,i}^k = \overline{br}'_{1,i} = \overline{br}_{2,i}^{k+1}$ and for all x, $\mathit{\Gamma}_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^k\langle x \rangle = \mathit{\Gamma}'_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^{k+1}\langle x \rangle$.
  The result follows trivially.
- $com_{2,i}^k = \mathsf{stop}; com$. Here, $com_{2,i}^{k+1} = com'_{1,i} = com$ and $time_{1,i} = time_{2,i}^k = time'_{1,i} = time_{2,i}^{k+1}$ and $term_{1,i} = term_{2,i}^k = term'_{1,i} = term_{2,i}^{k+1}$ and $block_{1,i} = block_{2,i}^k = block'_{1,i} = block_{2,i}^{k+1}$ and $\overline{pc}_{1,i} = \overline{pc}_{2,i}^k = \overline{pc}'_{1,i} = \overline{pc}_{2,i}^{k+1}$ and $\overline{br}_{1,i} = \overline{br}_{2,i}^k = \overline{br}'_{1,i} = \overline{br}_{2,i}^{k+1}$ and for all x, $\mathit{\Gamma}_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^k\langle x \rangle = \mathit{\Gamma}'_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^{k+1}\langle x \rangle$.
  The result follows trivially.
- $com_{2,i}^k = \mathsf{stop}$. Here, $com_{2,i}^{k+1} = com'_{1,i} = \mathsf{term}$ and $time_{1,i} = time_{2,i}^k = time'_{1,i} = time_{2,i}^{k+1}$ and $term_{1,i} = term_{2,i}^k = term'_{1,i} = term_{2,i}^{k+1} = \bot$ and $block_{1,i} = block_{2,i}^k = block'_{1,i} = block_{2,i}^{k+1} = \bot$ and $\overline{pc}_{1,i} = \overline{pc}_{2,i}^k = \overline{pc}'_{1,i} = \overline{pc}_{2,i}^{k+1} = \epsilon$ and $\overline{br}_{1,i} = \overline{br}_{2,i}^k = \overline{br}'_{1,i} = \overline{br}_{2,i}^{k+1} = \epsilon$ and for all x, $\mathit{\Gamma}_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^k\langle x \rangle = \mathit{\Gamma}'_{1,i}\langle x \rangle = \mathit{\Gamma}_{2,i}^{k+1}\langle x \rangle$.
  The result follows trivially.
- $com_{2,i}^k = x := e$.
  Here, $com_{2,i}^{k+1} = com'_{1,i} = \mathsf{stop}$ and $time_{1,i} = time_{2,i}^k = time'_{1,i} = time_{2,i}^{k+1}$ and $term_{1,i} = term_{2,i}^k = term'_{1,i} = term_{2,i}^{k+1}$ and $block_{1,i} = block_{2,i}^k = block'_{1,i} = block_{2,i}^{k+1}$ and $\overline{pc}_{1,i} = \overline{pc}_{2,i}^k = \overline{pc}'_{1,i} = \overline{pc}_{2,i}^{k+1}$ and $\overline{br}_{1,i} = \overline{br}_{2,i}^k = \overline{br}'_{1,i} = \overline{br}_{2,i}^{k+1}$.
  For the typing environment, there are two cases to consider. If $lmdst_{1,i} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)]$ then $\ell = \mathit{\Gamma}_{1,i}\langle e \rangle \sqcup time_{1,i} \sqcup (\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i}$ and $\ell \sqsubseteq \mathcal{L}(x)$ and $\mathit{\Gamma}'_{1,i} = \mathit{\Gamma}_{1,i}\langle x \mapsto_{lmdst_{1,i}} \ell \rangle$. But in this case, since $lmdst_{1,i} = lmdst_{2,i}^k$, and from the other equalities, we have $\mathit{\Gamma}_{2,i}^{k+1} = \mathit{\Gamma}_{2,i}^k\langle x \mapsto_{lmdst_{2,i}} \ell \rangle$, and the result follows.
  The other case is that $lmdst_{1,i} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)]$ and so $\ell = \mathit{\Gamma}_{1,i}\langle e \rangle \sqcup time_{1,i} \sqcup (\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i}$ and $\mathit{\Gamma}'_{1,i} = \mathit{\Gamma}_{1,i}\langle x \mapsto_{lmdst_{1,i}} \ell \rangle$. Here again, since $lmdst_{1,i} = lmdst_{2,i}^k$, and from the other equalities, we have $\mathit{\Gamma}_{2,i}^{k+1} = \mathit{\Gamma}_{2,i}^k\langle x \mapsto_{lmdst_{2,i}} \ell \rangle$, and the result follows.
- $com_{2,i}^k = \mathsf{input}\ ch\ \mathsf{to}\ x$. This case is nearly identical to the assignment case above.

- $com^k_{2,i} = \mathsf{output}\ e\ \mathsf{to}\ ch$.
  Here, $com^{k+1}_{2,i} = com'_{1,i} = \mathsf{stop}$ and $time_{1,i} = time^k_{2,i} = time'_{1,i} = time^{k+1}_{2,i}$ and $term_{1,i} = term^k_{2,i} = term'_{1,i} = term^{k+1}_{2,i}$ and $block_{1,i} = block^k_{2,i} = block'_{1,i} = block^{k+1}_{2,i}$ and $\overline{pc}_{1,i} = \overline{pc}^k_{2,i} = \overline{pc}'_{1,i} = \overline{pc}^{k+1}_{2,i}$ and $\overline{br}_{1,i} = \overline{br}^k_{2,i} = \overline{br}'_{1,i} = \overline{br}^{k+1}_{2,i}$ and for all x, $\Gamma_{1,i}\langle x\rangle = \Gamma^k_{2,i}\langle x\rangle = \Gamma'_{1,i}\langle x\rangle = \Gamma^{k+1}_{2,i}\langle x\rangle$.
  Moreover, we have $lmdst_{1,i} \triangleright \mathsf{mayread}(e)$ and $\Gamma_{1,i}\langle e\rangle \sqcup time_{1,i} \sqcup (\sqcup\overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \sqsubseteq ch$. But since $lmdst_{1,i} = lmdst^k_{2,i}$, and from the other equalities, we have $lmdst^k_{2,i} \triangleright \mathsf{mayread}(e)$ and $\Gamma^k_{2,i}\langle e\rangle \sqcup time^k_{2,i} \sqcup (\sqcup\overline{pc}^k_{2,i}) \sqcup term^k_{2,i} \sqcup block^k_{2,i} \sqsubseteq ch$.
  We can thus construct an appropriate derivation, and the result follows easily.
- $com^k_{2,i} = \mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}$. Here, $com^{k+1}_{2,i} = com'_{1,i} = \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ and $time_{1,i} = time^k_{2,i} = time'_{1,i} = time^{k+1}_{2,i}$ and $term_{1,i} = term^k_{2,i} = term'_{1,i} = term^{k+1}_{2,i}$ and $block_{1,i} = block^k_{2,i} = block'_{1,i} = block^{k+1}_{2,i}$ and for all x, $\Gamma_{1,i}\langle x\rangle = \Gamma^k_{2,i}\langle x\rangle = \Gamma'_{1,i}\langle x\rangle = \Gamma^{k+1}_{2,i}\langle x\rangle$.
  Also, we have $lmdst_{1,i} \triangleright \mathsf{mayread}(e)$ and
  $$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma_{1,i}, lmdst_{1,i}, \overline{pc}_{1,i}, time_{1,i}, term_{1,i}, block_{1,i})$$
  and $\overline{pc}'_{1,i} = \overline{pc}_{1,i}\cdot\ell_{\mathsf{sb}}$ and $\overline{br}'_{1,i} = \overline{br}_{1,i}\cdot(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.
  Since $lmdst_{1,i} = lmdst^k_{2,i}$, and from the other equalities, we have $lmdst^k_{2,i} \triangleright \mathsf{mayread}(e)$ and $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma^k_{2,i}, lmdst^k_{2,i}, \overline{pc}^k_{2,i}, time^k_{2,i}, term^k_{2,i}, block^k_{2,i})$.
  We can thus construct an appropriate derivation, where $\overline{pc}^{k+1}_{2,i} = \overline{pc}^k_{2,i}\cdot\ell_{\mathsf{sb}}$ and $\overline{br}^{k+1}_{2,i} = \overline{br}^k_{2,i}\cdot(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$, and the result follows easily.
- $com^k_{2,i} = \mathsf{join}$. Here, $com^{k+1}_{2,i} = com'_{1,i} = \mathsf{stop}$ and $\overline{pc}_{1,i} = \overline{pc}'_{1,i}\cdot\ell_{\mathsf{sb}}$ and $\overline{br}_{1,i} = \overline{br}'_{1,i}\cdot(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$, and $time'_{1,i} = time_{1,i} \sqcup time_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}}$ and $term'_{1,i} = term_{1,i} \sqcup term_{\mathsf{sb}}$ and $block'_{1,i} = block_{1,i} \sqcup block_{\mathsf{sb}}$ and
  $$\Gamma'_{1,i} = \lambda x. \begin{cases} \Gamma_{1,i}(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_{1,i}) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_{1,i}(x) & \text{if } x \in pre(\Gamma_{1,i}) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$
  We can easily construct an appropriate derivation, such that $time'_{1,i} = time^{k+1}_{2,i}$ and $term'_{1,i} = term^{k+1}_{2,i}$ and $block'_{1,i} = block^{k+1}_{2,i}$ and $\overline{pc}'_{1,i} = \overline{pc}^{k+1}_{2,i}$ and $\overline{br}'_{1,i} = \overline{br}^{k+1}_{2,i}$ and for all x, $\Gamma'_{1,i}\langle x\rangle = \Gamma^{k+1}_{2,i}\langle x\rangle$, whereupon the result follows trivially.
- $com^k_{2,i} = \mathsf{if}\ e\ \mathsf{then}\ com_t\ \mathsf{else}\ com_f\ \mathsf{fi}$. Here, either $com'_{1,i} = com_t$ or $com'_{1,i} = com_f$ and $time_{1,i} = time^k_{2,i} = time'_{1,i} = time^{k+1}_{2,i}$ and $term_{1,i} = term^k_{2,i} = term'_{1,i} = term^{k+1}_{2,i}$ and $block_{1,i} = block^k_{2,i} = block'_{1,i} = block^{k+1}_{2,i}$ and for all x, $\Gamma_{1,i}\langle x\rangle = \Gamma^k_{2,i}\langle x\rangle = \Gamma'_{1,i}\langle x\rangle = \Gamma^{k+1}_{2,i}\langle x\rangle$.
  Also, we have $lmdst_{1,i} \triangleright \mathsf{mayread}(e)$ and
  $$(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma_{1,i}, lmdst_{1,i}, \overline{pc}_{1,i}, time_{1,i}, term_{1,i}, block_{1,i})$$
  and $\overline{pc}'_{1,i} = \overline{pc}_{1,i}\cdot\ell_{\mathsf{sb}}$ and $\overline{br}'_{1,i} = \overline{br}_{1,i}\cdot(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.
  Since $lmdst_{1,i} = lmdst^k_{2,i}$, and from the other equalities, we have $lmdst^k_{2,i} \triangleright \mathsf{mayread}(e)$ and $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}, \Gamma^k_{2,i}, lmdst^k_{2,i}, \overline{pc}^k_{2,i}, time^k_{2,i}, term^k_{2,i}, block^k_{2,i})$.
  By Definition 2, we have $\Gamma_{1,i}\langle e\rangle \sqsubseteq \ell_{\mathsf{sb}}$.
  We consider two possible cases, based on whether $\ell_{\mathsf{sb}} \sqsubseteq \ell$.
  If $\ell_{\mathsf{sb}} \sqsubseteq \ell$ then by ℓ-equivalence of $gcnf_1$ and $gcnf^k_2$, we have $mem_1(e) = mem^k_2(e)$. Thus, based on the inference rules for conditionals, $com'_{1,i} = com^{k+1}_{2,i}$, and we can construct appropriate derivations such that $pool'_1(i)$ is time-insensitive-ℓ-equivalent to $pool^{k+1}_2(i)$ because all of the thread state is identical.
  Otherwise, $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$, and so we don't know whether $com'_{1,i} = com^{k+1}_{2,i}$. However, in this case, $(\sqcup\overline{pc}'_{1,i}) \sqcup term'_{1,i} \sqcup block'_{1,i} \not\sqsubseteq \ell$ and $(\sqcup\overline{pc}^{k+1}_{2,i}) \sqcup term^{k+1}_{2,i} \sqcup block^{k+1}_{2,i} \not\sqsubseteq \ell$, and so Condition 2 of time-insensitive-ℓ-equivalence is trivially satisfied, as is Condition 3 for $j = |\overline{pc}^{k+1}_{2,i}|$.
- $com^k_{2,i} = \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$. Here, either $com'_{1,i} = com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$ or $com'_{1,i} = \mathsf{stop}$, based on the value of $mem_1(e)$. We consider these cases separately.
  - $com'_{1,i} = com; \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od}$.
    Here $time_{1,i} = time'_{1,i}$ and $term_{1,i} = term'_{1,i}$ and $block_{1,i} = block'_{1,i}$ and for all x, $\Gamma_{1,i}\langle x\rangle = \Gamma'_{1,i}\langle x\rangle$.
    Also, we have $lmdst_{1,i} \triangleright \mathsf{mayread}(e)$ and $\overline{pc}'_{1,i} = \overline{pc}_{1,i} = \overline{pc}\cdot\ell_{\mathsf{sb}}$ and $\overline{br}'_{1,i} = \overline{br}_{1,i} = \overline{br}\cdot(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.
    By Definition 2, we have $\Gamma_{1,i}\langle e\rangle \sqsubseteq \ell_{\mathsf{sb}}$.
    Since $lmdst_{1,i} = lmdst^k_{2,i}$, and from the other equalities, we have $lmdst^k_{2,i} \triangleright \mathsf{mayread}(e)$ and $\overline{br}^k_{2,i} = \overline{br}\cdot(time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.

We consider two possible cases, based on whether $\ell_{\mathsf{sb}} \sqsubseteq \ell$.
If $\ell_{\mathsf{sb}} \sqsubseteq \ell$ then by ℓ-equivalence of $gcnf_1$ and $gcnf_2^k$, we have $mem_1(e) = mem_2^k(e)$. Thus, based on the inference rules for loops, $com'_{1,i} = com_{2,i}^{k+1}$, and we can construct appropriate derivations such that $pool'_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2^{k+1}(i)$ because all of the thread state is identical.
Otherwise, $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$, and so we don't know whether $com'_{1,i} = com_{2,i}^{k+1}$. However, in this case, $(\sqcup \overline{pc}'_{1,i}) \sqcup term'_{1,i} \sqcup block'_{1,i} \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}_{2,i}^{k+1}) \sqcup term_{2,i}^{k+1} \sqcup block_{2,i}^{k+1} \not\sqsubseteq \ell$, and so Condition 2 of time-insensitive-ℓ-equivalence is trivially satisfied, as is Condition 3 for $j = max(|\overline{pc}'_{1,i}|, |\overline{pc}_{2,i}^{k+1}|)$.

– $com'_{1,i} = \mathsf{stop}$.
Here, $lmdst_{1,i} \triangleright \mathsf{mayread}(e)$ and $\overline{pc}_{1,i} = \overline{pc}'_{1,i} \cdot \ell_{\mathsf{sb}}$ and $\overline{br}_{1,i} = \overline{br}'_{1,i} \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ and $time'_{1,i} = time_{1,i} \sqcup time_{\mathsf{sb}} \sqcup \ell_{\mathsf{sb}}$ and $term'_{1,i} = term_{1,i} \sqcup term_{\mathsf{sb}}$ and $block'_{1,i} = block_{1,i} \sqcup block_{\mathsf{sb}}$ and

$$\Gamma'_{1,i} = \lambda x. \begin{cases} \Gamma_{1,i}(x) \sqcup \Gamma_{\mathsf{sb}}(x) & \text{if } x \in pre(\Gamma_{1,i}) \cap pre(\Gamma_{\mathsf{sb}}) \\ \Gamma_{1,i}(x) & \text{if } x \in pre(\Gamma_{1,i}) \setminus pre(\Gamma_{\mathsf{sb}}) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Also, we have By Definition 2, we have $\Gamma_{1,i}\langle e \rangle \sqsubseteq \ell_{\mathsf{sb}}$.
Since $lmdst_{1,i} = lmdst_{2,i}^k$, and from the other equalities, we have $lmdst_{2,i}^k \triangleright \mathsf{mayread}(e)$ and $\overline{br}_{2,i}^k = \overline{br}'_{1,i} \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.
We consider two possible cases, based on whether $\ell_{\mathsf{sb}} \sqsubseteq \ell$.
If $\ell_{\mathsf{sb}} \sqsubseteq \ell$ then by ℓ-equivalence of $gcnf_1$ and $gcnf_2^k$, we have $mem_1(e) = mem_2^k(e)$. Thus, based on the inference rules for loops, $com'_{1,i} = com_{2,i}^{k+1}$, and we can construct appropriate derivations such that $pool'_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2^{k+1}(i)$ because all of the thread state is identical.
Otherwise, $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$, and so we don't know whether $com'_{1,i} = com_{2,i}^{k+1}$. However, in this case, $(\sqcup \overline{pc}'_{1,i}) \sqcup term'_{1,i} \sqcup block'_{1,i} \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}_{2,i}^{k+1}) \sqcup term_{2,i}^{k+1} \sqcup block_{2,i}^{k+1} \not\sqsubseteq \ell$, and so Condition 2 of time-insensitive-ℓ-equivalence is trivially satisfied, as is Condition 3 for $j = max(|\overline{pc}'_{1,i}|, |\overline{pc}_{2,i}^{k+1}|)$.

- $com_{2,i}^k = \mathsf{term}$. This case is impossible, as $com_{1,i} = com_{2,i}^k$, and there is no $com'_{1,i}$ such that $(com_{1,i}, mem_1, \tau_1) \xrightarrow{\alpha,\epsilon}_{\sigma_1} (com'_{1,i}, mem'_1, \tau'_1)$.

■

Lemma 6 (Time-insensitive-ℓ-equivalence high-to-low step). *Let $gcnf_1 = \langle\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\rangle$ be a global configuration and let $gcnf_2^0$, $gcnf_2^1$, $gcnf_2^2$, $gcnf_2^3$, $\ldots$ be a (finite or infinite) sequence of global configurations such that for all $j < n$ we have $gcnf_2^j \twoheadrightarrow_{\sigma_2} gcnf_2^{j+1}$ (without taking a synchronization step), and the execution is fair (i.e., for any thread that can take a non-synchronization step, it eventually does). Moreover, suppose that $gcnf_1$ and $gcnf_2^0$ are well aligned, and that $gcnf_1$ is ℓ-equivalent to $gcnf_2^j$ for all $gcnf_2^j$ in the sequence.*
Let $gcnf_2^j = \langle\langle pool_2^j, mem_2^j, \tau_2^j, gmon_2^j \rangle\rangle$.
Let $i \in pre(pool_1)$ and assume $pool_1(i)$ and $pool_2^0(i)$ are time-insensitive-ℓ-equivalent.
Let $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf'_1$ be a non-synchronization step. Let $gcnf'_1 = \langle\langle pool'_1, mem'_1, \tau'_1, gmon'_1 \rangle\rangle$ and

$$pool_1(i) = [com_{1,i}, \langle \Gamma_{1,i}, lmdst_{1,i}, \overline{pc}_{1,i}, \overline{br}_{1,i}, time_{1,i}, term_{1,i}, block_{1,i} \rangle, mdst_{1,i}]$$

and

$$pool'_1(i) = [com'_{1,i}, \langle \Gamma'_{1,i}, lmdst'_{1,i}, \overline{pc}'_{1,i}, \overline{br}'_{1,i}, time'_{1,i}, term'_{1,i}, block'_{1,i} \rangle, mdst'_{1,i}]$$

If $pool_1(i) \neq pool'_1(i)$ and

$$(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$$

and

$$(\sqcup \overline{pc}'_{1,i}) \sqcup term'_{1,i} \sqcup block'_{1,i} \sqsubseteq \ell$$

then there exists a k such that $pool'_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2^{k+1}(i)$ (i.e., the second configuration will take some number of high steps, and eventually get to an equivalent thread state).

Proof: Assume $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}'_{1,i}) \sqcup term'_{1,i} \sqcup block'_{1,i} \sqsubseteq \ell$. By examination of the monitor rules, we see that the only way this can be true is if the monitor rule used for the step is (M-Join) or (M-Leave), i.e., the thread is exiting the scope of an if command or while command. More specifically, we must have $\overline{pc}'_{1,i} = \overline{pc}_{1,i} \cdot \ell_{\mathsf{sb}}$ where $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$ and $\overline{br}'_{1,i} = \overline{br}_{1,i} \cdot (time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ where $term_{\mathsf{sb}} \sqsubseteq \ell$ and $block_{\mathsf{sb}} \sqsubseteq \ell$. Moreover, since an if command or while command just finished, either $com'_{1,i} = \mathsf{stop}; com$ for some com, or $com'_{1,i} = \mathsf{stop}$. We will assume that $com'_{1,i} = \mathsf{stop}; com$, as the other case proceeds very similarly.

By Property 1, we have that $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a conservative approximation of the branching, timing, termination, and floating behavior for some thread state such that the execution of the ith thread had when it started the if command or while command that was just completed. Moreover, the join of the pc stack, termination, and blocking levels of the local monitor when it started the if command or while command was bounded above by ℓ (since termination and blocking levels increase monotonically, and the pc stack follows a stack discipline).

Let $j = |\overline{pc}_{1,i}| - 1$. We have $\overline{pc}_{1,i}(0) \sqcup \ldots \sqcup \overline{pc}_{1,i}(j-1) \sqcup term_{1,i} \sqcup block_{1,i} \sqsubseteq \ell$, since $\overline{pc}_{1,i}(0) \sqcup \ldots \sqcup \overline{pc}_{1,i}(j-1) = (\sqcup \overline{pc}'_{1,i})$, and $term_{1,i} \sqsubseteq term'_{1,i}$ and $block_{1,i} \sqsubseteq block'_{1,i}$ and $(\sqcup \overline{pc}'_{1,i}) \sqcup term'_{1,i} \sqcup block'_{1,i} \sqsubseteq \ell$.

So because $pool_1(i)$ and $pool_2^0(i)$ are time-insensitive-ℓ-equivalent, we have that $pool_1(i).\mathsf{com} = com_{cnt}$ for some com_{cnt} and either $pool_2^0(i).\mathsf{com} = com_{cnt}$ or $pool_2^0(i).\mathsf{com} = com'; com_{cnt}$ for some com'. Moreover, let $pool_2^0(i).\mathsf{lmon} = \langle \Gamma^0_{2,i}, lmdst^0_{2,i}, \overline{pc}^0_{2,i}, \overline{br}^0_{2,i}, time^0_{2,i}, term^0_{2,i}, block^0_{2,i} \rangle$; we have that $\overline{br}^0_{2,i}(j) = \overline{br}_{1,i}(j) = (\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$.

Thus $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ is a conservative approximation for the command that $pool_2^0(i)$ is in the middle of executing.

Since $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}) \sqcup term \sqcup block \sqsubseteq \ell$ and $term_{\mathsf{sb}} \sqcup block_{\mathsf{sb}} \sqsubseteq \ell$, we have that for any execution through the command that $pool_2^0(i)$ is in the middle of executing, it cannot encounter a while loop or possibly reach a state where the monitor might block (since otherwise the termination or blocking level would be at least ℓ_{sb}, which is impossible, by the definition of conservative approximation). Since it cannot loop indefinitely or block, it must terminate.

Thus, we have that there exists a k such that $pool_2^{k+1}(i).\mathsf{com} = pool'_1(i).\mathsf{com}$.

Consider the time-insensitive-ℓ-equivalence of $pool'_1(i)$ and $pool_2^{k+1}(i)$. Conditions 2 and 3 of Definition 6 follow immediately from the fact that $pool_2^{k+1}(i).\mathsf{com} = pool'_1(i).\mathsf{com}$. By examining the (M-Join) and (M-Leave) monitor rules, we see that $\ell_{\mathsf{sb}} \sqsubseteq time'_{1,i}$, and since $\ell_{\mathsf{sb}} \not\sqsubseteq \ell$, we have $time'_{1,i} \not\sqsubseteq \ell$, thus satisfying Condition 1. ■

We can now prove Lemma 3.

Proof of Lemma 3: Let $gcnf_1 = \langle\!\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\!\rangle$ and $gcnf_2 = \langle\!\langle pool_2, mem_2, \tau_2, gmon_2 \rangle\!\rangle$ be ℓ-equivalent global configurations, and let σ_1 and σ_2 be ℓ-equivalent strategies. Assume that $gcnf_1$ and $gcnf_2$ are well aligned.

Let $gcnf'_1 = \langle\!\langle pool'_1, mem'_1, \tau'_1, gmon'_1 \rangle\!\rangle$.

Let $gcnf_1 \twoheadrightarrow^*_{\sigma_1} gcnf'_1$ without executing a synchronization step.

Assume $i \in pre(pool_1)$ and $pool_1(i)$ and $pool_2(i)$ are time-insensitive-ℓ-equivalent, and

$$pool'_1(i) = [com'_i, \langle \Gamma'_i, lmdst'_i, \overline{pc}'_i, \overline{br}'_i, time'_i, term'_i, block'_i \rangle, mdst'_i]$$

where

$$(\sqcup \overline{pc}'_i) \sqcup term'_i \sqcup block'_i \sqsubseteq \ell.$$

Suppose we have a fair execution from $gcnf_2$ without executing a synchronization step (i.e., we do not get to choose how $gcnf_2$ evolves, but each thread eventually gets the opportunity to make progress).

We proceed by induction on the length of the execution $gcnf_1 \twoheadrightarrow^*_{\sigma_1} gcnf'_1$.

Base case: Clearly if $gcnf_1 = gcnf'_1$ then the result holds immediately.

Inductive case: Consider $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf''_1$. For notational convenience, let $gcnf''_1 = \langle\!\langle pool''_1, mem''_1, \tau''_1, gmon''_1 \rangle\!\rangle$, and

$$pool''_1(i) = [com''_{1,i}, \langle \Gamma''_{1,i}, lmdst''_{1,i}, \overline{pc}''_{1,i}, \overline{br}''_{1,i}, time''_{1,i}, term''_{1,i}, block''_{1,i} \rangle, mdst''_{1,i}].$$

There are 4 (mutually exclusive and exhaustive) cases.

1) $pool''_1(i) = pool_1(i)$.
   Here, the step $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf''_1$ did not advance thread i, and so $pool''_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2(i)$ as required.
2) $pool''_1(i) \neq pool_1(i)$ and $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}''_{1,i}) \sqcup term''_{1,i} \sqcup block''_{1,i} \not\sqsubseteq \ell$.
   Here, the ith thread took a step and had a high (time-exclusive) pc level before and after the step.
   By Lemma 4, $pool''_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2(i)$ as required.
3) $pool''_1(i) \neq pool_1(i)$ and $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \sqsubseteq \ell$.
   Here, the ith thread took a step and had a low (time-exclusive) pc level before the step.
   By Lemma 5, the next step that thread i takes in the 2nd execution will be to a thread configuration that is time-insensitive-ℓ-equivalent, as required.
4) $pool''_1(i) \neq pool_1(i)$ and $(\sqcup \overline{pc}_{1,i}) \sqcup term_{1,i} \sqcup block_{1,i} \not\sqsubseteq \ell$ and $(\sqcup \overline{pc}''_{1,i}) \sqcup term''_{1,i} \sqcup block''_{1,i} \sqsubseteq \ell$.
   Here, the ith thread took a step and had a high (time-exclusive) pc level before the step, and a low (time-exclusive) pc level before the step.

By Lemma 6, thread i in the 2nd execution will eventually reach a thread configuration that is time-insensitive-ℓ-equivalent, as required.

■

We state and prove one additional lemma, that shows that time-insensitive-ℓ-equivalence is established when two ℓ-equivalent local configurations go from a "low pc" to a "high pc".

Lemma 7 (Establishment of time-insensitive-ℓ-equivalence). *Let ℓ be a security level. Let σ_1 and σ_2 be ℓ-equivalent strategies and let*

$$lcnf_1 = \langle [com_1, \langle \Gamma_1, lmdst_1, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1\rangle, mdst_1], mem_1, \tau_1\rangle$$

and

$$lcnf_2 = \langle [com_2, \langle \Gamma_2, lmdst_2, \overline{pc}_2, \overline{br}_2, time_2, term_2, block_2\rangle, mdst_2], mem_2, \tau_2\rangle$$

be ℓ-equivalent local configurations such that

$$time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell.$$

let

$$lcnf'_1 = \langle [com'_1, \langle \Gamma'_1, lmdst'_1, \overline{pc}'_1, \overline{br}'_1, time'_1, term'_1, block'_1\rangle, mdst'_1], mem'_1, \tau'_1\rangle$$

and

$$lcnf'_2 = \langle [com'_2, \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle, mdst'_2], mem'_2, \tau'_2\rangle$$

be ℓ-equivalent local configurations such that

$$time'_1 \sqcup (\sqcup \overline{pc}'_1) \sqcup term'_1 \sqcup block'_1 \not\sqsubseteq \ell.$$

Let $lcnf_1 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_1} lcnf'_1$ and $lcnf_2 \xrightarrow{\epsilon,\epsilon,\epsilon}_{\sigma_2} lcnf'_2$.
Then thread states

$$[com'_1, mc'_1, mdst'_1] \text{ and } [com'_2, mc'_2, mdst'_2]$$

are time-insensitive-ℓ-equivalent, where

$$mc'_1 = \langle \Gamma'_1, lmdst'_1, \overline{pc}'_1, \overline{br}'_1, time'_1, term'_1, block'_1\rangle$$

and

$$mc'_2 = \langle \Gamma'_2, lmdst'_2, \overline{pc}'_2, \overline{br}'_2, time'_2, term'_2, block'_2\rangle.$$

Proof: By the definition of ℓ-equivalence of local configurations (condition 7), since

$$time_1 \sqcup (\sqcup \overline{pc}_1) \sqcup term_1 \sqcup block_1 \sqsubseteq \ell$$

we have

$$time_2 \sqcup (\sqcup \overline{pc}_2) \sqcup term_2 \sqcup block_2 \sqsubseteq \ell$$

and since

$$time'_1 \sqcup (\sqcup \overline{pc}'_1) \sqcup term'_1 \sqcup block'_1 \not\sqsubseteq \ell$$

we have

$$time'_2 \sqcup (\sqcup \overline{pc}'_2) \sqcup term'_2 \sqcup block'_2 \not\sqsubseteq \ell.$$

Thus Condition 1 of time-insensitive-ℓ-equivalence holds.

For Conditions 2 and 3 of time-insensitive-ℓ-equivalence, consider how the pc level can be raised. It must be because one of the monitor rules M-Branch, M-Join, M-Enter, or M-Leave was used. In all of these cases, since the monitor states and commands of $lcnf_1$ and $lcnf_2$ were identical, the monitor states and commands of $lcnf'_1$ and $lcnf'_2$ are identical. ■

Having proven Lemma 3, we can now state and prove the global barrier theorem.

Theorem 13 (Global barrier). *Let $gcnf_1$ and $gcnf_2$ be ℓ-equivalent global configurations, and let σ_1 and σ_2 be ℓ-equivalent strategies. Assume that $gcnf_1$ and $gcnf_2$ are well aligned.*

Suppose $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf'_1$ by all live thread configurations taking a synchronization step (i.e., second rule of Figure 2).

There exists a finite sequence of global configurations $gcnf^0_2$, $gcnf^1_2$, ..., $gcnf^n_2, gcnf^{n+1}_2$, such that $gcnf^0_2 = gcnf_2$ and for all $i < n$ we have $gcnf^i_2 \twoheadrightarrow_{\sigma_2} gcnf^{i+1}_2$ (without taking a synchronization step), and $gcnf^n_2 \twoheadrightarrow_{\sigma_2} gcnf^{n+1}_2$ is a synchronization step. Moreover, for all $0 \leq i \leq n$ we have that $gcnf^i_2$ is ℓ-equivalent to $gcnf_1$ and $gcnf^{n+1}_2$ is ℓ-equivalent to $gcnf'_1$.

Proof: Let $gcnf_1 = \langle\langle pool_1, mem_1, \tau_1, gmon_1 \rangle\rangle$ and $gcnf_2 = \langle\langle pool_2, mem_2, \tau_2, gmon_2 \rangle\rangle$.

First, construct a fair schedule for the second execution (i.e., that gives each thread in $gcnf_2$ infinitely many time steps). Execute $gcnf_2$ using that schedule, preventing barrier synchronization steps. Note that at this point, the execution may be finite (if all threads either get stuck, terminate, or reach a barrier), or infinite (if at least one thread diverges).

Now consider each thread $i \in pre(pool_1)$. Let

$$pool_1(i) = [com_1^i, \langle \Gamma_1^i, lmdst_1^i, \overline{pc}_1^i, \overline{br}_1^i, time_1^i, term_1^i, block_1^i \rangle, mdst_1^i]$$

and

$$pool_2(i) = [com_2^i, \langle \Gamma_2^i, lmdst_2^i, \overline{pc}_2^i, \overline{br}_2^i, time_2^i, term_2^i, block_2^i \rangle, mdst_2^i].$$

If $time_1^i \sqcup (\sqcup \overline{pc}_1^i) \sqcup term_1^i \sqcup block_1^i \sqsubseteq \ell$ then, by ℓ-equivalence of $gcnf_1$ and $gcnf_2$ we have $time_1^i = time_2^i$ and $term_1^i = term_2^i$ and $block_1^i = block_2^i$ and $\overline{pc}_1^i = \overline{pc}_2^i$ and $\overline{br}_1^i = \overline{br}_2^i$ and $com_1^i = com_2^i$ and for all x, $\Gamma_1^i\langle x \rangle = \Gamma_2^i \langle x \rangle$. This implies that (1) if thread $i \notin alive(pool_1)$, then $com_1^i = \mathsf{term} = com_2^i$, and so $i \notin alive(pool_2)$; and (2) if $i \in alive(pool_1)$ then $pool_1(i)$ is ready to execute the barrier step, and all pre-conditions are satisfied, and thus $pool_2(i)$ is also ready to execute the barrier.

Otherwise, $time_1^i \sqcup (\sqcup \overline{pc}_1^i) \sqcup term_1^i \sqcup block_1^i \not\sqsubseteq \ell$ but $(\sqcup \overline{pc}_1^i) \sqcup term_1^i \sqcup block_1^i \sqsubseteq \ell$. Consider the execution that lead up to $gcnf_1$. There was some step from one global configuration to some global configuration $gcnf_1^{init}$ such that the timing-inclusive program counter level (i.e., $time \sqcup (\sqcup \overline{pc}) \sqcup term \sqcup block$) was $\sqsubseteq \ell$ before the step, and $\not\sqsubseteq \ell$ after the step, and remained $\not\sqsubseteq \ell$ until $gcnf_1$ was reached. Note that since that step, no synchronization step could have been taken during the execution. If we consider the execution that lead to $gcnf_1$, there was some global configuration $gcnf_2^{init}$ such that $gcnf_1^{init}$ and $gcnf_2^{init}$ are ℓ-equivalent, and $gcnf_2^{init} \twoheadrightarrow_{\sigma_2}^* gcnf_2$. Note that the ith thread of $gcnf_1^{init}$ is time-insensitive-ℓ-equivalent to the ith thread of $gcnf_2^{init}$, by Lemma 7. Then, by Lemma 3, the execution from $gcnf_2^{init}$ to $gcnf_2$ and beyond will eventually reach a global configuration $gcnf_2' = \langle\langle pool_2', mem_2', \tau_2', gmon_2' \rangle\rangle$ such that $gcnf_2'(i) = [com_2', \langle \Gamma_2', lmdst_2', \overline{pc}_2', \overline{br}_2', time_2', term_2', block_2' \rangle, mdst_2']$ is time-insensitive-ℓ-equivalent to $pool_1(i)$. Since $(\sqcup \overline{pc}_1^i) \sqcup term_1^i \sqcup block_1^i \sqsubseteq \ell$, by Definition 6 (Condition 2), we have that $time_1^i = time_2'$ and $term_1^i = term_2'$ and $block_1^i = block_2'$ and $\overline{pc}_1^i = \overline{pc}_2'$ and $\overline{br}_1^i = \overline{br}_2'$ and $com_1^i = com_2'$ and for all x, $\Gamma_1^i\langle x \rangle = \Gamma_2'\langle x \rangle$. This implies that (1) if thread $i \notin alive(pool_1)$, then $com_1^i = \mathsf{term} = com_2'$, and so $i \notin alive(pool_2')$; and (2) if $i \in alive(pool_1)$ then $pool_1(i)$ is ready to execute the barrier step, and all pre-conditions are satisfied, and thus $pool_2'(i)$ is also ready to execute the barrier.

Thus, since each thread i eventually reaches the barrier or terminates, and no thread can take a step over the barrier until all non-terminated threads are ready to step over the barrier, we have that there is a finite sequence of global configurations $gcnf_2^0$, $gcnf_2^1$, $\ldots$, $gcnf_2^n$, such that $gcnf_2^0 = gcnf_2$ and for all $i < n$ we have $gcnf_2^i \twoheadrightarrow_{\sigma_2} gcnf_2^{i+1}$ (without taking a synchronization step). By induction on this execution, we can show, by repeated application of Theorem 12, that for all $0 \le i \le n$ we have that $gcnf_2^i$ is ℓ-equivalent to $gcnf_1$.

Thus, $gcnf_2^n$ can take a barrier step, since all preconditions for the barrier step are satisfied: $gcnf_2^n \twoheadrightarrow_{\sigma_2} gcnf_2^{n+1}$. We now need to show that $gcnf_2^{n+1}$ is ℓ-equivalent to $gcnf_1'$. To give names to the components of $gcnf_1'$, $gcnf_2^n$, and $gcnf_2^{n+1}$ let

$$gcnf_1' = \langle\langle pool_1', mem_1', \tau_1', gmon_1' \rangle\rangle$$

and

$$gcnf_2^n = \langle\langle pool_2^n, mem_2^n, \tau_2^n, gmon_2^n \rangle\rangle$$

and

$$gcnf_2^{n+1} = \langle\langle pool_2^{n+1}, mem_2^{n+1}, \tau_2^{n+1}, gmon_2^{n+1} \rangle\rangle$$

and

$$pool_1'(i) = [com_{1,i}', \langle \Gamma_{1,i}', lmdst_{1,i}', \overline{pc}_{1,i}', \overline{br}_{1,i}', time_{1,i}', term_{1,i}', block_{1,i}' \rangle, mdst_{1,i}']$$

and

$$pool_2^n(i) = [com_{2,i}^n, \langle \Gamma_{2,i}^n, lmdst_{2,i}^n, \overline{pc}_{2,i}^n, \overline{br}_{2,i}^n, time_{2,i}^n, term_{2,i}^n, block_{2,i}^n \rangle, mdst_{2,i}^n]$$

and

$$pool_2^{n+1}(i) = [com_{2,i}^{n+1}, \langle \Gamma_{2,i}^{n+1}, lmdst_{2,i}^{n+1}, \overline{pc}_{2,i}^{n+1}, \overline{br}_{2,i}^{n+1}, time_{2,i}^{n+1}, term_{2,i}^{n+1}, block_{2,i}^{n+1} \rangle, mdst_{2,i}^{n+1}].$$

Note that we have $alive(pool_1') = alive(pool_2^{n+1})$, and that the annotation requests in both executions are identical (since annotation requests come from the commands, which are identical).

Note that since $gcnf_2^n$ is ℓ-equivalent to $gcnf_1$ and the barrier step does not modify the trace or memory, or domain of the pool states, we have that $\tau_1' \downarrow \ell = \tau_2^{n+1} \downarrow \ell$ and $pre(pool_1') = pre(pool_2^{n+1})$.

It just remains to show that $\forall i \in pre(pool_1') : \langle pool_1'(i), mem_1', \tau_1' \rangle$ is ℓ-equivalent to $\langle pool_2^{n+1}(i), mem_2^{n+1}, \tau_2^{n+1} \rangle$. Let $i \in pre(pool_1')$. We will apply Theorem 9. Note that if $time_{2,i}^n \sqcup (\sqcup \overline{pc}_{2,i}^n) \sqcup term_{2,i}^n \sqcup block_{2,i}^n \sqsubseteq \ell$, then from ℓ-equivalence of

$\langle pool_1(i), mem_1, \tau_1\rangle$ and $\langle pool_2^n(i), mem_2^n, \tau_2^n\rangle$, the preconditions of Theorem 9 are satisfied. Otherwise, if $time_{2,i}^n \sqcup (\sqcup \overline{pc}_{2,i}^n) \sqcup term_{2,i}^n \sqcup block_{2,i}^n \not\sqsubseteq \ell$, then, by construction, $pool_1(i)$ is time-insensitive-ℓ-equivalent to $pool_2^n(i)$, and the preconditions of Theorem 9 are satisfied. Either way, we can apply Theorem 9 and get the required result.

■

G. Finally, the soundness proof

Using the previously defined lemmas, we now define the key lemma, which says that given ℓ-equivalent global configurations, when one configuration takes a step, the other configuration can take zero or more steps to an ℓ-equivalent global configuration.

Lemma 8. *Let $gcnf_1$ and $gcnf_2$ be ℓ-equivalent global configurations, and let σ_1 and σ_2 be ℓ-equivalent strategies. Assume that $gcnf_1$ and $gcnf_2$ are well aligned.*

If $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf_1'$ then there exists $gcnf_2'$ such that $gcnf_2 \twoheadrightarrow_{\sigma_2}^* gcnf_2'$ and $gcnf_1'$ and $gcnf_2'$ are ℓ-equivalent and $gcnf_1'$ and $gcnf_2'$ are well aligned.

Proof: Consider the possible derivations of $gcnf_1 \twoheadrightarrow_{\sigma_1} gcnf_1'$. If the step was due to a barrier synchronization, then the result holds immediately by Theorem 13. Otherwise, if the step was an individual thread, then it was either a low step or a high step. If it was a low step, the result follows by Theorem 11, and if it was a high step, the result follows by Theorem 12. ■

We are now ready to prove Theorem 3.

Proof of Theorem 3: Let $pool \in PSt$, $\sigma \in \Sigma$, and $\langle\langle pool', mem', \tau', gmon'\rangle\rangle \in reach_\sigma(pool)$. Let ℓ be a security level. We need to show that

$$\kappa(\ell, pool, \tau') = \{\sigma' \in \Sigma \mid \sigma =_\ell \sigma'\}$$

where

$$\kappa(\ell, pool, \tau') = \{\sigma' \in \Sigma \mid \exists \langle\langle pool'', mem'', \tau'', gmon''\rangle\rangle \in reach_{\sigma'}(pool) : \tau'' \downarrow \ell = \tau' \downarrow \ell\}$$

Let σ' be a strategy such that $\sigma =_\ell \sigma'$. We need to show that $\sigma' \in \kappa(\ell, pool, \tau')$. That is, we need to show that there exists $\langle\langle pool'', mem'', \tau'', gmon''\rangle\rangle \in reach_{\sigma'}(pool)$ such that $\tau'' \downarrow \ell = \tau' \downarrow \ell$.

Since $\langle\langle pool', mem', \tau', gmon'\rangle\rangle \in reach_\sigma(pool)$, there is an execution $\langle\langle pool, mem_{init}, \tau_{init}, gmon_{init}\rangle\rangle \twoheadrightarrow_\sigma^* \langle\langle pool', mem', \tau', gmon'\rangle\rangle$.

First note that $\langle\langle pool, mem_{init}, \tau_{init}, gmon\rangle\rangle$ is ℓ-equivalent to itself, and so by repeated applications of Lemma 8, we can construct a global configuration $\langle\langle pool'', mem'', \tau'', gmon''\rangle\rangle$ such that

$$\langle\langle pool, mem_{init}, \tau_{init}, gmon_{init}\rangle\rangle \twoheadrightarrow_{\sigma'}^* \langle\langle pool'', mem'', \tau'', gmon''\rangle\rangle$$

and $\langle\langle pool'', mem'', \tau'', gmon''\rangle\rangle$ is ℓ-equivalent to $\langle\langle pool', mem', \tau', gmon'\rangle\rangle$. From the definition of ℓ-equivalence for global configurations (Definition 4), we have $\tau'' \downarrow \ell = \tau' \downarrow \ell$ as required.

■

APPENDIX F
SOUNDNESS OF STATIC BOUNDS ORACLE

In this section we define the static bounds oracle, and sketch the proof of its soundness.

STATIC-SKIP

$$\frac{}{\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{skip} : \bot, \overline{pc}, \overline{br}, \mathit{\Gamma}, time, term, block}$$

STATIC-ASSIGN

$$\frac{\begin{array}{c}\ell = \mathit{\Gamma}\langle e\rangle \sqcup time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block \qquad \mathit{\Gamma}' = \mathit{\Gamma}\langle x \mapsto_{lmdst} \ell\rangle \\ block' = \begin{cases} block & \text{if } lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)] \text{ and } \ell \sqsubseteq \mathcal{L}(x) \\ block & \text{if } lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)] \\ block \sqcup term \sqcup time \sqcup (\sqcup\overline{pc}) & \text{otherwise}\end{cases}\end{array}}{\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash x := e : \bot, \overline{pc}, \overline{br}, \mathit{\Gamma}', time, term, block'}$$

STATIC-INPUT

$$\frac{\begin{array}{c}\ell^* = time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block \qquad \ell = ch \sqcup \ell^* \qquad \mathit{\Gamma}' = \mathit{\Gamma}\langle x \mapsto_{lmdst} \ell\rangle \\ block' = \begin{cases} block & \text{if } lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{othersmightread}(x)] \text{ and } \ell^* \sqsubseteq ch \text{ and } \ell \sqsubseteq \mathcal{L}(x) \\ block & \text{if } lmdst \triangleright [\mathsf{maywrite}(x), \mathsf{exclusiveread}(x)] \text{ and } \ell^* \sqsubseteq ch \\ block \sqcup term \sqcup time \sqcup (\sqcup\overline{pc}) & \text{otherwise}\end{cases}\end{array}}{\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{input}\ ch\ \mathsf{to}\ x : \bot, \overline{pc}, \overline{br}, \mathit{\Gamma}', time, term, block'}$$

STATIC-OUTPUT

$$\frac{block' = \begin{cases} block & \text{if } lmdst \triangleright \mathsf{mayread}(e) \text{ and } \mathit{\Gamma}\langle e\rangle \sqcup\ time \sqcup (\sqcup\overline{pc}) \sqcup term \sqsubseteq ch \\ block \sqcup term \sqcup time \sqcup (\sqcup\overline{pc}) & \text{otherwise}\end{cases}}{\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{output}\ e\ \mathsf{to}\ ch : \bot, \overline{pc}, \overline{br}, \mathit{\Gamma}, time, term, block'}$$

STATIC-SEQ

$$\frac{\begin{array}{c}\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com_1 : \ell_1, \overline{pc}_1, \overline{br}_1, \mathit{\Gamma}_1, time_1, term_1, block_1 \\ \mathit{\Gamma}_1, lmdst, \overline{pc}_1, \overline{br}_1, time_1, term_1, block_1 \vdash com_2 : \ell_2, \overline{pc}_2, \overline{br}_2, \mathit{\Gamma}_2, time_2, term_2, block_2\end{array}}{\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com_1; com_2 : \bot, \overline{pc}_2, \overline{br}_2, \mathit{\Gamma}_2, time_2, term_2, block_2}$$

STATIC-BRANCH

$$\frac{\begin{array}{c} com_i \in \mathbf{S} \qquad block' = \begin{cases} block & \text{if } lmdst \triangleright \mathsf{mayread}(e) \\ block \sqcup term \sqcup time \sqcup (\sqcup\overline{pc}) & \text{otherwise}\end{cases} \qquad \ell = \mathit{\Gamma}\langle e\rangle \sqcup time \qquad \ell \neq \bot \\ \overline{br}'_i = \overline{br}\cdot(time_i, term_i, block_i, \mathit{\Gamma}_i) \qquad \mathit{\Gamma}, lmdst, \overline{pc}\cdot\ell, \overline{br}'_i, time, term, block' \vdash com_i : \ell_i, \overline{pc}\cdot\ell, \overline{br}'_i, \mathit{\Gamma}_i, time_i, term_i, block_i \\ (i = 1, 2) \qquad \mathit{\Gamma}' = \lambda y \in pre(\mathit{\Gamma})\ .\ \mathit{\Gamma}_1(y) \sqcup \mathit{\Gamma}_2(y)\end{array}}{\begin{array}{l}\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi} : \\ \qquad\qquad \ell, \overline{pc}, \overline{br}, \mathit{\Gamma}', time_1 \sqcup time_2 \sqcup \ell, term_1 \sqcup term_2, block_1 \sqcup block_2\end{array}}$$

Fig. 10. Flow-sensitive (timing/control-sensitive) static analysis of the termination and typing environments bounds (part 1)

Definition 7 (Surface syntax). *Define set* $\mathbf{S}$ *to be the set of commands that do not contain commands* join *or* more e do com od.

To define the static bounds oracle, we use a flow-sensitive type judgement

$$\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell', \mathit{\Gamma}', time', term', block'$$

This judgment is defined in Figures 10–12. Using this judgment we define the static bounds oracle below.

Definition 8 (Static bounds operator). *Given program $com \in \mathbf{S}$, environment $\mathit{\Gamma}$, mode state $lmdst$, and levels $time$, $term$ and $block$, define partial function* $\mathsf{SB}(com, \mathit{\Gamma}, lmdst, \overline{pc}, time, term, block)$ *to return a tuple of form* $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \mathit{\Gamma}_{\mathsf{sb}})$ *as follows.*

1) If com is if e then com_1 else com_2 fi *such that* $(\sqcup\overline{pc}) \sqcup \mathit{\Gamma}\langle e\rangle \sqcup time = \bot$ *then return tuple* $(\bot, \bot, \bot, \bot, \mathit{\Gamma})$.

$$\textsc{Static-Loop-Algorithmic}$$

$$\dfrac{\begin{array}{c} com \in \mathbf{S} \qquad \forall x \in e.\ x \in \mathit{NonFloatVar} \qquad \ell_0 = \Gamma\langle e\rangle \sqcup time \qquad \ell_{i+1} = \Gamma'_{i+1}\langle e\rangle \sqcup \ell_i \qquad \ell_i \neq \bot \\ \overline{br}'_i = \overline{br}\cdot(time'_i, term'_i, block'_i, \Gamma'_i) \\ \Gamma'_i, lmdst, \overline{pc}\cdot\ell_i, \overline{br}'_i, time, term, block \vdash com : \ell^\star_i, \overline{pc}\cdot\ell_i, \overline{br}'_i, \Gamma''_i, time''_i, term''_i, block''_i \qquad 0 \leq i \leq n \\ \Gamma'_0 = \Gamma, \quad \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \quad \Gamma'_{n+1} = \Gamma'_n \\ time'_i = time''_1 \sqcup \ldots \sqcup time''_i \quad time'_{n+1} = time'_n \\ term'_i = term''_1 \sqcup \ldots \sqcup term''_i \sqcup \ell_i \sqcup \overline{pc}, \quad term'_{n+1} = term'_n \\ block'_i = \begin{cases} block \sqcup block''_1 \sqcup \ldots \sqcup block''_i & \text{if } lmdst \rhd \mathsf{mayread}(e) \\ block \sqcup block''_1 \sqcup \ldots \sqcup block''_i \sqcup term \sqcup time \sqcup (\sqcup\overline{pc}) & \text{otherwise} \end{cases} \\ block'_{n+1} = block'_n \end{array}}{\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od} : \ell_n, \overline{pc}, \overline{br}, \Gamma'_n, time'_n, term'_n, block'_n}$$

Fig. 11. Flow-sensitive (timing/control-sensitive) static analysis of the termination and typing environments bounds (part 2)

$$\textsc{Static-Stop}$$

$$\dfrac{}{\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{stop} : \bot, \overline{pc}, \overline{br}, \Gamma, time, term, block}$$

$$\textsc{Static-Join}$$

$$\dfrac{\Gamma'' = \lambda y \in pre(\Gamma)\ .\ \Gamma(y) \sqcup \Gamma'(y)}{\Gamma, lmdst, \overline{pc}\cdot\ell, \overline{br}\cdot(time', term', block', \Gamma'), time, term, block \vdash \mathsf{join} : \bot, \overline{pc}, \overline{br}, \Gamma'', time \sqcup time' \sqcup \ell, term \sqcup term', block \sqcup block'}$$

$$\textsc{Static-More}$$

$$\dfrac{\begin{array}{c} com \in \mathbf{S} \qquad \overline{br} = \overline{br}'\cdot(time^\star, term^\star, block^\star, \Gamma^\star) \qquad \overline{pc} = \overline{pc}'\cdot\ell^\star \\ \forall x \in e.\ x \in \mathit{NonFloatVar} \qquad \ell_0 = \Gamma\langle e\rangle \sqcup time \qquad \ell_{i+1} = \Gamma'_i\langle e\rangle \sqcup \ell_i \qquad \ell_i \neq \bot \qquad \overline{br}'_i = \overline{br}'\cdot(time_i, term_i, block_i, \Gamma_i) \\ \Gamma'_i, lmdst, \overline{pc}'\cdot\ell_i, \overline{br}'_i, time, term, block \vdash com : \ell^\star_i, \overline{pc}'\cdot\ell_i, \overline{br}'_i, \Gamma''_i, time''_i, term''_i, block''_i \qquad 0 \leq i \leq n \\ \Gamma'_0 = \Gamma, \quad \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma \quad \Gamma'_{n+1} = \Gamma'_n \\ time'_i = time''_1 \sqcup \ldots \sqcup time''_i \quad time'_{n+1} = time'_n \\ term'_i = term''_1 \sqcup \ldots \sqcup term''_i \sqcup \ell_i \sqcup \overline{pc} \quad term'_{n+1} = term'_n \\ block'_i = \begin{cases} block \sqcup block''_1 \sqcup \ldots \sqcup block''_i & \text{if } lmdst \rhd \mathsf{mayread}(e) \\ block \sqcup block''_1 \sqcup \ldots \sqcup block''_i \sqcup term \sqcup time \sqcup (\sqcup\overline{pc}) & \text{otherwise} \end{cases} \\ block'_{n+1} = block'_n \\ \Gamma'_n \sqsubseteq \Gamma^\star \qquad time'_n \sqsubseteq time^\star \qquad term'_n \sqsubseteq term^\star \qquad block'_n \sqsubseteq block^\star \end{array}}{\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{more}\ e\ \mathsf{do}\ com\ \mathsf{od} : \bot, \overline{pc}', \overline{br}', \Gamma^\star, time^\star, term^\star, block^\star}$$

Fig. 12. Extension of flow-sensitive (timing/control-sensitive) static analysis to intermediate commands

2) *If com is* $\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}$ *such that* $\forall x \in e.\ x \in \mathit{NonFloatVar}$ *and* $(\sqcup\overline{pc}) \sqcup \Gamma\langle e\rangle \sqcup time = \bot$ *then return tuple* $(\bot, \bot, \bot, \bot, \Gamma)$.
3) *If none of the above applies, then return* $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ *such that* $\Gamma, lmdst, \overline{pc}, \epsilon, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}, \epsilon, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$

Note that the static bounds oracle is a partial function: it may not return any result, which would cause the local monitor to block.

The static bounds operator is sound, in that it returns a conservative approximation for its arguments. We sketch the proof of this claim below.

Definition 9 (Well-formedness of typing environments with respect to mode states)**.** *Given Γ and $lmdst$, say that Γ is* well-formed *w.r.t. $lmdst$, written $lmdst$* ok *Γ, if*

1) *For all variables x, $lmdst \rhd \mathsf{othersmightread}(x) \implies \Gamma\langle x\rangle \sqsubseteq \mathcal{L}(x)$*

2) For all variables x, $lmdst \triangleright \mathsf{othersmightwrite}(x) \implies \mathcal{L}(x) \sqsubseteq \mathit{\Gamma}\langle x\rangle$

Lemma 9 (Local properties of static bounds)**.** *Given $\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block$ such that*

1) $term \sqsubseteq time$
2) $lmdst$ ok $\mathit{\Gamma}$
3) $\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \mathit{\Gamma}_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$

then it holds that

1) $term_{\mathsf{sb}} \sqsubseteq time_{\mathsf{sb}}$
2) $lmdst$ ok $\mathit{\Gamma}_{\mathsf{sb}}$

Proof: Straightforward induction on the typing derivation. ■

Lemma 10 (Monotonicity of branching environments)**.** *For all com, $\mathit{\Gamma}$, lmdst, $\overline{pc}$, $\overline{br}$, time, term, block, and $\overline{br}'$ it holds that*

$$\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \mathit{\Gamma}_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

then $\overline{br}_{\mathsf{sb}}$ is a prefix of $\overline{br}$.

Proof: By induction on the typing derivation. ■

Lemma 11 (Irrelevance of branching environments for surface commands)**.** *For all com $\in$ **S**, such that, and all $\mathit{\Gamma}$, lmdst, $\overline{pc}$, $\overline{br}$, time, term, block, and $\overline{br}'$ it holds that*

$$\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}, \mathit{\Gamma}_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

then

$$\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}', time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}', \mathit{\Gamma}_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

Proof: By induction on the typing derivation using Lemma 10. ■

Lemma 12 (Static bound typing rules are deterministic)**.** *For all com, $\mathit{\Gamma}$, lmdst, $\overline{pc}$, $\overline{br}$, time, term, block, and $\overline{br}'$ such that*

$$\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}, \mathit{\Gamma}_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

and

$$\mathit{\Gamma}, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell'_{\mathsf{sb}}, \overline{pc}'_{\mathsf{sb}}, \overline{br}', \mathit{\Gamma}'_{\mathsf{sb}}, time'_{\mathsf{sb}}, term'_{\mathsf{sb}}, block'_{\mathsf{sb}}$$

it holds that

1) $\ell_{\mathsf{sb}} = \ell'_{\mathsf{sb}}$
2) $\overline{pc}_{\mathsf{sb}} = \overline{pc}'_{\mathsf{sb}}$
3) $\overline{br}_{\mathsf{sb}} = \overline{br}'_{\mathsf{sb}}$
4) $\mathit{\Gamma}_{\mathsf{sb}} = \mathit{\Gamma}'_{\mathsf{sb}}$
5) $time_{\mathsf{sb}} = time'_{\mathsf{sb}}$
6) $term_{\mathsf{sb}} = term'_{\mathsf{sb}}$
7) $block_{\mathsf{sb}} = block'_{\mathsf{sb}}$

Proof: By induction on the typing derivation. ■

Lemma 13 (Monotonicity of typing for surface commands)**.** *Given a com $\in$ **S**, $\mathit{\Gamma}$, lmdst, $\overline{pc}{\cdot}\ell$, $\overline{br}$, time, term, block and $\mathit{\Gamma}'$, lmdst$'$, $\overline{pc}{\cdot}\ell'$, $\overline{br}$, time, term$'$, block$'$ such that*

1) $lmdst$ ok $\mathit{\Gamma}$
2) $lmdst$ ok $\mathit{\Gamma}'$
3) $\mathit{\Gamma}, lmdst, \overline{pc}{\cdot}\ell, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}, \mathit{\Gamma}_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$
4) $time' \sqsubseteq time$
5) $term' \sqsubseteq term$
6) $block' \sqsubseteq block$
7) $\forall x \in Var.\ \mathit{\Gamma}'\langle x\rangle \sqsubseteq \mathit{\Gamma}\langle x\rangle$
8) $\ell' \sqsubseteq \ell$

then there are $\mathit{\Gamma}'_{\mathsf{sb}}, lmdst'_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}{\cdot}\ell', time'_{\mathsf{sb}}, term'_{\mathsf{sb}}, block'_{\mathsf{sb}}$ such that

1) $\mathit{\Gamma}', lmdst', \overline{pc}', \overline{br}, time', term', block' \vdash com : \ell'_{\mathsf{sb}}, \overline{pc}'_{\mathsf{sb}}, \overline{br}, \mathit{\Gamma}'_{\mathsf{sb}}, time'_{\mathsf{sb}}, term'_{\mathsf{sb}}, block'_{\mathsf{sb}}$
2) $time'_{\mathsf{sb}} \sqsubseteq time_{\mathsf{sb}}$
3) $term'_{\mathsf{sb}} \sqsubseteq term_{\mathsf{sb}}$

4) $block'_{\mathsf{sb}} \sqsubseteq block_{\mathsf{sb}}$
5) $\forall x \in Var.\ \Gamma'_{\mathsf{sb}}\langle x\rangle \sqsubseteq \Gamma_{\mathsf{sb}}\langle x\rangle$

Proof: By induction on the typing derivation. ■

Lemma 14 (Preservation of static bounds)**.** *Given $pool \in PSt$, $i \in pre(pool)$, $\sigma \in \Sigma$, $mem \in Mem$, $\tau \in Tr$, $gmon \in GMon$, and $pool' \in PSt$, $mem' \in Mem$, $\tau' \in Tr$, $gmon' \in GMon$ and Γ', $lmdst'$, $\overline{pc}'$, $\overline{br}'$, $time'$, $term'$, and $block'$ such that*

1) *$pool(i) = [com, mc, mdst]$ where $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$*
2) *$\langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle \twoheadrightarrow_\sigma \langle\!\langle pool', mem', \tau', gmon'\rangle\!\rangle$*
3) *$pool'(i) = [com', mc', mdst']$ where $mc' = \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle$*
4) *$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$*
5) *com does not satisfy conditions (1) and (2) in Definition 8.*

then

1) *$\Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block' \vdash com' : \ell'_{\mathsf{sb}}, \overline{pc}'_{\mathsf{sb}}, \overline{br}'_{\mathsf{sb}}, \Gamma'_{\mathsf{sb}}, time'_{\mathsf{sb}}, term'_{\mathsf{sb}}, block'_{\mathsf{sb}}$*
2) *$time'_{\mathsf{sb}} = time_{\mathsf{sb}}$*
3) *$term'_{\mathsf{sb}} = term_{\mathsf{sb}}$*
4) *$block'_{\mathsf{sb}} = block_{\mathsf{sb}}$*
5) *$\Gamma'_{\mathsf{sb}} = \Gamma_{\mathsf{sb}}$*

Proof: There are two possible inference rules that allow a global configuration to take a step. In one of them, all local configurations take a synchronization step. This does not satisfy assumptions of our Lemma, because barrier command cannot be typed according to the rules of Figures 10 and 11. Consider the other inference rule that allows the global configuration to take a step. There are two sub-cases here. In one of the sub-cases, the step is taken by some thread $j \neq i$, and therefore, $pool(i) = pool'(i)$; then we are done trivially. In the other sub-case, the i-th configuration takes a step:

$$\langle pool(i), mem, \tau\rangle \xrightarrow{\epsilon,\gamma,\epsilon}_\sigma \langle pool'(i), mem', \tau'\rangle$$

Consider the local configuration step and an associated local monitor step.

$$(com, mem, \tau) \overset{\alpha,\epsilon}{\twoheadrightarrow}_\sigma (com', mem', \tau')$$

and

$$\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle \longrightarrow^{\delta,\alpha}_{perm} \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle$$

We proceed by induction on the typing derivation

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

Case Static-Skip In this case, it must be that $com' = \mathsf{stop}$; the corresponding local monitor transition rule is (M-Skip); then we are done by inspecting rule (Static-Stop).

Case Static-Assign We have that $com' = \mathsf{stop}$. There are two possible local monitor transition rules: (M-Assign1) and (M-Assign2). In both cases, we observe that it holds that $time' = time$, $term' = term$, and $block' = block$. Moreover, the environment Γ' is obtained by updating Γ in both of the rules exactly as it is updated in (Static-Assign), namely by using $\ell = \Gamma\langle e\rangle \sqcup time \sqcup (\sqcup\overline{pc}) \sqcup term \sqcup block$. Then we are done by inspecting the rule (Static-Stop).

Case Static-Input Similar to (Static-Assign).

Case Static-Output Similar to (Static-Assign), but we only have one matching monitor rule.

Case Static-Seq By induction hypothesis.

Case Static-Branch Assume we take branch i (i = 1,2). In this case $com' = com_i; \mathsf{join}$. We have that $\alpha = \mathsf{b}(e, com_1, com_2)$ and the only matching monitor rule is (M-Branch). We have

$$\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle \longrightarrow^{\delta,\mathsf{b}(e,com_1,com_2)}_{perm} \langle \Gamma, lmdst, \overline{pc}\cdot\ell^*, \overline{br}\cdot(time^*, term^*, block^*, \Gamma^*), time, term, block\rangle$$

and

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi} : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

In this case, *-annotated environments are obtained from running the static oracle

$$\mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}, \Gamma, lmdst, \overline{pc}, time, term, block)$$

By Definition 8, we have three possible cases

1) $(\sqcup\overline{pc}) \sqcup \Gamma\langle e\rangle \sqcup time = \bot$. This contradicts condition (5) of our Lemma.
2) Does not match our command.

3) This is the only applicable case. We consider it below. In this case, by Definition 8 (Static bounds operator), it must be that

$$\Gamma, lmdst, \overline{pc}, \epsilon, time, term, block \vdash \mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi} : \ell^*, \overline{pc}, \epsilon, \Gamma^*, time^*, term^*, block^*$$

By Lemma 11 (Irrelevance of branching environments for surface commands), we have that

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi} : \ell^*, \overline{pc}, \overline{br}, \Gamma^*, time^*, term^*, block^*$$

By Lemma 12 (Static bound typing rules are deterministic), we have that

- $\ell_{\mathsf{sb}} = \ell^*$
- $\overline{pc}_{\mathsf{sb}} = \overline{pc}^*$
- $\overline{br}_{\mathsf{sb}} = \overline{br}^*$
- $\Gamma_{\mathsf{sb}} = \Gamma^*$
- $time_{\mathsf{sb}} = time^*$
- $term_{\mathsf{sb}} = term^*$
- $block_{\mathsf{sb}} = block^*$

By examining the rule (Static-Branch) we have that

$$\Gamma, lmdst, \overline{pc}{\cdot}\ell, \overline{br}{\cdot}(time_i, term_i, block_i, \Gamma_i), time, term, block' \vdash com_i : \\ \ell_i, \overline{pc}{\cdot}\ell, \overline{br}{\cdot}(time_i, term_i, block_i, \Gamma_i), \Gamma_i, time_i, term_i, block_i$$

where for all $y \in pre(\Gamma)$ it holds that $\Gamma_i(y) \sqsubseteq \Gamma^*$, $time_i \sqsubseteq time^*$, $term_i \sqsubseteq term^*$, and $block_i \sqsubseteq block_i$. By Lemma 11 (Irrelevance of branching environments for surface commands) we have that

$$\Gamma, lmdst, \overline{pc}{\cdot}\ell, \overline{br}{\cdot}(time^*, term^*, block^*, \Gamma^*), time, term, block' \vdash com_i : \\ \ell_i, \overline{pc}{\cdot}\ell, \overline{br}{\cdot}(time^*, term^*, block^*, \Gamma^*), \Gamma_i, time_i, term_i, block_i$$

From the typing rules (Static-Join) and (Static-Seq), we obtain that

$$\Gamma, lmdst, \overline{pc}{\cdot}\ell, \overline{br}{\cdot}(time^*, term^*, block^*, \Gamma^*), time, term, block' \vdash com_i; \mathsf{join} : \ell_i, \overline{pc}, \overline{br}, \Gamma^*, time^*, term^*, block^*$$

This concludes this case.

Case Static-Stop Not applicable, because there are no semantic transitions from stop.

Case Static-Join Immediate from the typing rule.

Case Static-Loop-Algorithmic We have that $com = \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}$, in this case $com' = \mathsf{more}\ e\ \mathsf{do}\ c\ \mathsf{od}$, and the only matching monitoring rule is (M-Enter). That is,

$$\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \rangle \longrightarrow^{\epsilon, \mathsf{enter}(e, com)}_{perm} \\ \langle \Gamma, lmdst, \overline{pc}{\cdot}\ell^*, \overline{br}{\cdot}(time^*, term^*, block^*, \Gamma^*), time, term, block \rangle$$

and

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

where *-environments are obtained from running the static bounds oracle:

$$(\ell^*, time^*, term^*, block^*, \Gamma^*) = \mathsf{SB}(\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}, \Gamma, lmdst, \overline{pc}, time, term, block)$$

By Definition 8 (Static bounds operator), it must be that

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} : \ell^*, \overline{pc}, \overline{br}, \Gamma^*, time^*, term^*, block^*$$

By Lemma 12 (Static bound typing rules are deterministic), we have that

- $\ell_{\mathsf{sb}} = \ell^*$
- $\overline{pc}_{\mathsf{sb}} = \overline{pc}^*$
- $\overline{br}_{\mathsf{sb}} = \overline{br}^*$
- $\Gamma_{\mathsf{sb}} = \Gamma^*$
- $time_{\mathsf{sb}} = time^*$
- $term_{\mathsf{sb}} = term^*$
- $block_{\mathsf{sb}} = block^*$

Then we are done by matching the premises of the rule (Static-More) with the premises of the rule (Static-Loop).

Case Static-More We have that $com = \mathsf{more}\ e\ \mathsf{do}\ c\ \mathsf{od}$. There are two possible semantic and monitor transitions.

1) (M-More). In this case we continue executing the loop. In this case, $com' = c; \mathsf{more}\ e\ \mathsf{do}\ c\ \mathsf{od}$.

$$\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle \longrightarrow_{perm}^{\epsilon, \mathsf{more}(e, com)} \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$$

and

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash \mathsf{more}\ e\ \mathsf{do}\ c\ \mathsf{od} : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

For clarity, we fold the reasoning about the timing, termination, and blocking environments below, into the reasoning about the typing environment. We have that there is an n such that

$$\begin{aligned} \Gamma \ldots &\vdash c : \ldots \Gamma''_0 \ldots \\ \Gamma''_0 \sqcup \Gamma \ldots &\vdash c : \ldots \Gamma''_1 \ldots \\ \Gamma''_1 \sqcup \Gamma \ldots &\vdash c : \ldots \Gamma''_2 \ldots \\ &\ldots \\ \Gamma''_{n-1} \sqcup \Gamma \ldots &\vdash c : \ldots \Gamma''_n \ldots \\ \Gamma''_n \sqcup \Gamma \ldots &\vdash c : \ldots \Gamma''_{n+1} \ldots \end{aligned}$$

where $\Gamma''_{n-1} \sqcup \Gamma = \Gamma''_n \sqcup \Gamma$ and $\Gamma''_n \sqcup \Gamma \sqsubseteq \Gamma_{\mathsf{sb}}$, where Γ_{sb} is the typing environment on the top of the branching environment stack $\overline{br}$. Note also that, by Lemma 13 (Monotonicity of typing for surface commands), it holds that $\Gamma''_i \sqsubseteq \Gamma''_{i+1}$, for $0 \le i \le n-1$, and by Lemma 12 (Static bound typing rules are deterministic), $\Gamma''_n = \Gamma''_{n+1}$.

Consider command $c; \mathsf{more}\ e\ \mathsf{do}\ c\ \mathsf{od}$. Because $c \in \mathbf{S}$, it does not change the branching or pc environments. By Lemma 12 (Static bound typing rules are deterministic), we have that

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash c : \ell''_0, \overline{pc}, \overline{br}, \Gamma''_0, term''_0, term''_0, block''_0$$

We want to show that

$$\Gamma''_0, lmdst, \overline{pc}, \overline{br}, time''_0, term''_0, block''_0 \vdash \mathsf{more}\ e\ \mathsf{do}\ c\ \mathsf{od} : \ell_{\mathsf{sb}}, \overline{pc}_{\mathsf{sb}}, \overline{br}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

Using Lemma 13 (Monotonicity of typing for surface commands) we have:

$$\Gamma''_0 \sqcup \Gamma \ldots \vdash c : \ldots \Gamma''_1 \ldots \implies \Gamma''_0 \ldots \vdash c : \ldots \Delta''_0 \ldots \text{ such that } \Delta''_0 \sqsubseteq \Gamma''_1$$

From this, using that $\Gamma''_0 \sqsubseteq \Gamma''_1$, we obtain that

$$\Gamma''_1 \sqcup \Gamma \ldots \vdash c : \ldots \Gamma''_2 \ldots \implies \Delta''_0 \sqcup \Gamma''_0 \ldots \vdash c : \ldots \Delta''_1 \ldots \text{ such that } \Delta''_1 \sqsubseteq \Gamma''_2$$

Subsequently, for all $i \ge 1$, we have

$$\Gamma''_i \sqcup \Gamma \ldots \vdash c : \ldots \Gamma''_{i+1} \ldots \implies \Delta''_{i-1} \sqcup \Gamma''_0 \ldots \vdash c : \ldots \Delta''_i \ldots \text{ such that } \Delta''_i \sqsubseteq \Gamma''_{i+1}$$

We observe that the sequence $\Delta''_0, \Delta''_1, \ldots, \Delta''_i, \ldots$ is monotonically increasing, and is bound from above by Γ''_n. Then, there must exist k such that $\Delta''_k = \Delta''_{k+1}$, and correspondingly, it must be that

$$\begin{aligned} \Gamma''_0 \ldots &\vdash c : \ldots \Delta''_0 \ldots \\ \Delta''_0 \sqcup \Gamma''_0 \ldots &\vdash c : \ldots \Delta''_1 \ldots \\ &\ldots \\ \Delta''_{k-1} \sqcup \Gamma''_0 \ldots &\vdash c : \ldots \Delta''_k \ldots \\ \Delta''_k \sqcup \Gamma''_0 \ldots &\vdash c : \ldots \Delta''_{k+1} \ldots \end{aligned}$$

where $\Delta''_{k-1} \sqcup \Gamma''_0 = \Delta''_k \sqcup \Gamma''_0$. Moreover, from $\Delta''_k \sqsubseteq \Gamma''_n$ and $\Gamma''_0 \sqsubseteq \Gamma''_n$, we obtain that and $\Delta''_k \sqcup \Gamma''_0 \sqsubseteq \Gamma''_n$. Hence, $\Delta''_k \sqcup \Gamma''_0 \sqsubseteq \Gamma_{\mathsf{sb}}$. The same reasoning follows for the timing, termination, and branching environments. This shows that we satisfy all the premises of (Static-More). Then we are done by (Static-Seq).

2) (M-Leave). In this case, we leave the loop, and $com' = \mathsf{stop}$. Then we are done by inspecting the rule (Static-Stop).

■

Lemma 15. *Given $pool \in PSt$, $i \in pre(pool)$, $\sigma \in \Sigma$, $mem \in Mem$, $\tau \in Tr$, $gmon \in GMon$, and $pool' \in PSt$, $mem' \in Mem$, $\tau' \in Tr$, $gmon' \in GMon$ and Γ', $lmdst'$, $\overline{pc}'$, $\overline{br}'$, $time'$, $term'$, and $block'$ such that*

- $pool(i) = [com, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
- $\langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle \twoheadrightarrow^*_\sigma \langle\!\langle pool', mem', \tau', gmon'\rangle\!\rangle$
- $pool'(i) = [\mathsf{stop}, \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle, mdst']$
- $\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$

we have:

- *Either* $\ell_{\mathsf{sb}} = \bot$ *or (* $time' \sqsubseteq time_{\mathsf{sb}}$ *and* $term' \sqsubseteq term_{\mathsf{sb}}$ *and* $block' \sqsubseteq block_{\mathsf{sb}}$ *and* $pre(\Gamma') = pre(\Gamma_{\mathsf{sb}})$ *and* $\forall x \in Var.\ \Gamma'\langle x\rangle \sqsubseteq \Gamma_{\mathsf{sb}}\langle x\rangle$*)*

Proof: By induction on the number of local steps. Base cases include all the transitions where the final configuration is reached in one step, and the result of the lemma is obtained by analyzing the associated monitor transitions. For the inductive cases, we use Lemma 14 to relate the static bounds of the given command with the bounds obtained through the induction hypothesis. ■

Lemma 16. *Given $com = \mathsf{while}\ e\ \mathsf{do}\ com_1\ \mathsf{od}$, $pool \in PSt$, $i \in pre(pool)$, $\sigma \in \Sigma$, $mem \in Mem$, $\tau \in Tr$, $gmon \in GMon$, and $pool' \in PSt$, $mem' \in Mem$, $\tau' \in Tr$, $gmon' \in GMon$, and Γ', $lmdst'$, $\overline{pc}'$, $\overline{br}'$, $time'$, $term'$, and $block'$ such that*

- $pool(i) = [\mathsf{while}\ e\ \mathsf{do}\ com_1\ \mathsf{od}, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
- $\langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle \twoheadrightarrow^*_\sigma \langle\!\langle pool', mem', \tau', gmon'\rangle\!\rangle$
- $pool'(i) = [\mathsf{more}\ e\ \mathsf{do}\ com_1\ \mathsf{od}, \langle \Gamma', lmdst', \overline{pc}', \overline{br}', time', term', block'\rangle, mdst']$
- $\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$

we have all of the following:

- $\Gamma'\langle e\rangle \sqcup time' \sqsubseteq \ell_{\mathsf{sb}}$.
- *Either* $\ell_{\mathsf{sb}} = \bot$ *or (* $time' \sqsubseteq time_{\mathsf{sb}}$ *and* $term' \sqsubseteq term_{\mathsf{sb}}$ *and* $block' \sqsubseteq block_{\mathsf{sb}}$ *and* $pre(\Gamma') = pre(\Gamma_{\mathsf{sb}})$ *and* $\forall x \in Var.\ \Gamma'\langle x\rangle \sqsubseteq \Gamma_{\mathsf{sb}}\langle x\rangle$*)*

Proof: By induction on the number of local steps using Lemma 14, similar to Lemma 15. ■

Lemma 17. *Given $pool \in PSt$, $i \in pre(pool)$, $\sigma \in \Sigma$, $mem \in Mem$, $\tau \in Tr$, $gmon \in GMon$ such that*

- $\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$
- $pool(i) = [com, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
- *execution of* $pool(i)$ *diverges.*
- $com \in \mathbf{S}$

we have:

- $(\sqcup\overline{pc}) \sqsubseteq term_{\mathsf{sb}}$.

Proof: By induction on the typing derivation. ■

Lemma 18. *Given $pool \in PSt$, $i \in pre(pool)$, $\sigma \in \Sigma$, $mem \in Mem$, $\tau \in Tr$, $gmon \in GMon$ such that*

- $\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$
- $pool(i) = [com, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
- *execution of* $pool(i)$ *blocks.*
- $com \in \mathbf{S}$

we have:

- $(\sqcup\overline{pc}) \sqsubseteq block_{\mathsf{sb}}$.

Proof: By induction on the typing derivation. ■

Lemma 19. *Given $pool \in PSt$, $i \in pre(pool)$, $\sigma \in \Sigma$, $mem \in Mem$, $\tau \in Tr$, $gmon \in GMon$ such that*

- $\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$
- $pool(i) = [com, mc, mdst]$ *where* $mc = \langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block\rangle$
- *execution of* $pool(i)$ *may fail to terminate normally because either* $pool(i)$ *diverges or the monitor for* $pool(i)$ *blocks*

we have:

- $\ell_{\mathsf{sb}} \sqsubseteq term_{\mathsf{sb}}$ *or* $\ell_{\mathsf{sb}} \sqsubseteq block_{\mathsf{sb}}$

Proof: We consider possible divergence and blocking behaviors separately.

Execution of *com* **diverges.** We proceed by induction on the typing derivation

$$\Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \vdash com : \ell_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}$$

Cases (Static-Skip, Static-Assign, Static-Input, Static-Output, Static-Seq, Static-Stop, Static-Join, Static-More). In all of these cases, $\ell_{\mathsf{sb}} = \bot$, and we are done trivially.

Case (Static-Branch) Follows from Lemma 17, because ℓ_{sb} is pushed on the pc-stack when typing either branches.

Case (Static-Loop-Algorithmic) Immediate from the typing rule.

Execution of com **is blocked by the monitor.** Similar to the previous case, using Lemma 18.

■

Theorem 14 (Soundness of static bounds function)**.** *Given program com, environment Γ, mode state lmdst, program counter stack* $\overline{pc}$*, and levels time, term and block, if* $\mathsf{SB}(com, \Gamma, lmdst, \overline{pc}, time, term, block) = (\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ *then* $(\ell_{\mathsf{sb}}, time_{\mathsf{sb}}, term_{\mathsf{sb}}, block_{\mathsf{sb}}, \Gamma_{\mathsf{sb}})$ *is a conservative approximation of the timing, termination, blocking, and floating behavior for com, Γ, lmdst,* $\overline{pc}$*,* $\overline{br}$*, time, term, and block.*

Proof: We proceed by analyzing the cases in Definition 8.

1) com is $\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}$ such that $(\sqcup\overline{pc}) \sqcup \Gamma\langle e\rangle \sqcup time = \bot$. We have that $\ell_{\mathsf{sb}} = \bot$, $time_{\mathsf{sb}} = \bot$, $term_{\mathsf{sb}} = \bot$, $block_{\mathsf{sb}} = \bot$, and $\Gamma_{\mathsf{sb}} = \Gamma$.
   We examine all of the requirements imposed by Definition 2. Requirement (1) is trivial. Requirements (2) and (3) follow by assumption. Requirements (4) - (6) are trivial. Requirement (7) follows from the assumption in the current case. Requirement (8) is not applicable.
2) com is $\mathsf{while}\ e\ \mathsf{do}\ com\ \mathsf{od}$ such that $\forall x \in e.\ x \in NonFloatVar$ and $(\sqcup\overline{pc}) \sqcup \Gamma\langle e\rangle \sqcup time = \bot$.
   Similar to the above, with the only difference that Requirement (7) is not applicable, and Requirement (8) follows from the assumption in the current case.
3) None of the above applies. We examine all of the requirements imposed by Definition 2.
   - Requirements (1) – (3) follow from Definition 8 and Lemma 9.
   - Requirements (4) and (8) follow from Lemma 15 and 16 respectively.
   - Requirement (5) follows from Lemma 19.
   - Requirement (6) follows from analysis of the rules in Figures 10–12, and observing that they contain no judgment for the barrier command.
   - Requirement (7) is straightforward by analyzing rule (Static-Branch).

■