8-7-2014

Panini: A Concurrent Programming Model for Solving Pervasive & Oblivious Interference

Mehdi Bagherzadeh
Iowa State University, mbagherz@iastate.edu

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Panini: A Concurrent Programming Model for Solving Pervasive & Oblivious Interference

Abstract
Modular reasoning about concurrent programs is complicated by the pervasiveness of interferences that do not give out any information about the behaviors of potentially interfering concurrent tasks, at the interference points. Reasoning about a concurrent program would be easier if a programmer, modularly and statically: (1) knows precisely the program points at which the interference may happen, and (2) has some insights into the behaviors of other potentially concurrent tasks at these points. In this work, we present a core concurrent calculus with these properties and their proofs. Using our calculus a programmer can design and implement concurrent modules of their software using capsules and modularly reason about properties of these concurrent modules using static types referred to in their code, and then automatically have these properties hold in the larger concurrent system formed by composing these capsules. We also illustrate how this concurrent calculus can be used in conjunction with standard Hoare style reasoning to modularly verify concurrent programs with interference.

Keywords
Modular reasoning, pervasive interference, oblivious interference, sparse interference, cognizant interference, concurrency, capsule-oriented programming

Disciplines
Programming Languages and Compilers

Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference

Mehdi Bagherzadeh Hridesh Rajan

TR #14-09
Initial Submission: August 07, 2014
Revised: October 10, 2014

Keywords: concurrency, modular specification, modular reasoning, pervasive interference, oblivious interference, sparse interference, cognizant interference, interference-free behavioral contracts, Panini

CR Categories:
D.2.10 [*Software Engineering*] Design
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference

Mehdi Bagherzadeh Hridesh Rajan
Iowa State University, Ames, IA
{mbagherz,hridesh}@iastate.edu

Abstract

Modular reasoning about concurrent programs is complicated by the possibility of interferences happening between any two instructions of a task (*pervasive interference*), and these interferences not giving out any information about the behaviors of potentially interfering concurrent tasks (*oblivious interference*). Reasoning about a concurrent program would be easier if a programmer, modularly and statically *(1)* knows precisely the program points at which interferences may happen (*sparse interference*), and *(2)* has some insights into behaviors of potentially interfering tasks at these points (*cognizant interference*). In this work we present *Panini*, a core concurrent calculus which guarantees sparse interference, by controlling sharing among concurrent tasks, and cognizant interference, by controlling dynamic name bindings and accessibility of states of tasks. *Panini* promotes capsule-oriented programming whose concurrently running capsules own their states, communicate by asynchronous invocations of their procedures and dynamically transfer ownership. *Panini* limits sharing among two capsules to other capsules and futures, limits accessibility of a capsule states to only through its procedures and dispatches a procedure invocation on the static type of its receiver capsule. We formalize *Panini*, present its semantics and illustrate how its interference model, using behavioral contracts, enables Hoare-style modular reasoning about concurrent programs with interference.

1. Introduction

Modular reasoning is important for scalable software development, because it allows programmers and tools [19, 40] to discharge verification obligations about a module by just considering the implementation of the module and the interfaces (not implementations) of *static types* named in that module. Concurrent programming, i.e. the ability to perform two or more simultaneous computations in a single program, is also vital for creating modern software systems. We believe that concurrent programming stubbornly remains difficult and error-prone because we cannot, yet, do modular reasoning about concurrent programs.

1.1 Problem

There are two fundamental obstacles to modular reasoning about concurrent programs [25, 40, 49]:

❶ the *pervasive interference* problem, and

❷ the *oblivious interference* problem.

By the *pervasive interference* we mean that in a concurrent program, between any two consecutive instructions of a concurrent task, interleaving instructions of another task could change the state of program such that it influences the subsequent behavior of the first task [49]. This in turn means, there are too many points in a program, that a programmer must analyze before they can understand the behavior of their program. For example, in the single straight line code below there could be three interference points that occur between instructions that read the value of `x`, read the value of `y`, add the two values, and write the value of `x`.

```
x = x + y;
```

That is, this code actually looks like $^{\alpha}x = {}^{\alpha}x + {}^{\alpha}y$ where $^{\alpha}$ denotes an interference point.

By the *oblivious interference* we mean that interference points do not give out any information, either concrete or abstract, about what other concurrent tasks may interfere or what are their behaviors. This in turn means, a programmer must consider all potentially concurrent tasks to determine whether their interleavings would be harmful and cause interferences (global reasoning). For example, in the straight line code above, there is no information about how interfering tasks at interference points $^{\alpha}$ may or may not modify values of `x` and `y`.

The key difference between pervasive and oblivious interference is that the former is about locations of interferences (where) whereas the latter is concerned about behaviors of interferences (what). Though, these two are well-known concurrency problems we coin the terms "pervasive interference" and "oblivious interference" to refer to them.

Pervasive and oblivious interference together, magnify the challenges of modular reasoning about concurrent programs [19, 25, 40, 48, 49]. Several prior techniques have been proposed for controlling interference and interfering behaviors of concurrent programs. Figure 1 summarizes the most related work and compares them regarding pervasiveness and obliviousness of interferences. §7 details the comparison of previous work regarding pervasive and oblivious interference for all related works.

Atomicity, transactional memory, cooperability and automatic mutual exclusion (AME) An atomic block [13, 17, 45] is a code block which is free of interference, i.e. the code in the block behaves as if it executes sequentially. Atomicity limits the interference points to the code outside atomic blocks, however, for the code outside an atomic block interference could still happen between any two instructions. Atomic blocks do not say anything about behaviors of interfering tasks at interference points.

Cooperability [43, 48, 49] and automatic mutual exclusion [1, 23, 42] are the inverse of atomic blocks, i.e. the code is atomic and

	Interference	
	Pervasive	Oblivious
atomicity, transactional memory, cooperability , AME	✗	✓
actors, active objects	✗	✓
rely-guarantee, Owicki-Gries	✓	✗

Figure 1. Comparison of previous work regarding pervasive and oblivious interference problems. ✗ and ✓ mark absence and presence of the problems, respectively.

interference-free unless explicitly specified. These techniques limit interferences to explicitly specified interference points but similarly do not say anything about behaviors of interfering tasks.

Actors and active objects Actors [3–6] encapsulate data and control and communicate by passing messages. A class of actor models (similarly active objects [10, 41]) in which actors provide confinement and do not permit unfettered internal concurrency limits its interference points to message sends or receives, i.e. a block of code between two message receives are atomic (macro-step semantics) [5]. However, actor models that allow uncontrolled internal concurrency or arbitrary data sharing among actors still could have interference between any two instructions. Actor models do not solve the oblivious interference problem because their dynamic name bindings could make the exact static type of a receiver or sender of a message unknown. Thus, a programmer cannot tell which concurrent tasks are interfering at interference points

Rely-guarantee and Owicki-Gries Rely-guarantee based reasoning approaches [19, 25] and Owicki-Gries's work [35] specify the behavior of the environment (other concurrently running tasks) of a task and thus limits the interference behavior. However, interferences can still happen between any two instructions.

1.2 Solution: *Panini*

A concurrent programming model can enable more modular reasoning if it provides the following properties to a programmer:

❶ *Sparse interference*: interference points are not pervasive in a program, instead they can be explicitly identified by certain program constructs; and

❷ *Cognizant interference*: interference points are not oblivious in the program, instead each explicitly identified interference point provides an abstraction of behaviors of all potentially interfering concurrent tasks.

The language *Panini* presents such a programming model called *capsule-oriented programming* [37]. Before discussing *Panini*'s interference model and semantics, we provide a gentle introduction using the example in Figure 2. The example, declares a capsule type Counter, with a single state x, on line 2, and the procedure add, on lines 3–5. The body of add contains the straight line code shown in the previous section, which adds y to x. A capsule instance of Counter, say c, can be declared and it behaves like a process.

```
1  capsule Counter {
2    int x;
3    void add( int y ) {
4      x = x + y;
5    }
6  }
```

Figure 2. Capsule Counter with state x and procedure add.

In *Panini*, a program is a set of concurrently running capsule instances which own their states and communicate with each other through asynchronous procedure invocations. Invocation of a capsule instance procedure, appends the invoked procedure to the tail of the instance's queue and returns a future [46] for its result. The single execution thread of a capsule instance dequeues its invoked procedures from the head of the queue and executes them in the order they appear in the queue. Invocation and returning of a procedure transfers the ownership of its parameters and return values between its invoking and invoked capsule instances, respectively.

1.2.1 Sparse Interference, to Solve Pervasive Interference

Panini's semantics controls and limits sharing among two capsule instances to other capsule instances and futures. This in turn allow a *Panini* program to limit its interference points to after capsule procedure invocations and guarantee sparse interference. As an example, there are no interferences in the body of the procedure add of a capsule instance c of type Counter, which is in contrast with straight line code in the previous section with pervasive interference and possible interferences between any two instructions. This is because *Panini*'s semantics guarantees that the state x is owned and is only accessible to its enclosing capsule instance c and there is only one thread of execution running c and accessing x. This in turn allows interferences in the code for the procedure add to be safely swapped out [30], as in the code runs with no interferences.

A reader familiar with synchronization features in languages like Java could perhaps achieve the same by implementing the counter as a class and marking its add method as **synchronized**, however, the capsule model saves the programmer from worrying about if there still could be interferences on line 4 and whether acquiring an object-level lock is actually sufficient to *protect* x. A lock protects a memory location, if throughout a program, every access to the location is preceded by acquiring the lock [17].

```
1  capsule Counter {
2    Number x;
3    void add( Number y ) {
4      x.add( y.value() );
5    }
6    Number value() { return x.value(); }
7  }
```

Figure 3. Capsule Counter with a reference type state.

To illustrate *Panini*'s semantics more, consider the capsule Counter in Figure 3 which creates the possibility of sharing the state x and the capsule procedure argument y, among an instance of Counter and other capsule instances, by changing their types from integer to a reference type Number.

Again, the body of the procedure add of a capsule instance c does not have any interferences. This is because *Panini*'s semantics guarantees that not only the state x but also its *representation* [9], i.e. the object graph rooted at x, is owned by the instance c and is only accessible from within c itself. The semantics also guarantees that upon invocation of the procedure add and throughout its execution, its argument y and its representation is owned by c. This in turn allows interferences in the code for add to be safely swapped out, as in the code runs with no interferences.

Note that the alternative using **synchronized** or locks works only if proper locks are put in place, to guarantee that they protect not only the state x and its representation but also the parameter y and its representation [17]. *Panini*'s semantics does not allow lock splitting, i.e. protecting fields in a capsule by different locks.

So far we have looked at capsules with no interference points in the bodies of their procedures. To illustrate capsules with interference points consider the capsule Client in Figure 4. This capsule imports an external capsule instance c of type Counter, on line 1, that a capsule instance of type Client can interact with. The body of the procedure test of Client contains asynchronous invocations of procedures value and add on the receiver capsule instance c, on lines 4–6. Asynchronous invocation of value, on line 4, appends the

body of the procedure to the end of the queue for c and immediately returns the unresolved future oldVal without waiting for the execution of value. The future is resolved and ready whenever the capsule instance c dequeues the body of value from its queue and executes it. Any attempt to access an unresolved future, e.g. on line 7, blocks until the future is resolved. Lines 5–6, append the queue of c with invocations of procedures add and value, respectively. The capsule instance c dequeues its invoked procedures from its queue in the same order they appear in the queue and executes them, i.e. first in first out (FIFO) order. For invocations of value and add, on lines 5–6, execution of value starts after the execution of add finishes.

```
1   capsule Client ( Counter c ) {
2     Number oldVal, newVal;
3     void test( Number y ) {
4       oldVal = c.value();
5       c.add( y );
6       newVal = c.value();
7       //@ assert newVal >= oldVal; Φ
8     }
9   }
```

Figure 4. Capsule Client with JML-like assertion Φ on line 7.

There are three interference points in the body of test, one right after each asynchronous invocation of procedures value and add, on lines 4–6. This is because, *Panini*'s semantics guarantees that the imported capsule instance c is the only shared resource among a capsule instance of type Client and other capsule instances in the system, with unsynchronized access. Other shared resources are futures oldVal and newVal, however, their accesses are synchronized and do not cause interference [47]. *Panini*'s semantics also guarantees that upon invocation of add, on line 5, the ownership of its parameter y is transferred to c, and the body of test does not access y after its ownership is transferred.

As figures 2, 3 and 4 illustrate, unlike pervasive interference in which interferences can happen between any two instructions in a program, *Panini*'s semantics guarantees that in a *Panini* program, potential interference points are explicitly identified and are limited to after asynchronous procedure invocations.

1.2.2 Cognizant Interference, to Solve Oblivious Interference

Panini also control and limits accessibility of states of a capsule instance to only through its procedures and dispatches procedure invocations using the static type of their receiver capsule instances. This in turn, allow a *Panini* program to limit the interference behavior at an interference point to the Kleene closure of procedures of the static type of the receiver of the procedure invocation, and guarantee cognizant interference. The Kleene closure of a set of procedures, contains the empty set and any concurrent composition of any number of procedures in the set.

For example, the interfering behavior at the interference point after the invocation of the procedure value on capsule instance c, on line 3, is contained in the Kleene closure $\theta = \{c.value(), c.add(\_)\}^*$. This is because *Panini*'s semantics guarantees that the imported capsule instance c is the only shared resource with unsynchronized access between an instance of Client and other capsule instance in a system. *Panini* semantics also guarantees that state of the capsule instance c is only accessible through its procedures add and value. Finally *Panini*'s semantics guarantees that the invocation of value, at the interference point on line 3, is dispatched using the static type of its receiver capsule instance c and not its subtype capsules which may have more procedures than add and value.

To interfere with c, other concurrently running capsule instances in the system can change the state of c by invoking its two procedures add and value any number of times and in any order which basically is the same as the closure θ. The closure θ is a closure of 0 or more, concurrently running, invocations of *all* procedures of the capsule instance c. The Kleene closure for value and add includes empty set $\emptyset$ and is closed under the concurrent composition and execution operation $||$. For example, $c.value()$, $c.value() \;||\; c.value()$, $c.add(\_)$ and $c.value() \;||\; c.add(\_)$ are few elements of θ.

As Figure 4 illustrates, unlike oblivious interference in which the interference behavior is unknown, *Panini* semantics guarantees that interference behavior for an interference point after an asynchronous procedure invocation, is limited to the Kleene closure of procedures of the static type of the receiver of the invocation.

1.2.3 Modular Reasoning Using *Panini*'s Interference Model

Panini's sparse and cognizant interference in turn enable static modular reasoning about its concurrent programs. To modularly understand a module in a *Panini* program *(1)* using sparse interference, the interference points of the module can be identified *syntactically* by only considering the implementation of the module, since interference points are explicitly identified by asynchronous procedure invocations; *(2)* using cognizant interference, the interfering behaviors at interference points can be identified *statically* by just considering the interfaces of static types of receivers of procedure invocations at these interference points, since interfering behaviors are Kleene closures of procedures of receivers of procedure invocations; *(3)* finally, for each interference point identified in *(1)* its interfering behavior identified in *(2)* could be inserted at the interference point in the module to arrive at a result module that takes interference and its behavior into account. Such a module could be modularly understood [19] using Hoare-style [21] reasoning.

For example, consider static verification of the assertion Φ, on line 7 of Figure 4. In this example, sparse interference limits the interference points to after asynchronous invocations of procedures value and add, 4- 6, and cognizant interference limits the interference behavior at these interference points to the Kleene closure $\theta = \{c.value(), c.add(\_)\}^*$. The assertion Φ could be modularly verified after inserting the interfering behavior θ at each interference point in the procedure test using Hoare-style [21] reasoning.

Such reasoning is modular because the programmer only needs to consider the implementation of the procedure test and the interface of the static types it refers to, i.e. capsule type Counter. Identification of interference points and interfering behavior in test is similarly modular. The behaviors of procedures value and add say that the former does not change the value of the counter whereas the latter only increases it. Behaviors of these procedures are specified by their behavioral contracts, illustrated in §5.

1.2.4 Contributions

In summary, the contributions of this work are the following:

- Formalization of *Panini*'s core calculus and its semantics; and
- Proving sparse and cognizant properties of *Panini*'s interference model; and
- Illustration of modular Hoare-style reasoning using behavioral contracts for concurrent *Panini* programs with interference.

1.2.5 Paper Outline

§2 discusses capsule-oriented programming in *Panini* by presenting its syntax. §3 formalizes *Panini*'s dynamic semantics and §4 proves its sparse and cognizant interference model and their underlying properties. §5 illustrates Hoare-style reasoning in *Panini*, using behavioral contracts in the presence of interference. §6 briefly discusses *Panini*'s expressiveness and usability and Kleene closure analysis. §7 presents related work, and §8 concludes.

$prog ::= \overline{decl}$	program
$decl ::=$ **capsule** $C\ (\overline{imp})\ \{$ $design\ \overline{state}\ \overline{proc}\ \}$	capsule declaration
$imp ::= D\ i$	import
$state ::= T\ f$ **;**	state declaration
$proc ::= T\ p\ (\ \overline{form}\ )\ \{\ e^{\ \alpha}\}$	procedure declaration
$form ::= T\ \ var$	formal
$design ::=$ **design** $\{\ \overline{ins}\ \ \overline{wire}\ \}$	design declaration
$ins ::= C\ \ i$ **;**	instance declaration
$wire ::= i\ (\overline{j})$ **;**	wiring declaration
$e ::=$	expression
$\mid i.p\ (\overline{e})^{\ \alpha}$	global procedure invocation
$\mid$ **self**$.p\ (\overline{e})$	local procedure invocation
$\mid$ **self**$.f$	state read
$\mid$ **self**$.f$ **:=** e	state assignment
$\mid$ **ref** e	reference
$\mid\ !\ e$	dereference
$\mid e$ **:=** e	reference assignment
$\mid$ **let** $x = e$ **in** e	let
$\mid ()$	unit value

$C, D \in \mathfrak{C}$	set of capsule names
$T \in \mathfrak{T}$	set of variable types
$f \in \mathfrak{F}$	set of state names
$p \in run \cup \mathfrak{P}$	set of procedure names
$x,var \in \mathfrak{X}$	set of variable names
$i,j,h \in \mathfrak{I}$	set of capsule instance names
$\alpha, \beta \in \mathfrak{B}$	set of labels

Figure 5. *Panini*'s core syntax, based on [37].

2. *Panini*'s Syntax

Figure 5 shows *Panini*'s expression-based core syntax. In this figure, the superscript $\overline{term}$ shows a sequence of zero or more terms. A *Panini* program is a set of capsule declarations. A capsule declaration contains a capsule name C and declares a set of imported capsule instances $\overline{imp}$, a design *design*, a set of capsule states $\overline{state}$ and capsule procedures $\overline{proc}$. A capsule instance, can interact and invoke procedures of two kinds of other capsule instances: imported and locally declared. An import declaration declares an imported capsule instance by specifying its capsule type D and capsule name i. A local design declaration declares a set of local capsule instances $\overline{ins}$ and specifies their connections together in a wiring declaration *wire*. A wiring declaration $i(\overline{j})$ assigns capsule instances $\overline{j}$ to the imported capsule instances of the capsule i. Local capsule instances of a local capsule's design are not accessible to other capsules.

Capsule instances of a *Panini* program interact *only* by invoking procedures of each other. A procedure declaration of a capsule has a variable return type T, a name p, set of formal parameters $\overline{form}$ and a body e. Body of a capsule procedure is a sequence of global and local expressions. Using global expressions, a procedure can *asynchronously* invoke a procedure of another capsule instance. However, using local expressions, a procedure of a capsule can *synchronously* invoke another procedure of the same capsule, access the state of the capsule through `self`, or allocate and access memory locations. Labels $^{\alpha}$ denote possible interference points after asynchronous procedure invocations. The sequence of expressions $e_1; e_2$ is a syntactic sugar for the let expression **let** $x = e_1$ **in** e_2 in which variable x is free in e_2.

Panini's type system, in §A, distinguishes capsule types C from variable types T. Unlike variable types, capsule types *cannot subtype each* other and thus their exact types are statically known. This in turn enables statically-bound procedure invocations in which the exact type of the receiver of a capsule is statically known.

To illustrate, Figure 4 declares a capsule type Client with the imported capsule c of capsule type Counter and a procedure test with formal parameter y of variable type Number. The procedure body is a sequence of three asynchronous invocations of procedures value and add which are statically dispatched on the imported capsule instance c. As another example, the capsule type Main in Figure 6, declares a design declaration on lines 2–6 that includes declarations of two capsule instances client and counter of types Client and Counter, respectively. The design declaration contains a wiring declaration on line 5 that connects the client and counter instances by passing counter as the client's imported capsule instance.

```
1  capsule Main() {
2    design {
3      Counter counter;     // a counter
4      Client  client ;     // a client
5      client (counter);    // a wiring
6    }
7    void run() { client . test () ; }
8  }
```

Figure 6. Design and wiring declarations in capsule Main.

3. Operational Semantics

In *Panini*'s operational (dynamic) semantics each concurrently running capsule instance owns its states and their representations, i.e. reference graphs reachable from its states; dynamically transfers ownership of parameters and return values of its global procedure invocations; and uses only one thread of execution to dequeue and execute its invoked procedures. These in turn result in the following properties of a *Panini* programs that are critical to its sparse and cognizant interference model:

1. sharing among two capsule instances is limited to their imported capsule instances and unresolved future locations;
2. states of a capsule instance and their representations are only accessible through its procedures.

Panini's interference model and its underlying properties are formalized in §4. *Panini*'s type system is formalized in §A.

3.1 Dynamic Objects

Panini's dynamic semantics relies on three additional expressions l, **resolve** and **OWE**, shown in Figure 7, that are not part of its surface syntax. The expression l represents a memory location in the store. The expression $\textbf{resolve}(l, e, id, p)$ returns the result of the asynchronous invocation of procedure p with body e into future location l in the invoking capsule instance id. The ownership transfer exception **OWE** denotes accessing a transferred location that a capsule instance no longer owns or transferring a location in representation of a capsule's state.

Panini's operational semantics transitions from one global (program) configuration to another, as shown in Figure 7. A global configuration $\mathscr{K}$ is a concurrent composition $||$ of capsule instance configurations Σ [1]. A capsule configuration Σ contains a *unique* capsule identifier id, a queue Q with an expression e under evaluation at its head, a local store S, a capsule record r and an instance mapping I. A queue is a possibly empty queue of expressions e. The local store is a mapping from locations l accessible to the capsule instance to their values v. The capsule record contains the static capsule type C of the instance and a state mapping F from capsule fields f to locations. The instance mapping, maps the names of imported and locally declared capsule instances i to their identifiers id. A capsule configuration also includes P which contains capsule declarations, similar to the class table in Featherweight Java, and is

[1] Concurrent composition $||$ is commutative, i.e. $\Sigma \,||\, \Sigma'$ is equal to $\Sigma' \,||\, \Sigma$.

Added syntax:

$e ::= ..$

$\mid l$	memory location
$\mid \texttt{resolve}(l, e, id, p)$	resolve a future location
$\mid \texttt{OWE}$	ownership transfer exception

Evaluation contexts:

$\mathscr{E} ::= - \mid i.p(v..\mathscr{E}\,e..) \mid \texttt{self}.p(v..\mathscr{E}\,e..)$
$\quad \mid \texttt{self}.f := \mathscr{E} \mid \texttt{ref}\,\mathscr{E} \mid !\mathscr{E}$
$\quad \mid \mathscr{E} := e \mid \texttt{let}\ x = \mathscr{E}\ \texttt{in}\ e$

Evaluation relations $\overset{a}{\hookrightarrow}$ and $\overset{a}{\rightsquigarrow}$: $\mathscr{K} \overset{a}{\hookrightarrow} \mathscr{K}'$ and $\Sigma \overset{a}{\rightsquigarrow} \Sigma'$

Domains:

$\mathscr{K} ::= \bullet \mid \Sigma \mid\mid \mathscr{K}$	global (program) configurations
$\Sigma ::= \langle P, id, e.Q, S, r, I \rangle$	capsule instance configurations
$r ::= [C.F]$	capsule records
$F ::= \{f_k \mapsto l_k\}$	state maps
$Q ::= \bullet \mid e.Q$	queues
$S ::= \{l_k \mapsto v_k\}$	local stores
$I ::= \{i_k \mapsto id_k\}$	instance maps
$v ::=$	values
$\quad l$	location values
$\quad \mid \varepsilon$	unresolved future values
$\quad \mid \square$	transferred location values

Actions:

$a ::=$

$\mid read(id, l)$	read
$\mid write(id, l)$	write
$\mid invoke(id, id', p, l)$	invoke
$\mid resolve(id, id', p, l)$	resolve
$\mid local(id)$	local

$l \in \mathfrak{L}$	set of locations
$id, id' \in \mathfrak{N}$	set of capsule identifiers
$k \in \mathfrak{K}$	is finite
P	program declarations

Figure 7. Added syntax, evaluation contexts, configurations and actions in *Panini*'s semantics.

similarly used to look up declarations and procedure bodies when invoking a procedure.

In *Panini*, a value can be a location l. A value can also be an unresolved future value ε that denotes the result of an asynchronous procedure invocation before it is ready; or it can be a transferred value $\square$ denoting the value of a location whose ownership has been transferred and no longer is accessible to a capsule instance.

Panini uses a left-most inner-most call-by-value evaluation policy. Evaluation contexts, in Figure 7, specify the evaluation order of an expression and the evaluation position in the expression.

Execution of a *Panini* program produces a trace of observable actions. Actions are basic units of execution and each action represents execution of a single *indivisible* (atomic) instruction. Figure 7 shows a core set of actions observed during execution of a *Panini* program. An action can be: a read or write of a memory location l by a capsule instance id; asynchronous invocation of a procedure p of another capsule id' with the future result l; resolving the result of an asynchronous procedure invocation into the future location l; or it can be a local action of a capsule id, such as invocation of a synchronous procedure of the capsule or dequeuing an expression from its queue.

3.2 Local and Global Semantics

Panini's operational semantics has two sets of evaluation rules for its local and global semantics. A local evaluation $\overset{a}{\rightsquigarrow}$ denotes transition from a capsule configuration to another performing the action a. A local transition in turn causes a global transition $\overset{a}{\hookrightarrow}$ from a program configuration to another in which capsule instances run concurrently. A *preemptive scheduler nondeterministically* chooses a capsule instances for evaluation at each point in time.

Figure 8 shows *Panini*'s substitution-based operational semantics for normal execution, i.e. no exceptions thrown. Figure 10 shows its exceptional operational semantics.

3.2.1 Sequential, Synchronous Local Semantics

Local evaluation relation $\overset{a}{\rightsquigarrow}$ in a capsule instance denotes evaluation of an expression e at the head of its queue to another expression e' and performing the action a. This evaluation causes transition from a capsule configuration to another with a possibly modified queue and local store [2]. In local semantics, a capsule instance can access its state through `self`, allocate and access memory locations, invoke a procedure of itself or dequeue an expression in its queue. Local evaluation is synchronous and sequential, i.e., a capsule instance only has one execution thread.

A capsule can read and write its state through the variable `self` in the rules (STATE READ) and (STATE ASSIGN). A capsule state is accessible through state mapping F and local store S. A capsule's state name is mapped to a location in F and then there is a mapping between the state location and its value in S. To read the value of a state in (STATE READ), the notation $F[f = l]$ checks if the field name f maps to a location l and the notation $S[l = v]$ checks if the value of the location in the local store is equal to v and if so returns v. Reading a state stored at the location l by a capsule instance id causes a $read(id, l)$ action in the execution trace of the program. To assign a value to a state in (STATE ASSIGN), the notation $S[l := v]$ [3] replaces the old value of the location l with its new value v, such that the rest of S stays intact. Similar to read, writing a state stored at the location l by capsule instance id causes a $write(id, l)$ action.

A capsule can also create a reference, dereference it and assign to it in the rules (REF), (DEREF) and (REF ASSIGN). To create a new reference with a value v in (REF), $fresh(l)$ returns a fresh location which then is mapped to its value in the local store of the capsule. A fresh location is a location that does not belong to the local store of any other capsule instance in the program, as shown in Figure 9. By mapping the newly allocated location l to its value in the local store S, the rule (REF) makes the capsule instance id the owner of the location l, as well. To dereference a location l in (DEREF), its value is retrieved from the store *unless the value is equal to the unresolved future value* ε. Trying to dereference an unresolved future value causes the capsule instance to *block*. The capsule instance unblocks and can continue execution when the value of the future is resolved, i.e. is not equal to ε anymore. The blocking condition $v \neq \varepsilon$ in (DEREF) *synchronizes* access to unresolved future locations and does not allow concurrent access to them. To assign to a reference in (REF ASSIGN), its value is simply updated in the local store of the capsule. Again, trying to assign to an unresolved future location causes the capsule instance to block. Evaluation rules (REF), (DEREF) and (REF ASSIGN) perform their corresponding write, read and write actions, respectively.

A reference location manipulated by these rules resides in a capsule's local store unless its ownership is transferred to other capsules via procedure invocations. In other words, a capsule instance cannot access locations in other capsule instances if their ownership is not transferred to the capsule through procedure invocations.

A capsule instance can *synchronously* invoke a procedure of itself in (SELF PROC INVOC). Invocation of a local procedure causes the body of the procedure to replace the procedure invocation, after

[2] Evaluation of a configuration does not change its mapping I and record r.

[3] Notation $S[l = v]$ does not modify S whereas $S[l := v]$ does.

$$\boxed{\textbf{\textit{Local evaluation relation}}\ \overset{a}{\rightsquigarrow}: \quad \langle P,id,\mathscr{E}[e].Q,S,r,I\rangle \overset{a}{\rightsquigarrow} \langle P,id,\mathscr{E}[e'].Q',S',r,I\rangle}$$

(STATE READ)

$$\langle P,id,\mathscr{E}[\texttt{self}.f].Q,S[l=v],[C.F[f=l]],I\rangle \overset{read(id,l)}{\rightsquigarrow} \langle P,id,\mathscr{E}[v].Q,S,[C.F],I\rangle$$

(STATE ASSIGN)

$$\langle P,id,\mathscr{E}[\texttt{self}.f := v].Q,S,[C.F[f=l]],I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P,id,\mathscr{E}[v].Q,S[l:=v],[C.F],I\rangle$$

(SELF PROC INVOC)

$$\left\langle P[\texttt{capsule}\ C(..)\{..\ T\ p(\overline{T\ var})\{e\}\ ..\}],id,\mathscr{E}[\texttt{self}.p(\bar{v})].Q,S,[C.F],I\right\rangle \overset{local(id)}{\rightsquigarrow} \langle P,id,\mathscr{E}[e\ [\bar{v}/\overline{var}]].Q,S,[C.F],I\rangle$$

(REF)

$$\frac{fresh(l)}{\langle P,id,\mathscr{E}[\texttt{ref}\ v].Q,S,r,I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P,id,\mathscr{E}[l].Q,S[l:=v],r,I\rangle}$$

(DEREF)

$$\frac{v \neq \varepsilon}{\langle P,id,\mathscr{E}[!\ l].Q,S[l=v],r,I\rangle \overset{read(id,l)}{\rightsquigarrow} \langle P,id,\mathscr{E}[v].Q,S,r,I\rangle}$$

(REF ASSIGN)

$$\langle P,id,\mathscr{E}[l:=v].Q,S[l\neq\varepsilon],r,I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P,id,\mathscr{E}[v].Q,S[l:=v],r,I\rangle$$

(LET BINDING)

$$\langle P,id,\mathscr{E}[\texttt{let}\ x=v\ \texttt{in}\ e].Q,S,r,I\rangle \overset{local(id)}{\rightsquigarrow} \langle P,id,\mathscr{E}[e[v/x]].Q,S,r,I\rangle$$

(FIFO DEQUEUE)

$$\langle P,id,v.e.Q,S,r,I\rangle \overset{local(id)}{\rightsquigarrow} \langle P,id,e.Q,S,r,I\rangle$$

$$\boxed{\textbf{\textit{Global evaluation relation}}\ \overset{a}{\hookrightarrow}: \quad \mathscr{K}\ ||\ \langle P,id,\mathscr{E}[e].Q,S,r,I\rangle \overset{a}{\hookrightarrow} \mathscr{K}'\ ||\ \langle P,id,\mathscr{E}[e'].Q',S',r,I\rangle}$$

(CONGRUENCE)

$$\frac{\langle P,id,\mathscr{E}[e].Q,S,r,I\rangle \overset{a}{\rightsquigarrow} \langle P,id,\mathscr{E}[e'].Q',S',r,I\rangle}{\mathscr{K}\ ||\ \langle P,id,\mathscr{E}[e].Q,S,r,I\rangle \overset{a}{\hookrightarrow} \mathscr{K}\ ||\ \langle P,id,\mathscr{E}[e'].Q',S',r,I\rangle}$$

(PROC INVOC)

$$\frac{\begin{array}{c} id' = I(i) \qquad \Sigma' = \langle P,id',e'.Q',S',[C'.F'],I'\rangle \in \mathscr{K} \qquad \texttt{capsule}\ C'(\overline{D\ j})\{..\ T\ p(\overline{T\ var})\{e\}\ ..\} \in P \qquad e'' = e\ [\bar{v}/\overline{var},I'(\bar{j})/\bar{j}] \qquad R = reach(\bar{v},S) \\ \forall l' \in dom(F)\ .\ R' = \bigcup reach(l',S) \qquad R\cap R' = \emptyset \qquad fresh(l) \qquad \mathscr{K}' = \mathscr{K} \uplus \langle P,id',e'.Q'.\texttt{resolve}(l,e'',id,p),S' \oplus R,[C'.F'],I'\rangle \end{array}}{\mathscr{K}\ ||\ \langle P,id,\mathscr{E}[i.p(\bar{v})].Q,S,[C.F],I\rangle \overset{invoke(id,id',p,l)}{\hookrightarrow} \mathscr{K}'\ ||\ \langle P,id,\mathscr{E}[l].Q,S[l:=\varepsilon]\ominus R,[C.F],I\rangle}$$

(RESOLVE)

$$\frac{R = reach(v,S) \qquad \forall l' \in dom(F)\ .\ R' = \bigcup reach(l',S) \qquad R\cap R' = \emptyset \qquad \Sigma = \langle P,id,e.Q,S[l=\varepsilon],r,I\rangle \in \mathscr{K} \qquad \mathscr{K}' = \mathscr{K} \uplus \langle P,id,e.Q,S[l:=v]\oplus R,r,I\rangle}{\mathscr{K}\ ||\ \langle P,id',\texttt{resolve}(l,v,id,p).Q',S',[C'.F'],I'\rangle \overset{resolve(id',id,p,l)}{\hookrightarrow} \mathscr{K}'\ ||\ \langle P,id',v.Q',S'\ominus R,[C'.F'],I'\rangle}$$

Figure 8. Local and global operational semantics of *Panini*.

proper substitutions for its formal parameters $\overline{var}$ and imported capsule instances $\bar{j}$. The notation $e[\bar{v}/\overline{var}]$ substitutes in e, formal parameters $\overline{var}$ with their values $\bar{v}$. Invocation of a local procedure of a capsule instance *id* causes a local action $local(id)$.

In (FIFO DEQUEUE), after evaluation of the head of the queue in a capsule instance to a value v, the next expression in the queue is moved to the head of the queue for evaluation, if the queue is not empty. Dequeue blocks until the queue of the capsule instance is not empty. Dequeuing a queue of a capsule instance causes a local action as well. Semantics of a let expression is standard.

3.2.2 Concurrent Asynchronous Global Semantics

Global evaluation relation $\overset{a}{\hookrightarrow}$ denotes concurrent local evaluations of capsule instances of a *Panini* program as well as their asynchronous interactions through procedure invocations.

The rule (CONGRUENCE) plays the role of a preemptive scheduler that nondeterministically chooses a capsule instance *id* in the global configuration $\mathscr{K}$ to take an atomic action at each point in time, according to the local semantic rules.

In (PROC INVOC), a capsule instance *id* asynchronously invokes the procedure p of a capsule instance with the name i. The invoking capsule finds the identifier id' for the invoked capsule name i in its instance mapping I, finds its corresponding configuration Σ' in the global configuration $\mathscr{K}$ and retrieves the body e of its invoked procedure p. It then replaces in e, the formal parameters $\overline{var}$ of p with their values $\bar{v}$ and imports $\bar{j}$ of its capsule type C' with their identifiers from instance mapping I', to arrive at e''. Then it wraps e'' in a `resolve` expression with a fresh future location l for returning the result of the invocation, to its invoking capsule *id*, and appends the resolve expression to the *tail* of the queue Q' of the invoked capsule instance id'. The notation $\mathscr{K} \uplus \langle P,id',e'.Q'.\texttt{resolve}(l,e'',id,p),S'\oplus R,[C'.F'],I'\rangle$ denotes overriding the configuration $\langle P,id',e'.Q',S',[C'.F'],I'\rangle$ of the capsule instance id' in the global configuration $\mathscr{K}$, where $\uplus$ is an overriding operation. A resolve expression $\texttt{resolve}(l,e,id,p)$ is a sugar for $\texttt{let}\ x=e\ \texttt{in}\ \texttt{resolve}(l,x,id,p)$ where x is free in e.

Because of the asynchrony of the procedure invocation, in (PROC INVOC), the control immediately returns back to the invoking capsule *id* without waiting for the execution of the invoked procedure p. The future location l is now *shared* between the invoking and invoked capsule instances, marked in the invoking capsule instance *id* as an unresolved future location with its value ε. The invocation expression performs an $invoke(id,id',p,l)$ action.

The resolve expression ensures that the result of its evaluation is sent back to the invoking capsule instance when it is ready and is going to be accessible through the future location. In (RESOLVE) the

invoked capsule instance id' returns the result v of the evaluation of its expression e to the invoking capsule id and assigns the value to the future location l in its store, i.e. $S[l := v]$. The resolve expression performs a $resolve(id', id, p, l)$ action.

The rule (PROC INVOC) along with (FIFO DEQUEUE) enforce the first in first out (FIFO) evaluation order of expressions in the queue of a capsule where (PROC INVOC) appends to the tail of the queue and (FIFO DEQUEUE) dequeues from its head. A *Panini* program terminates normally when for *each capsule* instance id in the program configuration, the expression at head of the queue is evaluated to a value and the queue is empty, i.e. $\langle P, id, v.\bullet, S, r, I\rangle$.

To illustrate, consider the capsule Client in Figure 4 and its asynchronous invocation of the procedure value of the capsule Counter, on line 4. Upon invocation of value on the capsule instance c, its body, on lines 6 of Figure 3, is wrapped in a resolve expression and is appended to the tail of c's queue. The control immediately returns to Client and the unresolved future NewVal is shared between the client and counter capsule instances, as a placeholder for the invocation's result. Any attempt to access NewVal in the client capsule, e.g. line 7, blocks until c dequeues the resolve expression for invocation of value and executes it to resolve the future.

Asynchronous invocation, blocking expressions and procedure execution order Asynchronous invocations of capsule procedures and blocking access to unresolved future locations imposes an order on executions of invoke procedures which may or may not be the same as in a synchronous settings. To illustrate, the procedure add, on line 5 of Figure 4, is invoked on the capsule instance c before value is invoked on the same instance, on line 6. This in turn means the body of add is appended to the queue of c and show up before the body of value the queue. Consequently, the invoked procedure add is executed before value because of the FIFO execution of queue in c. This is true, even if either add or value or both contain blocking expressions, e.g. trying to access an unresolved future, in their body because the execution of value in c does not start before the execution of add is finished.

```
1  capsule Client ( Counter c, Counter d ) {  ..
2    void test( Number y ) {
3      ..
4      d.add( y );
5      newVal = c.value();
6      ..
7    }
8  }
```

In contrast, consider invocations of procedures add and value on different capsule instances c and d, in above variation of Client, on lines 4–5. In this example, add and value procedures can execute in any arbitrary order because instances c and d run concurrently and can execute bodies of their invoked procedures add and value in any arbitrary order. This is true even with blocking expressions in the bodies of add or value procedures or both.

In other words, for asynchronous invocations of procedures of the same capsule instance, (FIFO DEQUEUE) ensures that these procedures run in the same order they are invoked, even if they contain blocking expressions. This in turn guarantees that blocking does not disrupt modular reasoning using Kleene closures, because procedure invocations in a closure are on the same receiver instance.

3.2.3 Ownership Transfer Semantics

To control sharing, *Panini*'s global semantics uses dynamic transfer of ownership for parameters and the return value of a procedure invocation. Transferring the ownership of a location from one capsule to another, removes that location and locations reachable from it, i.e. its reach (representation), from the local store of the former instance and adds them to the local store of the latter. This in turn guarantees that an invocation of a procedure does not cause sharing of its parameters and the result between the invoking and invoked capsule instances. *Panini*'s ownership transfer resembles changing threads access sets in a multithreaded program [24] or inferred ownership transfer semantics in SOTER [32] for programs in the actor language ActorFoundry [2].

In (PROC INVOC), upon invocation of a capsule's procedure, the ownership of the actual parameters $\bar{v}$ of the procedure and their reach $reach(v,S)$, in Figure 9, is transferred from the invoking capsule instance to the invoked capsule. The auxiliary function $\ominus$, in Figure 9, removes locations from a local store of a capsule instance whereas $\oplus$ adds locations to the local store of the capsule instance. For example, in the configuration $\langle P, id, \mathscr{E}[l].Q, S[l := \varepsilon] \ominus R, [C.F], I\rangle$, locations R are removed from the local store S of the invoking capsule instance id, and in $\langle P, id', e'.Q'.\texttt{resolve}(l, e'', id, p), S' \oplus R, [C'.F'], I'\rangle$ the locations R are added to the local store S' of the invoked capsule instance id'.

After transferring a location, $\ominus$ maps the value of a transferred location to $\square$. This in turn means the capsule instance does not own the transferred location anymore and any attempt to access it results in an exceptional state.

A state of a capsule instance or its reach cannot be transferred to another capsule instance. The condition $R \cap R' = \emptyset$, checks that there is no shared location among transferred locations and their reach R and the locations in the capsule's state or its reach R'.

In (RESOLVE), upon returning the result of an invocation, the ownership of the resolved future value, holding the result, and its reach in the local store of the invoked capsule instance is transferred to the invoking capsule instance. Similar to (PROC INVOC), the state of the invoked capsule instance cannot be transferred when returning the result of an invocation.

To illustrate, the ownership of the parameter y, on line 5 of the procedure test in Figure 4, is transferred from the invoking instance of capsule Client to the invoked capsule instance c. The ownership of the future newVal, on line 6, is transferred from c to the invoking capsule instance when the future is resolved and is ready.

$$\text{(FRESH)}\quad \frac{\forall \langle P, id_k, \mathscr{E}[e_k].Q_k, S_k, r_k, I_k\rangle \in \mathscr{K} \;.\; l \notin dom(S_k)}{fresh(l)}$$

$$\text{(REACH)}\quad \frac{v \in \{(), \square\}}{reach(v,S) = \bullet} \qquad \text{(REACH LOCATION)}\quad \frac{v \notin \{(), \square\} \qquad S[v = v']}{reach(v,S) = \{(v,v')\} \cup reach(v', S)}$$

$$\text{($\ominus$ LOCATION)}\quad S \ominus \{(l,v)\} = S[l := \square] \qquad \text{($\ominus$ REACH)}\quad \frac{\forall \{(l',v')\} \in R}{S \ominus R = (S \ominus (l',v')) \ominus (R \setminus \{(l',v')\})}$$

$$\text{($\oplus$ LOCATION)}\quad S \oplus (l,v) = S[l := v] \qquad \text{($\oplus$ REACH)}\quad \frac{(l',v') \in R}{S \oplus R = (S \oplus (l',v')) \oplus (R \setminus \{(l',v')\})}$$

$$\begin{aligned}
&labels(v) = \emptyset\\
&labels(\texttt{self}.f) = \emptyset\\
&labels(i.p(\bar{e})^{\alpha}) = \{^{\alpha}\} \cup labels(\bar{e})\\
&labels(\texttt{self}.p(\bar{e})) = labels(\bar{e})\\
&labels(\texttt{self}.f := e) = labels(e)\\
&labels(\texttt{ref}\ e) = labels(e)\\
&labels(!e) = labels(e)\\
&labels(e_1 := e_2) = labels(e_1) \cup labels(e_2)\\
&labels(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = labels(e_1) \cup labels(e_2)\\
&labels(T\,p\,(\overline{T\ var})\{e^{\alpha}\}) = \{^{\alpha}\} \cup labels(e)
\end{aligned}$$

Figure 9. *Panini*'s auxiliary functions.

3.3 Exceptional Semantics

A *Panini* program terminates abnormally, throwing an exception **OWE**, if a capsule instance attempts to access a location whose ownership is transferred or pass or return capsule states or their representations into/from global procedure invocations. Figure 10 shows *Panini*'s exceptional semantics.

(X DEREF)

$$\langle P, id, \mathscr{E}[!\ l].Q, S[l = \square], r, I\rangle \overset{read(id,l)}{\rightsquigarrow} \langle P, id, \texttt{OWE}.Q, S, r, I\rangle$$

(X REF ASSIGN)

$$\langle P, id, \mathscr{E}[l := v].Q, S[l = \square], r, I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P, id, \texttt{OWE}.Q, S, r, I\rangle$$

(X PROC INVOC)

$$\frac{R = reach(\overline{v}, S) \qquad \forall l' \in dom(F)\ .\ R' = \bigcup reach(l', S) \qquad R \cap R' \neq \emptyset}{\mathscr{K}\ ||\ \langle P, id, \mathscr{E}[i.p(\overline{v})].Q, S, [C.F], I\rangle \overset{invoke(id,id',p,l)}{\hookrightarrow} \langle P, id, \texttt{OWE}.Q, S, [C.F], I\rangle}$$

(X RESOLVE)

$$\frac{R = reach(v, S) \qquad \forall l' \in dom(F)\ .\ R' = \bigcup reach(l', S) \qquad R \cap R' \neq \emptyset}{\mathscr{K}\ ||\ \langle P, id, \texttt{resolve}(l, v, id', p).Q, S, r, I\rangle \overset{resolve(id,id',p,l)}{\hookrightarrow} \langle P, id, \texttt{OWE}.Q, S, r, I\rangle}$$

Figure 10. Exceptional semantics of *Panini*, select rules.

In rules (X DEREF) and (X REF ASSIGN), attempting to dereference or assign to a location that is transferred out and is not owned by a capsule instance anymore results in throwing an ownership transfer exception **OWE**. A transferred location is marked with the value $\square$, by the transfer operation $\ominus$. In rules (X PROC INVOC) and (X RESOLVE), attempting to pass capsule states or any location in their reach as actual parameters or results of global procedure invocations, i.e. $R \cap R' \neq \emptyset$, causes throwing the ownership exception and termination of the program. A *Panini* program terminates abnormally when a capsule instance *id* in the program evaluates the head of its queue to **OWE**, i.e. $\langle P, id, \texttt{OWE}.Q, S, r, I\rangle$.

In *Panini*'s core semantics, for simplicity and without loss of generality, exceptions are final states and the program terminates after throwing an exception. However, in the current prototype implementation of *Panini*'s compiler, a violation of ownership transfer semantics is detected using a modular static analysis incorporated into its type system and is reported as a compile time warning rather than terminating the program at runtime.

3.4 Initial Configuration

Evaluation of a *Panini* program, follows a phase which builds the program's *initial* global and local configurations. Figure 11 shows *Panini*'s initial configuration rules. The initial configuration phase takes a *Panini* program and *recursively* processes design declarations of its capsules, such that for each capsule instance declaration in a design declaration, it instantiates a capsule instance and for each wiring declaration it connects the declared capsule instances together.

The initial configuration phase starts with the rule (MAIN). The rule constructs a special capsule instance main of capsule Main with the identifier 0, i.e. $Construct(Main\ main, 0)$. The capsule Main is the entry point to a *Panini* program. The rule (MAIN), set the capsule instance main to the global configuration $\mathscr{K}_0$ and calls functions *instantiateRec* and *wireupRec*, in rules (INSTANTIATE REC) and (WIREUP REC) to recursively instantiate and connect other capsule instances of the program.

For a capsule instance declaration *C i*, declared in the enclosing capsule instance *id* and a global configuration $\mathscr{K}$, the function

(MAIN)

$$\frac{\mathscr{K}_0 = instantiate(Main\ main, 0) = \langle P, 0, \bullet.Q, S, [Main.F], I\rangle \quad \texttt{capsule}\ Main()\{\overline{T\ f}\ \ \texttt{design}\{\overline{ins}\ \ \overline{wire}\}\ \ \overline{proc}\} \in P \quad \forall C\ i \in \overline{ins}\ .\ \mathscr{K} = instantiateRec(\mathscr{K}_0, 0, C\ i) \quad \forall i(\overline{j}) \in \overline{wire}\ .\ \mathscr{K}' = wireupRec(\mathscr{K}, 0, i(\overline{j}))}{construct(Main\ main, 0) = \mathscr{K}'}$$

(INSTANTIATION REC)

$$\frac{\Sigma' = \langle P, id', \bullet.Q', S', [C'.F'], I'\rangle = instantiate(id, C\ i) \quad \Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \quad \mathscr{K}' = \mathscr{K} \uplus \langle P, id, \bullet.Q, S, [C.F], I[i := id']\rangle \cup \Sigma' \quad \forall C'\ i' \in \overline{ins}\ .\ \mathscr{K}'' = instantiateRec(\mathscr{K}', id', C'\ i')}{instantiateRec(\mathscr{K}, id, C\ i) = \mathscr{K}''}$$

(WIRING REC)

$$\frac{\Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \in \mathscr{K} \qquad id_i = I(i) \quad \Sigma_i = \langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], I_i\rangle \in \mathscr{K} \qquad \Sigma_i' = wireup(id, i(\overline{j})) \quad \texttt{capsule}\ C_i(\overline{imp})\{..\ \texttt{design}\{\overline{ins}\ \ \overline{wire}\}\ ..\} \in P \qquad \mathscr{K}' = \mathscr{K} \uplus \Sigma_i' \quad \forall i'(\overline{j'}) \in \overline{wire}\ .\ \mathscr{K}'' = wireupRec(\mathscr{K}', id_i, i'(\overline{j'}))}{wireupRec(\mathscr{K}, id, i(\overline{j})) = \mathscr{K}''}$$

(INSTANTIATE)

$$\frac{fresh(id') \qquad \texttt{capsule}\ C(..)\{\overline{T\ f}\ \ \texttt{design}\{\overline{ins}\ \ \overline{wire}\}\ \ \overline{proc}\} \in P \quad F = \emptyset \qquad S = \emptyset \quad I = \emptyset \qquad Q = \bullet \qquad \forall (T\ f) \in \overline{T\ f}\ .\ fresh(l), F[f := l], S[l := ()] \quad \Sigma = \langle P, id', \bullet.Q, S, [C.F], I\rangle}{instantiate(id, C\ i) = \Sigma}$$

(WIREUP)

$$\frac{\Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \in \mathscr{K} \qquad id_i = I(i) \qquad \Sigma_i = \langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], I_i\rangle \in \mathscr{K} \quad \forall j \in \overline{j}\ .\ id_j = I(j), \langle P, id_j, \bullet.Q_j, S_j, [C_j.F_j], I_j\rangle \in \mathscr{K} \quad \texttt{capsule}\ C_i(\overline{imp})\{..\ \texttt{design}\{\overline{ins}\ \ \overline{wire}\}\ ..\} \in P \quad \Sigma_i' = \langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], \forall (D\ h) \in \overline{imp}, j \in \overline{j}\ .\ I_i[h := id_j]\rangle}{wireup(id, i(\overline{j})) = \Sigma_i'}$$

Figure 11. Rules to create initial configuration of *Panini* programs.

instantiateRec defined in (INSTANTIATE REC), instantiates a capsule configuration Σ' with the identifier id' and name *i*, using *instantiate*; changes the instance mapping *I* of its enclosing capsule instance to map the name *i* to its identifier id', i.e. $I[i := id']$ and adds Σ' to $\mathscr{K}$ to create a new global configuration $\mathscr{K}'$. To process the capsule instance declarations $C'\ i'$ in newly created id', *instantiateRec* is recursively called with the new global configuration $\mathscr{K}'$.

For a wiring declaration $i(\overline{j})$ declared in the enclosing capsule instance *id* and the global configuration $\mathscr{K}$, the function *wireupRec* defined in (WIREUP REC), connects instances *i* and $\overline{j}$, using *wireup*, to construct the new configuration Σ_i' for *i*; replaces the old configuration Σ_i with its new configuration Σ_i' in $\mathscr{K}$ to arrive at a new global configuration $\mathscr{K}'$. To process the wiring declarations $i'(\overline{j'})$ for the newly wired capsule instance id_i, *wireupRec* is recursively called with the identifier id_i and the new global configuration $\mathscr{K}'$.

Function *instantiate* defined in (INSTANTIATE) simply instantiates the capsule configuration Σ for the capsule instance declaration *C i* in its enclosing capsule *id*. Function *wireup* in (WIREUP) connects capsule instances *i* and $\overline{j}$ in their enclosing capsule *id*.

To illustrate, consider the capsule Main in Figure 6. In this example, $Construct(Main\ main, 0)$ creates a capsule instance configuration for Main with the identifier 0; then it calls *instantiateRec* for the instantiation of capsule instances counter and client, declared on lines 3–4, followed by a call to *wireupRec* to connect the counter and client instances, as declared on line 5.

There is no sharing among capsule instances of the initial configuration of a *Panini* program, as shown in Lemma 1.

LEMMA 1. ***(No sharing of memory locations in initial configuration)*** *Let $\Sigma = \langle P, id, \mathscr{E}[e].Q, S, r, I\rangle$ and $\Sigma' = \langle P, id', \mathscr{E}[e'].Q', S', r', I'\rangle$ be two arbitrary capsule instance configurations in the initial configuration $\mathscr{K}$ for a Panini program P constructed using the rules in Figure 11, i.e. $\Sigma, \Sigma' \in \mathscr{K}$. Let $A = dom(S) \cap dom(S')$ be the intersection of domains of the stores S and S'. Then $A = \emptyset$.*

In other words, there is no sharing between local stores of capsule instances in the initial configuration of a program.

Proof Sketch: The proof is based on the cases of the initial configuration rules of Figure 11. The rule (INSTANTIATE) is the only rule allocating memory locations and mapping them in the stores of capsule instances. The rule only uses fresh locations for instantiation of each capsule instance, and thus does not cause any sharing among capsule instances and thus their local stores.

3.5 Actions: Conflict and Happens-Before Relations

Evaluation of a *Panini* program, with its nondeterministic preemptive scheduler, results in a trace of interleaved actions, shown in Figure 7, performed by different capsule instances of the program. However, as illustrated in figures 2, 3 and Figure 4, for a trace of a capsule's procedure, interleaving actions of other capsule instances, can be moved in the trace, such that they only appear right after the global procedure invocations in the trace, i.e. sparse interference. This is because of *Panini*'s conflicting and happens-before relations and mover properties of its actions which in turn are affected by sharing semantics of *Panini*.

In the following, we define the execution trace of a *Panini* program, the conflict and happens-before relations, and prove mover properties of its comprising actions, using Lipton's reduction theory [30]. Definitions 1-4 are adapted from previous work [47].

DEFINITION 1. ***(Trace)*** *An execution trace of a Panini program P is a total order of actions a, as defined in Figure 7, performed by individual capsule instances in the program configuration $\mathscr{K}$ when evaluating the program thorough local and global evaluation rules of Figure 8.*

DEFINITION 2. ***(Adjacent & neighbor actions)*** *Two actions a and b in a trace $\mathscr{T}$ are adjacent if one follows immediately after another. Two adjacent actions a and b are neighbors if they are performed by different capsule instances, i.e. $instance(a) \neq instance(b)$. The auxiliary function instance returns the capsule identifier of an action.*

DEFINITION 3. ***(Commuting & conflicting actions)*** *Let a_1 and a_2 be neighbor actions of capsule instances id_1 and id_2 in an execution trace $\mathscr{K}_0 \overset{a_1}{\hookrightarrow} \mathscr{K}_1 \overset{a_2}{\hookrightarrow} \mathscr{K}_2$. Then actions a_1 and a_2 commute, written as a_1 !# a_2, if swapping them in the trace, results in the same final state in the trace starting with the same start state, i.e. $\mathscr{K}_0 \overset{a_2}{\hookrightarrow} \mathscr{K}_1' \overset{a_1}{\hookrightarrow} \mathscr{K}_2$. Otherwise, a_1 and a_2 conflict, written as $a_1 \# a_2$.*

There are several conflicting actions in *Panini*, considering its semantics: read or write of an unresolved future location conflicts with the resolution of the same future location; and invoke action of a procedure of a capsule instance conflicts with another invoke action on the same capsule instance.

A happens-before relation $\prec$ [27] orders the conflicting actions. For example, in *Panini* resolve of a future location l by a capsule instance id must happen-before any reads (or writes) of the location by another capsule instance id', i.e. $resolve(id, id', l, p) \prec read(id', l)$. Figure 12 shows *Panini*'s conflicting actions and their happens-before relations. The happens-before relation is a transitively closed partial order [49].

$$\frac{\langle P, id', \mathscr{E}[e].Q, S[l = \varepsilon], r, I\rangle \in \mathscr{K}}{read(id', l) \ \#\ resolve(id, id', l, p)} \qquad \frac{\langle P, id', \mathscr{E}[e].Q, S[l = \varepsilon], r, I\rangle \in \mathscr{K}}{write(id', l) \ \#\ resolve(id, id', l, p)}$$

$$invoke(id_1, id, p, l) \ \#\ invoke(id_2, id, p', l')$$

$$resolve(id, id', l, p) \prec read(id', l) \qquad resolve(id, id', l, p) \prec write(id', l)$$

Figure 12. Conflicting actions in *Panini* where # denotes conflict and their happens-before $\prec$ relation.

DEFINITION 4. ***(Right, left, both & non-mover actions)*** *Let a_1 and a_2 be neighbor actions of capsule instances id_1 and id_2 in an arbitrary execution trace $\mathscr{K}_0 \overset{a_1}{\hookrightarrow} \mathscr{K}_1 \overset{a_2}{\hookrightarrow} \mathscr{K}_2$.*

Then a_1 is a right mover if swapping a_1 with a_2 in the trace results in the same final state in the trace, beginning with the same start state, i.e. $\mathscr{K}_0 \overset{a_2}{\hookrightarrow} \mathscr{K}_1' \overset{a_1}{\hookrightarrow} \mathscr{K}_2$. Conversely, a_2 is a left mover if swapping it with a_1 results in the same final state. An action that can be swapped with its both left and right neighbor actions in any trace, is a both mover. Conversely, an action that cannot be swapped with neither its left nor right neighbors is a non-mover.

Panini's semantics determines mover properties of its actions. Lemma 2 specifies mover properties of *Panini*'s actions.

LEMMA 2. ***(Panini action's mover properties)*** *Let $\mathscr{T}$ be the execution trace of an arbitrary Panini program P.*

Then, in trace $\mathscr{T}$ read and write actions $read(id, l)$ and $write(id, l)$ of a capsule instance id of a memory location l are right movers, as defined in Definition 4; a global invocation action $invoke(id, id', p, l)$ of a procedure p from the invoking capsule id to the invoked capsule id' and result l is a non-mover; and a resolve action $resolve(id, id', p, l)$ for this global invocation is a left mover.

Proof Sketch: The proof is based on happens-before relations of *Panini*'s actions in Figure 12: resolve of a future location must happen before any read or write of the location and thus in a trace of a program, a read action $read(id, l)$ of future location l cannot be swapped with a left neighbor action $resolve(id', id, l, \_)$ resolving the same location. Thus, a read action is a right mover. The same applies to an action writing a future location. Similarly, a resolve action is a left mover. Swapping an invoke action $invoke(id', id, \_, \_)$ invoking a procedure of capsule id with another left or right neighbor invoke action $invoke(id'', id, \_, \_)$ on the same capsule id results in different program states especially different queues for the capsule instance id. Thus an invoke action is a non-mover. The notation _ denotes irrelevant values in actions.

3.6 Sharing of Capsule Instances and Futures

Panini limits sharing among two capsules to their imported capsule instances and unresolved futures of their procedure invocations.

```
1  capsule Main() {
2    design {
3      Counter counter;
4      Client  client1 ,  client2 ;
5      client1 (counter);
6      client2 (counter);
7    } ..
8  }
```

Figure 13. Sharing counter among client1 and client2.

Panini's semantics allows an an imported capsule instance to be freely shared among other instances as specified in a design declaration of their enclosing capsule. For example, in the design

declaration of Figure 13, the capsule instance counter is shared among two client instances client1 and client2, on line 4.

Panini's semantics limits sharing of memory locations to unresolved future locations, as shown by Lemma 3. A future location which is a placeholder for the result of an asynchronous procedure invocation, is shared among its invoking and invoked capsule instances as long as it is unresolved and accesses to it are synchronized. That is, any attempt to access an unresolved future location in the invoking capsule instance blocks until the future is ready.

LEMMA 3. ***(Sharing of unresolved future locations)*** *Let* $\Sigma = \langle P, id, \mathscr{E}[e].Q, S, r, I\rangle$ *and* $\Sigma' = \langle P, id', \mathscr{E}[e'].Q', S', r', I'\rangle$ *be two arbitrary capsule instance configurations in a global configuration* $\mathscr{K}$ *for a Panini program P, i.e.* $\Sigma, \Sigma' \in \mathscr{K}$. *Let* $\mathscr{T}$ *be the execution trace of the program P. Let* $A = dom_{\square}(S) \cap dom_{\square}(S')$ *be the intersection of domains of the stores S and* S' *minus their transfered locations with the value* $\square$. *Let action a and* a' *be any of the read and write actions of a location l in A in the trace* $\mathscr{T}$ *by capsule instances id and* id', *respectively.*

Then either $A = \emptyset$ *or:*

(i). $\forall l \in A\,.\ S[l = \varepsilon] \veebar S'[l = \varepsilon]$; *and*

(ii). $\forall l \in A, a \in \{read(id, l), write(id, l)\}, a' \in \{read(id', l), write(id', l)\}\,.$ $a \prec a' \veebar a' \prec a$.

In other words, (i) the only memory locations that local stores S and S' *may share are unresolved future locations with (ii) synchronized accesses (reads and writes), i.e. with happens-before relation between their reads and writes.*

Transferred locations with values $\square$ are irrelevant and thus taken out in $dom_{\square}$ of local stores, because any attempt to access them terminates the program. The notation $\veebar$ denotes an exclusive logical disjunction, in which at most one of the disjuncts can be true.

Proof Sketch: The proof is by cases on *Panini*'s normal and exceptional semantics rules in figures 8 and 10, happens-before relations in Figure 12 and Lemma 1: initial configuration does not cause any sharing of memory location among capsule instances; dynamic transfer of ownership of locations among capsule instances in *Panini*'s dynamic semantics and especially (PROC INVOC) and (RESOLVE), limits sharing of locations among capsules to only unresolved future locations; and rules (DEREF) and (REF ASSIGN) synchronize access to these shared future locations by enforcing that resolving of a future location in one capsule instance must happen-before any of its reads or writes in other capsule instances.

4. Sparse and Cognizant Interference

Panini guarantees sparse interference by limiting sharing among two capsules to other capsule instances and unresolved futures, and guarantees cognizant interference by limiting accessibility of states of a capsule instance to only through its procedures and dispatching a procedure invocation on the static type of its receiver capsule. In this section we formalize and sketch proofs of *Panini*'s sparse and cognizant interferences. Full proofs can be found in §B.

4.1 Sparse Interference

Theorem 5 formalizes *Panini*'s sparse interference property which limits the interference points of a program to points after its global procedure invocations.

THEOREM 5. ***(Sparse interference in Panini)*** *Let P be a program in Panini. Let* $\mathscr{I}$ *be a set of labels after global capsule invocations and after procedure bodies in P, i.e.* $\mathscr{I} = \{\alpha | \alpha \in labels(P),\ i.p(\overline{e})^{\alpha} \in P \vee T\ \ p(\overline{form})\{e\ ^{\alpha}\} \in P\}$ *where the auxiliary function labels is defined in Figure 9. Then* $\mathscr{I}$ *is the set of all potential interference points for P.*

Proof Sketch: The proof is based on Lemma 2 where in a trace of a capsule's procedure interfering actions of other capsule instances, can be safely moved to either after the global procedure invocation actions in the trace or at the beginning or end of the execution of the trace. The interference at the beginning of the trace of a procedure, can be safely moved out to the trace of its invoking procedure, either to the end of the invoking procedure's trace or after one of its global procedure invocations.

4.2 Cognizant Interference

Theorem 6 formalizes *Panini*'s cognizant interference property which limits the interfering behavior at each global invocation interference point to Kleene closure of behaviors of procedures in the static type of the invocation's receiver.

THEOREM 6. ***(Cognizant interference in Panini)*** *Let* $\Sigma = \langle P, id, \mathscr{E}[i.p_k(\overline{e})^{\alpha}; e].Q, S, r, I\rangle$ *be the configuration for capsule instance id in the global configuration* $\mathscr{K}$, *such that the capsule is about to evaluate the expression* $i.p_k(\overline{e})^{\alpha}; e$ *at the head of its queue with a single interference point* $^{\alpha}$, *i.e, there is no other interference in other expressions* $\overline{e}$ *and e in the sequence. Let* $\Sigma' = \langle P, id', \mathscr{E}[e'].Q'.\texttt{resolve}(l, e'_k, id, p_k), S', [C'.F'], I'\rangle \in \mathscr{K}$ *be the capsule configuration for capsule name i right at the interference point* α, *i.e. right after execution of invocation* $i.p_k(\overline{e})$ *in* Σ. *Let* C' *be the static capsule type for i with declared procedures* p_1 .. p_n, *i.e.* `capsule` $C'(..)$ {`design` *state* $T_1\ p_1(..)\{e_1\}$ .. $T_k\ p_k(..)\{e_k\}$ .. $T_n\ p_n(..)\{e_n\}\} \in P$. *Also let* $e'_1, ..e'_n$ *be the bodies of procedure* $p_1, .., p_n$ *with their formal parameters substituted with their values from their invocation sites and their local capsule names substituted with their identifiers from instance mappings* I'. *Also let* θ *be the interfering behavior of other capsule instances in the program at the interference point* $^{\alpha}$.

Then, θ *is the Kleene closure of behaviors of procedures of the capsule id', i.e.* $\theta = \{e'_1, .., e'_n\}^*$.

Proof Sketch: The proof is by cases on local and global evaluation rules in Figure 8 and that *Panini* limits sharing of memory locations to unresolved future locations with synchronized access, in Lemma 3, and limits accessibility of the state of a capsule instance to only through global invocation of its procedures, in Lemma 4.

LEMMA 4. ***(Global accessibility through procedures)*** *Let* $\Sigma = \langle P, id, \mathscr{E}[e].Q, S, [C.F], I\rangle$ *and* $\Sigma' = \langle P, id', \mathscr{E}[e'].Q', S', [C'.F'], I'\rangle$ *be arbitrary capsule instance configurations in a global configuration* $\mathscr{K}$, *i.e.* $\Sigma, \Sigma' \in \mathscr{K}$. *Let C be the static capsule type of id with state* $\overline{f}$ *and procedures* p_1 .. p_n, *i.e.* `capsule` $C(..)$ {`design` $\overline{T\ f}$ $U_1\ p_1(..)\{e_1\}$ .. $U_n\ p_n(..)\{e_n\}\} \in P$.

Then during evaluation of program P, the capsule instance id' can access (read and write) states $\overline{f}$ *of the instance id only through its procedures* p_1 .. p_n, *and not directly through memory locations.*

Proof Sketch: The proof follows from Lemma 3 that guarantees the only shared locations among capsules are unresolved future locations; and the rules (PROC INVOC) and (RESOLVE) in Figure 8 that prevent transfer of ownership of states of a capsule or its reach during procedure invocations and resolving of their results.

5. Hoare-Style Modular Reasoning

Standard Hoare logic [21] does not take interference into account and cannot be used right out of the box to reason about concurrent programs [19, 40]. *Panini*'s sparse and cognizant interference enables use of Hoare logic in the presence of interference by enabling interference points of a *Panini* program and their interfering behaviors to be statically known.

In a Hoare logic for *Panini*, the only rule in the logic that needs to take into account the interference is the rule for global

procedure invocations. This is because *Panini*'s sparse interference limits interference points to after global procedure invocations. Other rules can be used as if they are interference-free. To take into account interference, it is sufficient to consider the interfering behavior at each interference point [19].

To illustrate consider the Hoare triple $\{Pre\}\ i.p(\bar{v})\ \{Post\}$ which says that if the execution of the procedure invocation $i.p(\bar{v})$ starts in a state satisfying the predicate *Pre* and procedure p of the capsule instance i executes and terminates, it terminates in a state satisfying the predicate *Post*. To take into account the interference, the triple becomes $\{Pre\}\ i.p(\bar{v})^{\alpha}\ \{Post\}$ in which $^{\alpha}$ is an interference point. The new triple says if the execution of the procedure p of the capsule instance i starts in a state satisfying *Pre* and is interfered with by execution of some interfering tasks at $^{\alpha}$, if execution of p terminates, it terminates in a state satisfying the predicate *Post*.

Pure predicates In a Hoare logic, predicates *Pre* and *Post* must be side effect free and if they invoke a procedures, the procedures must be pure. A procedure is pure if it does not change the state of its program. In *Panini* the state of a program not only includes local stores of its capsule instances but also their queues. In other words, in *Panini* a procedure is pure if it does not change stores or queues of capsule instances in the program, including itself. To meet such a requirement, a pure procedure in a capsule instance can only read states or invoke other pure procedures of its enclosing capsule instance via **self**. Predicates *Pre* and *Post* can only invoke pure procedures of their enclosing capsule instance too and cannot even invoke pure procedures of other capsule instances. This is because, invocation of a pure procedure of another capsule instance, adds the invoked procedure to the queue of invoked capsule instance and changes the state of the system, i.e. the predicate is not pure.

Interference-free predicates In a Hoare triple $\{Pre\}\ e\ \{Post\}$ in a capsule instance in *Panini*, predicates *Pre* and *Post* are free from interference. This is because, these pure predicates only read states of their enclosing capsule instance and invoke only its pure procedures. Corresponding actions for reading states and invocation of self procedures are both movers and thus any interference in the predicates can be moved out as it appears to happen before evaluation of the *Pre* or after the evaluation of the *Post* predicate.

```
1   capsule Client ( Counter c ) {
2     Number newVal, oldVal;
3     //@ requires y.value() >= 0
4     //@ ensures newVal >= oldVal
5     void test( Number y ) {
6       oldVal = c.value();
7       c.add( y );
8       newVal = c.value();
9     }
10  }
11  capsule Counter {
12    Number x;
13    //@ requires y.value() >= 0
14    //@ ensures self.value() >= \old( self.value());
15    void add( Number y ) { .. }
16    //@ pure
17    Number value() { .. }
18  }
```

Figure 14. Static verification of the behavioral contract of test.

Behavioral contracts In *Panini*, a behavioral contract for a procedure, specifies the precondition and postcondition of a procedure. Figure 14 illustrates the contracts for procedure test of capsule Client, with its pre and postconditions on lines 3–4. The contract says, if the execution of the procedure test starts in a state satisfying the precondition $y.value() >= 0$ and its execution terminates, it terminates in a state satisfying the postcondition $newVal >= oldVal$. Similarly, the contract for procedure add of capsule Counter, on lines 13–14, says it only increases the value of the counter requiring that parameter y is a positive number. Finally, the contract for value, on line 16 says it is a pure procedure and does not change the state of its enclosing capsule or any other capsule in the program. The precondition and postcondition of a behavioral contract are free from interference, similar to the interference-free precondition and postcondition of a Hoare triple.

Modular reasoning Hoare-style reasoning, can be used to statically verify the contract of procedure test. To illustrate, consider static verification of the method test in the capsule Client. The Hoare triple representing such verification looks like the following:

$$\begin{aligned}\Gamma \models \{&y.value() >= 0\}\\ &\mathbf{self}.oldVal = c.value();\\ &c.add(y)\\ &\mathbf{self}.newVal = c.value();\\ &\{\mathbf{self}.newVal >= \mathbf{self}.oldVal\}\end{aligned}$$

The notation $\Gamma \models \{Pre\}\ e\ \{Post\}$ denotes that the Hoare triple $\{Pre\}\ e\ \{Post\}$ is valid in the typing environment Γ.

Following *Panini*'s sparse interference, interference only happens after global procedure invocations. Thus, the Hoare triple becomes like the following to take into account the interference, with $^{\alpha}$ denoting the interference points:

$$\begin{aligned}\Gamma \models \{&y.value() >= 0\}\\ &\mathbf{self}.oldVal = c.value()\ ^{\alpha};\\ &c.add(y)\ ^{\alpha};\\ &\mathbf{self}.newVal = c.value()\ ^{\alpha};\\ &\{\mathbf{self}.newVal >= \mathbf{self}.oldVal\}\end{aligned}$$

Following *Panini*'s cognizant interference, the interfering behavior at the interference points in the above Hoare triple is $\theta = \{c.value(), c.add(\_)\}^*$, i.e. the Kleene closure of procedure of the static capsule type Counter for the receiver c of the global procedure invocations:

$$\begin{aligned}\Gamma \models \{&y.value() >= 0\}\\ &\mathbf{self}.oldVal = c.value(y);\ \{c.value(), c.add(\_)\}^*;\\ &c.add(y);\ \{c.value(), c.add(-)\}^*;\\ &\mathbf{self}.newVal = c.value();\ \{c.value(), c.add(\_)\}^*;\\ &\{\mathbf{self}.newVal >= \mathbf{self}.oldVal\}\end{aligned}$$

The above triple could be easily verified assuming $c.value()$ is pure and $c.add(\_)$ only increases the counter, according to their contracts. This is because the closure $\{c.value(), c.add(\_)\}^*$ either maintains the value of the counter c or increases it, which in turn means $\mathbf{self}.newVal >= \mathbf{self}.oldVal$. Such reasoning is modular because it only uses the implementation of Client and the interface (contract) of procedures of capsule Counter it refers to. The notation _ stands for the parameter of the procedure add. The exact value of this parameter is not known statically, however using the precondition of add, on line 13, we know it is a positive number.

Without sparse and cognizant interference, one must consider possibility of interferences with unknown behaviors between any two instructions of a program and its contracts [40].

Adding a procedure, say subtract, to the capsule Counter in Figure 14 which decreases the counter, changes the interference behaviors to the Kleene closure $\{value(), add(\_), subtract(\_)\}^*$. Using this closure, one cannot verify the postcondition of procedure add anymore. However, this is not a limitation and is not specific to *Panini*. A similar situation happens in other reasoning techniques including rely-guarantee [18, 25], Owicki-Gries work [35], etc.

Similar to sequential reasoning, completeness of our reasoning is proportional to completeness of procedure specifications. That is,

using incomplete specifications we can still reason about whatever specifications specify. For example, using the contract of add in Figure 14 we can reason about values of a counter, however, we cannot reason about other values which are part of its state but not mentioned in the contract.

6. Discussion

Expressiveness, usability, scalability, concurrency granularity *Panini* language [37], has been tried out on hundreds of thousands of lines of code of concurrent programs including translations of JavaGrande, NPB and StreamIt benchmarks and actor programs from Basset, Habanero and Jetlang [36] covering a variety of patterns such as master/worker, pipeline, event based coordination, loop parallelism. Our previous work shows that *Panini* programs perform as well as their corresponding multithreaded programs. During our experience, we did not run into any granularity issues with capsules. However, focus of this paper is on the formalization of *Panini*'s semantics, interference model and modular reasoning.

Closure analysis Analysis of a Kleene closure can grow with number of procedures in a capsule. However, *Panini*'s cognizant interference is still a significant improvement over oblivious interference in which interfering behavior is completely unknown. Closure analysis could be further improved for example by eliminating pure procedures from closures or use of procedure invocation protocols to eliminate invalid invocation sequences and similar directions which are part of our future work plans. Also, we conjecture that number of a capsule's procedures on average will be on par with average number of methods in a class, which is not a large number. For all Java projects in SourceForge as of Sep 2013, this average is about 8 methods, as obtained using Boa software repository analysis infrastructure [12].

7. Related Work

Actor and active objects Our work builds on the actor model [3, 4]. Some variants of the actor model, such as Erlang [6], guarantee confinement, i.e. no shared locations among actors, and use a single thread of execution per actor. Actors of this variant address the pervasive interference via their macro-step semantics [5] which limits interference points to message receive sites in the code. However, variants of the actor model which do not guarantee confinement, e.g. Scala Actors [20], or allow multiple unsynchronized execution threads per actor instance, e.g. Habanero [22], could still suffer from pervasive interference. Actor models and their variants also do not address the oblivious interference problem due to their dynamic binding of actor names and message names (in some cases), e.g. ActorFoundry [2], which in turn does not allow the static type of actor instances to be known statically.

Active objects [29], similar to actors, encapsulate their state and control. Variants of active objects, which guarantee confinement and synchronized access to memory in active objects, such as JCoBox [41], address the pervasive interference problem. Several techniques including ownership type systems [10] or immutable data [41] can be used to enforce confinement. Again, similar to actors, dynamic binding of names could still lead to oblivious interference in active objects.

Atomicity, transactional memory, cooperability and automatic mutual exclusion (AME) Transactional memory [28] is a concurrent programming model that enforces atomic blocks at runtime. There are also a variety of static [17] and dynamic [13] analyses to detect atomicity violations. Atomic sets [45] put fields of objects into atomic sets such that access to the fields in these sets is guaranteed to be atomic. These techniques are not concerned about oblivious interference and only partially address the pervasive interference by limiting the interference points to the outside of specified atomic blocks. However, interference outside atomic blocks can still be pervasive and happen between each two instructions. An atomic block is an interference free block of code.

Automatic mutual exclusion [1, 23, 42] inverts the model of atomic blocks in transactional memory such that code is run atomically unless explicitly specified using yield expressions. Similarly, cooperative reasoning [48, 49] and observationally cooperative multithreading [43] limit interleaving points to yield expressions. Yield expressions are put in places in the code such that semantics of cooperative scheduling with context switches only at yield expressions is equal to the semantics of an arbitrary preemptive scheduler with context switches between any two instructions. Task Types [26] enforce pervasive atomicity, i.e. every piece of code must be in some atomic block, through a data-centric technique for specification of shared objects and syntactically explicated accesses to share objects. These techniques address the pervasive interference, however, they are not concerned about obliviousness of interference points.

Rely-guarantee and Owicki-Gries's work In rely-guarantee reasoning [25] a module satisfies a guarantee condition after each instruction and in turn can assume the rely condition satisfied by the environment. Previous work [19] leverages rely-guarantee reasoning for thread-modular verification of multithreaded programs in which rely and guarantee conditions are specified for threads and their environments. Similarly, in Owicki and Gries's work [35] and its variations [33] each instruction is annotated by an interference-free assertion which must hold locally in the presence of concurrent interfering tasks. Modular rely-guarantee and global Owicki-Gries reasoning extend Hoare logic and address the oblivious interference problem through environment assumptions, however, they still assume pervasive interference between each two instructions.

Concurrent separation logic and abstract predicates In concurrent separation logic [34] accesses to a shared resource among processes are synchronized through conditional critical regions and thus there is no interference in each process. A resource invariant specifies what must hold before and after accessing the resource. Concurrent separation logic can be treated as a specialization of rely-guarantee for well-synchronized programs [14]. Influenced by separation logic, concurrent abstract predicates [11] and its impredicative variation (iCAP) [44] are self-stable predicates that enable fine-grained modular reasoning about concurrent programs by specifying changes to shared resources or using impredicative protocols, as if each predicate represents a disjoint resource.

Aspect-orientation Interference and its pervasiveness and obliviousness is discussed in the context of aspect-oriented programming languages [15, 39]. In these sequential languages interference between aspects and base code could lead to unexpected behaviors. However, aspect-oriented paradigms are mainly sequential and thus their interference problems are not due to concurrency. Consequently, solutions to these problems in these paradigms are not directly applicable to concurrent programming models.

Data race freedom There is a vast number of previous work on finding, fixing and preventing data races [7–9, 16, 38]. However, the absence of data races does not guarantee the absence of interferences and errors due to interferences [17]

8. Conclusion and Future Work

In this paper, we presented *Panini*, a core concurrent calculus with a sparse and cognizant interference model to address pervasive and oblivious interference problems. We formalized *Panini*, presented its semantics and illustrated how its interference model, using behavioral contracts, enables Hoare-style modular reasoning about its concurrent programs with interference. One avenue of future work, is to investigate reasoning about other properties of interest for concurrent programs, such as sequential consistency [31].

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *TOPLAS'11*, 33(1).

[2] ActorFoundry. `http://osl.cs.uiuc.edu/af/`.

[3] G. Agha. Actors: a model of concurrent computation in distributed systems. Technical Report AITR-844, 1985.

[4] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science'85*.

[5] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.'97*, 7(1).

[6] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI'10*.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA'01*.

[9] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02*.

[10] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for Active Objects. In *APLAS '08*.

[11] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP'10*.

[12] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE '13*.

[13] A. Farzan and P. Madhusudan. Causal atomicity. In *CAV'06*.

[14] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP'07*.

[15] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[16] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI'09*.

[17] C. Flanagan and S. Qadeer. Types for atomicity. *TOPLAS'08*, 30(4).

[18] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP'02*, .

[19] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.'05*, 338 (1-3), .

[20] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci. '09*, 410.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM'69*, 12(10).

[22] S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *OOPSLA'12*.

[23] M. Isard and A. Birrell. Automatic mutual exclusion. In *HOTOS'07*.

[24] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *FMSE'06*.

[25] C. B. Jones. Specification and design of (parallel) programs. In *IFIP'83*.

[26] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *OOPSLA'10*.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM '78*, 21(7).

[28] J. Larus and C. Kozyrakis. Transactional memory. *CACM'08*, 51(7).

[29] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*. Addison-Wesley, 1996.

[30] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *CACM'75*, 18(12).

[31] Y. Long, M. Bagherzadeh, E. Lin, G. Upadhyaya, and H. Rajan. Quantification of sequential consistency in actor-like systems: An exploratory study. Technical Report 14-03, 2014.

[32] S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *PPoPP'11*.

[33] T. Nipkow and L. Nieto. Owicki/gries in Isabelle/HOL. In *FASE'99*.

[34] P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.'07*, 375(1-3).

[35] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica'75*, 6(4).

[36] H. Rajan, S. M. Kautz, E. Lin, S. Kabala, G. Upadhyaya, Y. Long, R. Fernando, and L. Szakács. Capsule-oriented programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013.

[37] H. Rajan, S. M. Kautz, E. Lin, S. L. Mooney, Y. Long, and G. Upadhyaya. Capsule-oriented programming in the Panini language. Technical Report 14-08, 2014.

[38] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP '09*.

[39] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE'04*.

[40] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP'05*.

[41] J. Schäfer and A. Poetzsch-Heffter. JCoBox: generalizing active objects to concurrent components. In *ECOOP'10*.

[42] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *OOPSLA '07*.

[43] C. A. Stone, M. E. O'Neill, and T. O. Team. Observationally cooperative multithreading. In *SPLASH'11*.

[44] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *Programming Languages and Systems'14*.

[45] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP'10*.

[46] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA'05*.

[47] J. Yi. *Cooperability: a new property for multithreading*. PhD thesis, 2011.

[48] J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Cooperative types for controlling thread interference in Java. In *ISSTA'12*, .

[49] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *PPoPP'11*, .

A. Static Semantics

Panini's type system distinguishes between two kinds of types: variable types and capsule types. Unlike variable types that can subtype each other, capsule types cannot. This in turn, allows the exact type of the receiver of a global capsule invocation to be statically known.

The type system also ensures that capsule instances cannot be passed as parameters or returned as return values of procedure invocations by requiring them to be of variable types.

A.1 Type Attributes

Panini's typing rules use type attributes of Figure 15. In this figure variable types are unit and reference types and capsule types are capsule names declared in a *Panini* program P.

The typing judgment $P, \Pi, \Gamma \vdash e : \theta$ says that for a program P in the typing environment Γ and store typing environment Π, the expression e has the type θ. The typing environment Γ maps variable names to variable types T and capsule names to capsule types C and the store typing environment Π maps locations to their variable types.

θ ::= type attributes
| T variable types
| C capsule types
T ::=
| **unit** unit types
| **ref**(T) reference types

$\Gamma ::= \{var : T, i : C\}$ variable typing environment
$\Pi ::= \{l : T\}$ store typing environment

$P, \Pi, \Gamma \vdash e : \theta$ typing judgement

where
$C \in \mathfrak{C}$ set of capsule names
$i \in \texttt{self} \cup \mathfrak{I}$ set of capsule instance names
$v \in \mathfrak{X}$ set of variable names
$l \in \mathfrak{L}$ set of locations

Figure 15. Type attributes

The notation $P \vdash \theta$ in *Panini*'s typing rules denotes that θ is a valid variable or capsule type and the notation $\vdash_C$ denotes well-typedness in the context of the declaration of a capsule type C.

A.2 Typing Rules

Figure 16 shows *Panini*'s select typing rules.

(T-CAPSULE DECL) type checks a capsule declaration. It ensures that the declarations of the capsule's imports, design, states and procedures are well typed. An import declaration is well typed if its imported names are of valid capsule types, i.e. $P \vdash D$. A well typed design declaration has well typed instance and wiring declarations. Similar to import declarations, an instance declaration is well typed if its declared name is of valid capsule type, i.e. $P \vdash G$. A state declaration is well typed if it declares a state name of a variable type, i.e. $P \vdash T$. This is because capsule instances cannot be part of state of other capsule instances. Wiring and procedure declarations should type check in the context of imported and locally declared capsule instances in the design declaration.

(T-PROC DECL) type checks a procedure declaration in the context of a capsule type C. It ensures that capsule instances cannot be declared as formal parameters or return type of a procedure, by requiring them to be of variable types This in turn prevents capsule instances to be passed to or returned from procedure invocations in (T-PROC INVOC). The rule also checks that the type of the body e of the procedure is a $<:$ subtype of its return type.

(T-PROC INVOC) type checks a global procedure invocation. It ensures that the receiver i of the invocation is of capsule type C and its actual parameters and return types are of variable types. It also checks that the receiver's capsule type contains the invoked method p and the actual parameters passed to the procedure are subtypes of formal parameters of the procedure. $\Gamma(i)$ returns the type of a capsule name i in the typing environment Γ. Type checking of local procedure invocations in (T-SELF PROC INVOC) is similar.

(T-WIRING DECL) type checks a wiring declaration. It ensures that types of capsule names $\overline{j}$ passed to a wiring declaration, are *the same as* the capsule types declared in the import declaration of capsule type C. This is because there is no subtyping among capsule types in *Panini*.

(T-RESOLVE) type checks a resolve expression. It ensures that the variable type of the expression e is a subtype of the variable type of the location that is going to hold the value of e. $\Gamma(l)$ returns the variable type of the location l in the typing environment Γ.

(T-ε) and (T-$\square$) type check unresolved future value ε and $\square$ value of transferred locations, respectively. These two values can have any arbitrary variable type T. (T-REF) type checks a reference creation expression. It ensures that capsule names cannot be stored in the store, by requiring e to be of a variable type. The same is true in (T-REF ASSIGN).

Soundness Proof of *Panini*'s type soundness follows standard progress and preservation arguments and thus is omitted.

B. Proofs

Theorem 5 *(**Sparse interference in Panini**)*

Proof: Let $read(id)$, $write(id)$, $invoke(id)$ and $resolve(id)$ stand as shorter versions of $read(id,-)$, $write(id,-)$, $invoke(id,-,-,-)$ and $resolve(id,-,-,-)$ of capsule instance id where $-$ denotes irrelevant values that do not matter to the discussion.

Let $\mathscr{T}_{id,p}$ denote a subtrace corresponding to execution of procedure p of capsule instance id. $\mathscr{T}_{id,p}$ starts with the first action of the procedure, a_s, and ends with the resolve action $resolve(id)$ of the procedure, *with actions of procedures of other capsule instances interleaving*. Furthermore, lets partition $\mathscr{T}_{id,p}$ to smaller subtraces $\mathscr{T}_{sub}$ such that actions $invoke(id)$, $resolve(id)$ and a_s only end up at the beginning or end of the subtrace. Subtraces $\mathscr{T}_{sub}$ do not overlap and their concatenation results in $\mathscr{T}_{id,p}$. A subtrace $\mathscr{T}_{sub}$ has one of the following four shapes: (1) it starts and ends with invoke actions $invoke(id)$ with zero or more read and write actions of id, i.e. $read(id)$ or $write(id)$ in between (2) it starts with a_s and ends with an invoke action $invoke(id)$ with read and write actions of id in between; (3) it start with an $invoke(id)$ and ends in $resolve(id)$ with read and write actions of id in between; or (4) it start with a_s and end with $resolve(id)$ with read and write actions of id in between. In any of these subtrace actions of other capsules are interleaving.

In a subtrace $\mathscr{T}_{sub}$ of form (1), using Lemma 2 any read and write actions $read(id)$ and $write(id)$ can be right swapped such that they form a transaction with the invoke action at the end of the subtrace. A transaction is a sequence of actions of a capsule instance that behaves as if executed sequentially with no interference. This in turn means, all neighboring actions of other capsule instances in $\mathscr{T}_{sub}$ move to after the invoke action at the start of the subtrace. In a subtrace of form (2), right swapping read and write actions $read(id)$ and $write(id)$ causes them to form a transaction with the invoke action at the end of the subtrace. Consequently, all neighboring actions in this subtrace moves to before the a_s action, i.e. before the execution of the body of the procedure p. In a subtrace of form (3) right swapping of $read(id)$ and $write(id)$ causes them to form a transaction with $read(id)$ and thus all neighboring actions of other capsules move to after the invoke at the beginning of the subtrace. Finally in a subtrace of form (4), right swap of $read(id)$ and $write(id)$ causes the whole subtrace to form a transaction and all neighboring actions move to before the execution of the procedure p. In other words, all the neighboring actions interleaving with actions of procedure p of capsule instance id can be moved to either before the execution of the procedure or to after invocation actions $invoke(id)$ of the procedure. The interference at the beginning of the trace of the procedure p, can be safely moved out to the trace of its invoking procedure, either to the end of the invoking procedure's trace or after one of its global procedure invocations. This argument could be repeated for execution of all procedure bodies in the trace $\mathscr{T}$ of a *Panini* program P.

Theorem 6 *(**Cognizant interference in Panini**)*

Proof: The proof is by cases on local and global evaluation rules in Figure 8 and the following properties of *Panini* which: limits sharing of memory locations to unresolved future locations with synchronized access; i.e. Lemma 3 and limits accessibility of the state of a capsule instance to only through global invocation of its procedures, i.e. Lemma 4. Let $\texttt{resolve}(e'_1)$ stand as abbreviation for $\texttt{resolve}(-,e'_1,-,-)$ in which $-$ denotes irrelevant values. Capsule instances id' and id are shared among other capsule instances in the global configuration $\mathscr{K}$ and thus could be suscep-

(T-CAPSULE DECL)
$$\frac{\forall proc \in \overline{proc}\,.\ P,\ \overline{i:D},\ \overline{h:G} \vdash_C proc \qquad \forall wire \in \overline{wire}\,.\ P,\ \overline{i:D},\ \overline{h:G} \vdash wire \qquad \forall D \in \overline{D}\,.\ P \vdash D \qquad \forall G \in \overline{G}\,.\ P \vdash G \qquad \forall T \in \overline{T}\,.\ P \vdash T}{P \vdash \texttt{capsule}\ C(\overline{D\ i})\{\ \texttt{design}\{\overline{G\ h}\ \overline{wire}\}\ \overline{T\ f}\ \overline{proc}\ \}}$$

(T-PROC DECL)
$$\frac{P,\Pi,\Gamma,\overline{var:T},\texttt{self}:C \vdash e:T'' \qquad P,\Pi,\Gamma \vdash T' \qquad T'' <: T' \qquad \forall T \in \overline{T}\,.\ P,\Pi,\Gamma \vdash T}{P,\Pi,\Gamma \vdash_C T'\ p(\overline{T\ var})\{e\}}$$

(T-DEREF)
$$\frac{P,\Pi,\Gamma \vdash e:\texttt{ref}(T)}{P,\Pi,\Gamma \vdash e!:T}$$

(T-PROC INVOC)
$$\frac{C=\Gamma(i) \qquad \texttt{capsule}\ C(..)\{..\ T''\ p(\overline{T'\ var'})\{e'\}\ ..\} \in P \qquad \forall e \in \overline{e}\,.\ P,\Pi,\Gamma \vdash e:T, T <: T'}{P,\Pi,\Gamma \vdash i.p(\overline{e}):T''}$$

(T-ε)
$$\frac{P \vdash T}{P \vdash \varepsilon : T}$$

(T-SELF PROC INVOC)
$$\frac{P,\Pi,\Gamma \vdash \texttt{self}:C \qquad \texttt{capsule}\ C(..)\{..\ T''\ p(\overline{T'\ var'})\{e'\}\ ..\} \in P \qquad \forall e \in \overline{e}\,.\ P,\Pi,\Gamma \vdash e:T, T <: T'}{P,\Pi,\Gamma \vdash \texttt{self}.p(\overline{e}):T''}$$

(T-$\square$)
$$\frac{P \vdash T}{P \vdash \square : T}$$

(T-WIRING DECL)
$$\frac{C=\Gamma(i) \qquad \texttt{capsule}\ C(\overline{D\ h})\{..\} \in P \qquad \forall j \in \overline{j}, D \in \overline{D}\,.\ D == \Gamma(j)}{P,\Pi,\Gamma \vdash i(\overline{j})}$$

(T-RESOLVE)
$$\frac{P,\Pi,\Gamma \vdash e:T \qquad T <: \Pi(l)}{P,\Pi,\Gamma \vdash \texttt{resolve}(l,e,id,p):T}$$

(T-STATE-READ)
$$\frac{P,\Pi,\Gamma \vdash \texttt{self}:C \qquad \texttt{capsule}\ C(..)\{..Tf..\} \in P}{P,\Pi,\Gamma \vdash \texttt{self}.f:T}$$

(T-STATE-ASSIGN)
$$\frac{P,\Pi,\Gamma \vdash e:T \qquad \texttt{capsule}\ C(..)\{..T'f..\} \in P \qquad T <: T'}{P,\Pi,\Gamma \vdash \texttt{self}.f := e:T}$$

(T-REFERENCE)
$$\frac{P,\Pi,\Gamma \vdash e:T}{P,\Pi,\Gamma \vdash \texttt{ref}\ e:\texttt{ref}(T)}$$

Figure 16. *Panini*'s select typing rules.

tible to interference. Using Lemma 4, the state of id' can only be accessed and modified through invocation of its procedures.

Case analysis for dynamic semantic rules:

(PROC INVOC): at the interference point $^{\alpha}$, *Panini*'s global procedure invocation along with preemptive and nondeterministic scheduler in (CONGRUENCE) allows any arbitrary number (zero or more) of procedures of id' to be invoked and their bodies, with their formal parameters substituted with their values and `self` substituted with id', to be appended to its queue Q'. Consequently the queue of id' will be of the form $\mathscr{E}[e].Q'.\texttt{resolve}(l,e'_k,id).\{\texttt{resolve}(e'_1),..,\texttt{resolve}(e'_n)\}^*$, in which zero or more bodies of the invoked procedures are appended to the end of the queue.

(FIFO DEQUEUE): at the interference point $^{\alpha}$, *Panini*'s dequeue rule (FIFO DEQUEUE) along with the scheduler (CONGRUENCE), allow an arbitrary number of resolve expressions of procedure bodies to be dequeued from Q' and evaluated.

(OTHER): the global rule (RESOLVE) or other local rules (STATE READ), (STATE ASSIGN), (REF), (DEREF), (REF ASSIGN), (LET BINDING) and (SELF PROC INVOC), do not cause any invocation of procedures of the capsule id' and thus are irrelevant to interfering behavior at $^{\alpha}$.

For the capsule instance id, using Lemma 4, its state can only be accessed and modified through invocation of its procedures. At interference point $^{\alpha}$, any invocation of procedures of id using (PROC INVOC) is appended to the end of its queue Q for later dequeuing using (FIFO DEQUEUE) and local sequential execution and thus does not interfere at $^{\alpha}$. Other dynamic semantic rules do not invoke any procedure on id and thus do not interfere at $^{\alpha}$.

Thus, according to the afore-mentioned case analysis of the dynamic semantics rules, the interfering behavior of other capsule instances at the interference point $^{\alpha}$ is $\theta = \{e'_1,..,e'_n\}^*$.

Lemma 3 (Sharing of unresolved future locations)

Transferred locations with values $\square$ are irrelevant and thus taken out in $dom_{\square}$ of local stores, because any attempt to access them terminates the program. The notation $\veebar$ denotes an exclusive logical disjunction, in which at most of the disjuncts can be true.

Proof : The proof is by cases on *Panini*'s normal and exceptional dynamic semantics rules in Figure 8 and Figure 10, happens-before relations in Figure 12 and Lemma 1:

Initial configuration Using Lemma 1 there is no shared location among capsule instances in the initial configuration, i.e. $A = \emptyset$, and thus the lemma holds.

Dynamic semantics Rules (PROC INVOC) and (RESOLVE), transfer ownership of memory locations among local stores of the invoking and invoked capsule instances. For each procedure invocation, these rules share a fresh future location among the invoking and invoked capsule instances, and set the value of the shared location in the local store of the invoking instance to ε. However, the condition $R \cap R' = \emptyset$ in these two rules prevents them to share other memory locations, upon transferring ownership of parameters of the procedure invocation or returning its result.

The rule (REF) allocates a fresh location in the local store of a capsule instance and thus do not cause any sharing.

The rules (DEREF) and (REF ASSIGN) block when attempting to read or write an unresolved future location with the value ε. This means for an unresolved future location l with value ε in the capsule instance Σ, its read action $a = read(id,l)$ does not unblock and happen unless the value of the future location is resolved by the capsule instance Σ; any write action $a' = write(id',l)$ of the location l by the capsule instance id should happen before the location is resolved, because of ownership transfer after resolve. That is $a' \prec resolve(id',id,l,-) \prec a$ which in turn means $a' \prec a$ because of the transitivity of the happens-before relation [49]. In other words, there is a happens-before relation between a and a' and thus they are synchronized. The same applies to other combination of a and a' actions in (ii).

Other rules in *Panini*'s normal and exceptional dynamic semantics, do not cause any transfer of ownership or memory allocation.

Lemma 4 (Global accessibility through procedures)

Proof : Using Lemma 3, there is no shared location among local stores S and S' of capsule instances, except future locations for returning the result of procedure invocations among capsules.

Let l be a shared future location among capsules id and id' when id' invokes a procedure of id. The future location cannot

be used to modify the state of *id* because, upon the invocation, the rule (PROC INVOC) guarantees that the future location is fresh and thus does not point to any state of *id*; during the execution of the procedure body, the rules (DEREF) and (REF ASSIGN) ensure that any attempts to access the location in *id* blocks until the future is resolved; and after the future location is resolved, the rule (RESOLVE) ensures that the resolved future does not point to any state of *id* or locations reachable from it.

Lemma 2 (Panini action's mover properties) *Let $\mathscr{T}$ be the execution trace of an arbitrary Panini program P.*

Then, in trace $\mathscr{T}$ read and write actions $read(id,l)$ and $write(id,l)$ of a capsule instance id of a memory location l are right movers, as defined in Definition 4; a global invocation action $invoke(id,id',p,l)$ of a procedure p from the invoking capsule id to the invoked capsule id' and result l is a non-mover; a resolve action $resolve(id,id',p,l)$ associated with this global invocation is a left mover.

Proof : Let a be an action with left and right neighbors a_l and a_r respectively in the subtrace $a_l \hookrightarrow a \hookrightarrow a_r$. We replace a with read, write, invoke and resolve actions of a capsule instance *id* to show their mover properties in an arbitrary trace with arbitrary left and right neighbor actions from other capsule instances.

In a subtrace $a_l \hookrightarrow read(id,l) \hookrightarrow a_r$, the read action of a location l conflicts with a left neighbor $resolve(id,id',p,l)$ action of the same location. This is because swapping the read action with its left neighbor allows reading a future location even before it is resolved. However, *Panini*'s happens-before relations, in Figure 12, does not allow this by ensuring that a future location is resolved before it is read or otherwise it blocks, i.e. $resolve(id,id',p,l) \prec read(id,l)$. This in turn means the read action cannot be a left mover. Since resolving of a future location must happen before its read, a resolve action $resolve(id,id',p,l)$ cannot be right neighbor to the read action $read(id,l)$ and thus the read action can be safely swapped with any of its right neighbors, i.e. the read action is a right mover. The same argument applies to a write action $write(id,l)$ and thus a write action is a right mover too.

Similarly, a resolve action $resolve(id,id',p,l)$ in a subtrace $a_l \hookrightarrow resolve(id,id',p,l) \hookrightarrow a_r$ only conflicts with read and write actions from and to the same location l, i.e. $read(id,l)$ and $write(id,l)$. Again, based on *Panini*'s happens-before relation a read or write of a future location happens only after the future is resolved and thus the read and write actions of a future location cannot be left neighbors to their resolve actions. This in turn means that the resolve action is a left mover and not a right mover.

An invoke action $invoke(id,id',p,l)$ only conflicts with another invoke action if they both invoke procedures on the same capsule instance id', since they both modify the queue of the capsule id'. In a subtrace $a_l \hookrightarrow invoke(id,id',p,l) \hookrightarrow a_r$ the invocation action cannot be safely swapped with neither its left nor its right neighbors and thus is a non-mover. This is because they neighbors could be invocation actions on the same capsule instance id'.

It is worth to note that local actions as well as read and write of non future locations are both movers.

Finally, in (X DEREF) and (X REF ASSIGN), trying to dereference or assign to a transferred location not owned by the capsule instance anymore, causes the program to throw an ownership transfer exception **OWE** and terminate. In (X PROC INVOC) and (X RESOLVE), the program terminates by throwing an ownership transfer exception upon any attempts to leak the states of an instance or its reach by passing them or returning them from a global procedure invocation.