A Formal Reference for SCOOP

Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland,
`firstname.lastname@inf.ethz.ch`,
`http://se.inf.ethz.ch/`

Abstract. Operational semantics is a flexible but rigorous means to describe the meaning of programming languages. Small semantics are often preferred, for example to facilitate model checking. However, omitting too many details in a semantics limits results to a core language only, leaving a wide gap towards real implementations. In this paper we present a comprehensive semantics of the concurrent programming model SCOOP (Simple Concurrent Object-Oriented Programming). The semantics has been found detailed enough to guide an implementation of the SCOOP compiler and runtime system, and to detect and correct a variety of errors and ambiguities in the original informal specification and prototype implementation. In our formal specification, we use abstract data types with preconditions and axioms to describe the state, and introduce a number of special operations to model the runtime system with our inference rules. This approach makes our large formal specification manageable, providing a first step towards reference documents for specifying concurrent object-oriented languages based on operational semantics.

1 Introduction

Concurrent programming has become an important part of mainstream software development, caused by the widespread use of multicore processors. The notorious difficulty of writing concurrent programs correctly has on the other hand spawned work into novel language abstractions to express concurrency and synchronization. One such language is SCOOP [21,25], an object-oriented programming model for concurrency based on the idea of contracts.

The main idea of SCOOP is to simplify the writing of correct concurrent programs for developers, who can use familiar concepts from object-oriented programming but are protected by the model from common concurrency errors such as data races. This is achieved by a runtime system that automatically takes care of operations such as obtaining and releasing of necessary locks, without the need for explicit program statements. While being based on conceptually simple ideas, the semantics of the language concepts and runtime system turns out to be very complex.

The question is therefore how the semantics can be properly documented. The initial version of SCOOP has been defined in [21], where all the main concepts are outlined but implementation aspects are neglected for the most part. A first prototype implementation was then introduced in [25], where the semantics was described only informally, with the exception of a formalization of the type system. In this paper we provide a full formalization of the operational behavior of SCOOP, specified by a structural operational semantics. The main contributions of the paper are:

- A formal specification of SCOOP that treats all important language elements.
- Clarification and correction of the informal specification in [25].

This work does not provide a formal completeness and soundness proof with respect to an axiomatic semantics. Sec. 6 discusses this possibility as part of future work. This work focuses on a formal reference for a concurrent programming language. We argue that this formal reference reflects and corrects the informal description by following a systematic approach.

This article is a condensed version of our technical report [24] on the same subject. This paper is structured as follows. The remainder of this introduction gives a brief overview of the main ideas of the SCOOP model to provide a basic intuition for the main part of the paper. Sec. 2 gives an overview of related work. Sec. 3 gives an overview of the considered language. The two following chapters contain the main parts of the formalization: Sec. 4 describes the state formalization and Sec. 5 the formalization of computations. Sec. 6 concludes and discusses future applications of the formalization.

1.1 An Informal Overview of SCOOP

The starting idea of SCOOP is that every object is associated for its lifetime with a processor, called its *handler*. A *processor* is an autonomous thread of control capable of executing actions (features) on objects. A processor can be a hardware CPU, but it can also be implemented in software, for example as a process or as a thread; any mechanism that can execute instructions sequentially is suitable as a processor.

A reference variable belonging to a processor (for example, a field of an object handled by that processor) can point to an object with the same handler, or to an object on another processor. In the second case the reference is said to be *separate*. The semantics of a call *x.f* depends on this distinction: if *x* is not separate (as always in sequential programming), the call is synchronous; if *x* is separate, meaning that it points to an object handled by a different processor, that processor will execute the call asynchronously. This possibility of asynchronous calls is the main source of concurrent execution.

The producer-consumer problem serves as a simple illustration of these ideas. A root class defines the entities *producer* and *consumer*. The keyword **separate** specifies that these entities may be handled by a processor different from the current one. A creation instruction on a separate entity such as *producer* and *consumer* will create an object on another processor; by default the instruction also creates that processor.

```
producer: separate PRODUCER
    -- The producer.
consumer: separate CONSUMER
    -- The consumer.
```

Both the producer and the consumer access an unbounded buffer:

```
buffer: separate BUFFER [INTEGER]
        -- The data structure for exchanging objects between the producer and the
           consumer.
```

Both the producer and the consumer need to access the buffer, in calls such as *buffer.put* (*x*) and *buffer.item*. The basic SCOOP rule to ensure mutual exclusion and guarantee the absence of data races is that any target that is declared as separate, such as *buffer*, must be an argument of an enclosing routine, which in turn guarantees that this routine has exclusive access to the corresponding separate object for the duration of its execution. The SCOOP scheduler locks the processors handling all objects corresponding to these *controlled* arguments. This rule prevents any data races on the group of controlled objects. For example, in a call *consume* (*buffer*), the buffer is controlled; the call gets exclusive access to its handler.

Condition synchronization relies on preconditions (after the **require** keyword) to express wait conditions. Any precondition of the form *x.some_condition* will make the execution of the routine wait until the condition is true. For example, the precondition of the *consume* routine ensures that the routine will wait until the buffer is not empty.

```
consume (buffer: separate BUFFER [INTEGER])
        -- Consume an item from the buffer.
    require
        not (buffer.count = 0)
    local
        consumed_item: INTEGER
    do
        consumed_item := buffer.item
    end
```

During a feature call, the consumer processor could pass its locks to the buffer processor if it has a lock that the buffer processor requires. This mechanism is known as *lock passing*. In such a case, the consumer processor would have to wait for the passed locks to return. For the feature call *buffer.item*, the buffer processor does not require any locks from the consumer processor. Hence, the consumer processor does not have to wait due to lock passing. However, the runtime system ensures that the result of the call *buffer.item* is properly assigned to the entity *consumed_item* using a mechanism called *wait by necessity*: while the consumer processor usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result of this call.

As the buffer is unbounded, the corresponding producer routine does not need a wait condition; mutual exclusion will be ensured as before:

```
produce (buffer: separate BUFFER [INTEGER])
        -- Produce an item and put it into the buffer.
    local
        produced_item: INTEGER
```

```
do
    produced_item := new_item
    buffer.put (produced_item)
end
```

The asynchronous nature of separate calls such as *buffer.put* (*x*) implies a distinction between the notion of *feature call* and *feature application*. In sequential programming, executing a call means executing the corresponding feature immediately. With asynchronous calls, the client processor logs the call with the supplier processor (feature call) and moves on. Only at some later time will the supplier processor actually execute the body (feature application).

The main part of the paper defines formally the implementation that gives rise to the behavior outlined above. It also introduces advanced concepts and additional language elements, which cannot be covered in a brief overview, and shows how these give rise to a complexity which can only be dealt with satisfactorily with a formal specification.

2 Related Work

The discussion is divided into work on SCOOP and work on other languages.

2.1 Approaches for SCOOP

In his dissertation, Nienaltowski [25] worked out the details of an implementation of SCOOP as suggested by Meyer [21], and provided a prototype implementation. The language semantics is described informally only, with the exception of the type system which is defined using an inference system. The informal description and the prototype contain various ambiguities and omissions, which we are able to clarify.

Torshizi et al. [33] have defined and implemented JSCOOP, a version of the SCOOP model for the Java language. Only the most important language elements are considered, and no attempt at formalization is made. In contrast, our specification and implementation [31] on top of Eiffel considers all language elements. We believe that our specification could help to extend JSCOOP to a full treatment of the language concepts.

Brooke, Paige and Jacob [5] have used CSP [13] to give a semantics to SCOOP as described by Meyer [21]. Their initial hope was to use tools for analyzing CSP specifications, such as FDR, to automatically check for deadlock in SCOOP programs, but found the size of the specification prohibitive. A benefit of their approach is that CSP provides the machinery needed to express concurrency and synchronization, leading to a relatively concise model. Our goal is to provide formal descriptions close to an actual implementation, and therefore prefer to design an own operational semantics, rather than going through the indirection of another process algebra.

Structural operational semantics, introduced by Plotkin [29], is a flavor of operational semantics that has been used with great success to define various concurrent systems. Our specification uses this style of semantics as well. To model SCOOP we also make use of established modeling concepts from process algebra, such as the notion of channels, which is present in most calculi such as CSP [13] or the π-calculus [23]. We use the theory of abstract data types (ADT) [18] to model the elements of a program text and to model the state of a SCOOP program.

Ostroff et al. [28] describe a structural operational semantics for SCOOP in the refined version by Nienaltowski [25]. This operational semantics inspired our work, and we have attempted to stay close to their modeling ideas where possible, so that [28] can be viewed as a reduced version of the semantics we describe in this paper. While [28] covers some of the most significant aspects of SCOOP, it falls short of describing a number of other critical language concepts: in their reduced model, a query routine handled by some processor *p* must not make calls to a processor other than *p*; lock passing, expanded objects and the import mechanism, once routines, evaluation of (asynchronous) postconditions and invariants, and explicit processor tags are not considered. We clarify these aspects in this paper. Furthermore, [28] have pursued the goal to check temporal logic properties of SCOOP programs using their semantics and the SPIN model checker, but were limited to small programs by state space explosion. We have the different goal of providing a reference document for SCOOP, and thus don't have to sacrifice coverage of the language for keeping the specification small.

2.2 Approaches for Other Concurrent Programming Languages

Axum [22] is a concurrent programming language based on the actor model. In Axum, actors are called agents. An agent is an isolated runtime component that executes in parallel with other agents. The agents communicate with each other by sending messages through channels. Each channel has input ports, output ports, and a protocol. The ports are queues of messages. The protocol is a state machine that defines how the channel behaves. Schemas define the structure of messages. Besides message passing, Axum also provides domains – shared state between groups of actors. Erlang [10] and Scala [27] are further examples of actor-based programming languages.

Cω [3] is an extension of C# that integrates elements of the Join Calculus [11]. Cω allows computations to be spawned off into different threads using asynchronous methods: while for synchronous methods the caller must wait until a routine completes, asynchronous methods return immediately while their body is scheduled for execution in another thread. Cω supports so-called chords, which associate the body of a routine with more than one method; the body is executed only if all methods have been called.

Another language is Cilk [4], which extends C with concurrency concepts. A method marked with the cilk keyword can be asynchronously spawned with the spawn keyword. The sync keyword requires the current method to wait for all previously spawned tasks to complete. An inlet function within a parent method receives the result of a spawned child method; the inlet functions of a parent method are guaranteed to execute atomically. Within an inlet function, the abort keyword tells the scheduler that any other child method spawned by the parent method can be aborted. Cilk also implements a work stealing mechanism to achieve high performance by dividing method executions efficiently among processors.

Ada [14] defines tasks – units that can run in parallel. A task is declared within a procedure; it consists of a specification and an implementation. The task is activated when the procedure starts executing. The task specification can define a number of entry points with parameters; an entry point specifies an action the task can synchronize on. An accept statement within the task body indicates the point where the rendezvous can take place. Another task calls the entry point to take part in the rendezvous. With a select statement, one can wait for multiple entry points; alternatives may be guarded with boolean expressions. Ada defines protected objects – a monitor-like construct with guards instead of conditional variables. A protected object is declared within a procedure; it has a specification and an implementation.

The occam programming language [32] builds on the CSP process algebra [13]. A parallel construct defines a number of processes that execute concurrently; the parallel construct terminates when all spawned processes terminated. Processes communicate with each other through named channels. The alternation construct defines a number of processes, where only one of them gets executed; a guard defines when a process can be executed.

X10 [7], Fortress [2], and Chapel [15] are based on the Partitioned Global Address Space (PGAS) model. PGAS uses a global shared memory. It defines portions on the global shared memory and associates them to specific processors to improve performance and scalability. X10 provides important abstractions such as places, asynchronous methods, future invocations, and barriers. However, it places a considerable burden on programmers. Fortress offers implicit parallelization of loops and operations on data structures. Chapel provides a higher-level multithreaded parallel programming model with abstractions for data parallelism, task parallelism, and nested parallelism.

Linda [12] is a coordination language to connect concurrent components; the components can be written in different programming languages. The coordination is based on a tuple space, which holds data tuples that can be stored and retrieved by the processes. Pattern matching is used to read and remove tuples; the operations block until a matching tuple is found. The eval construct creates a new process to evaluate an expression; the new process writes the evaluation result into the tuple space. Implementations of Linda can be found in several programming languages such as Java and C.

For the related languages mentioned above, we are not aware of rigorous behavioral specifications, with the exception of Cω and occam, which use the Join Calculus respectively CSP as the underlying model. For multi-threaded Java however, such formalizations have been attempted.

Ábrahám, de Boer, de Roever, and Steffen [1] present an operational semantics for a subset of multi-threaded Java. They focus on the most important multi-threaded aspects, i.e., dynamic thread creation, thread termination, and re-entrant monitors. The semantics consists of two components: the semantics for isolated objects and the semantics for interacting objects. The authors want to use the semantics to develop a proof system that is based on an existing proof-system for isolated objects. A configuration is a set of instance configurations. An instance configuration contains the attribute values of one object. It also contains the local environment and the expression of each thread that is concurrently executing within the object. In modeling the state of a program, our semantics strictly separates the actions to be executed from the data. This makes it easier to derive implementations from the semantics because an implementation is likely to keep the program text and data separate. Ábrahám et al. use transition labels to synchronize inference rules. The labels allow an external observer to follow the transitions.

Our semantics is a pure reduction semantics without labels because we do not require observable transitions.

Cenciarelli, Knapp, Reus, and Wirsing [6] also describe an operational semantics for a larger subset of multi-threaded Java. They cover a larger number of multi-threaded aspects than [1]. In particular they formalize Java's notification mechanism and the working memory. A configuration consists of a function that maps each thread to its expression and its local environments. The configuration also has a container with the objects and the static typing information. Lastly, the configuration consists of an event space. The event space is a partially ordered set of events that have been executed by the threads. The ordering reflects the order in which the events took place. An event space serves two purposes. First, it contains certain aspects of the state. For example, the lock and unlock actions tell us which thread owns which lock. Second, it records the history. A number of constraints state when an event space is valid. Hence, the event space indicates which further actions can take place. The authors use two different validity constraints for both Java's non-prescient semantics and its prescient semantics. Using this, they show that any prescient execution of a properly synchronized program can be simulated by a non-prescient execution. Compared to our semantics, there is no clean division between program text and the state and there is no clean division between the state and the typing information.

Lochbihler [19] suggest a different operational semantics for a large subset of multi-threaded Java. Just like [6], he covers the notification mechanism, but he does not formalize the working memory. He defines an instantiating semantics based on an extension of Jinja [17]. Jinja is an operational semantics for a subset of single-threaded Java. The instantiating semantics is used for the sequential case. Lochbihler defines a generic formal framework to lift the instantiating semantics to the concurrent case. The configuration of the instantiating semantics consists of the expression, a container with the objects, and the local environments. The state of the framework semantics consists of the lock status, the thread information with the thread's expression along with the thread's local environments, a container with the objects, and the wait sets. Lochbihler formalizes the notion of deadlocks, where deadlocks are either based on locks or on wait sets. He then proves that every program that satisfies certain criteria either produces a final value, throws an exception, or deadlocks. He also shows that every such program preserves type safety.

3 Language Overview

SCOOP is a programming language based on Eiffel, an object-oriented programming language, defined in the Eiffel ECMA standard [9]. SCOOP's concurrency model can be applied to other programming languages as well. For this reason, this work does not focus on SCOOP, but on its concurrency model. This section defines a subset of SCOOP, reduced to the parts that are relevant for the concurrency model. It presents the syntax of the subset and a list of simplifications. It then discusses the program representation that this formalization assumes.

3.1 Syntax

The following EBNF grammar defines the set of all considered programs:

```
program = class_declaration∗ root_procedure_declaration ;
root_procedure_declaration = {class_name}.routine_name ;
class_declaration =
  [”expanded”] ”class” class_name
    ”inherit” class_name
    [”create” routine_name {”,” routine_name}]
    ”feature” [”{” class_name {”,” class_name} ”}”] {feature_declaration}
    [”invariant” expression]
  ”end” ;

feature_declaration = routine_declaration | attribute_declaration ;
routine_declaration =
  routine_name [”(” entity_declaration {”,” entity_declaration} ”)”] [”:” type]
    [”require” expression]
    [”local” entity_declaration {entity_declaration}]
    (”do” | ”once”)
      instruction {”;” instruction}
    [”ensure” expression]
    ”end” ;
attribute_declaration = entity_declaration ;
entity_declaration = entity_name ”:” type ;

instruction =
  entity_name ”:=” expression |
  expression ”.” feature_name [”(” expression {”,” expression} ”)”] |
  ”create” entity_name ”.” routine_name [”(” expression {, expression} ”)”] |
  ”if” expression ”then” instruction {”;” instruction} ”else” instruction {”;”
      instruction} ”end” |
  ”until” expression ”loop” instruction {”;” instruction} ”end” ;

expression =
  literal |
  entity_name |
  expression ”.” feature_name [”(” expression {, expression} ”)”] ;
literal = boolean_literal | integer_literal | character_literal | void_literal ;
boolean_literal = ”True” | ”False” ;
integer_literal = [”–”](”0” | . . . | ”9”) {”0” | . . . | ”9”} ;
character_literal = ” ’ ” ”a” | . . . | ”z” | ”A” | . . . | ”Z” | ”0” | . . . | ”9” ” ’ ” ;
void_literal = ”Void” ;

type =
  [detachable_tag]
```

[**”separate”**] [*explicit_processor_specification*]
class_name [*actual_generics*] ;
detachable_tag =
”attached” | **”detachable”** ;
explicit_processor_specification =
qualified_explicit_processor_specification |
unqualified_explicit_processor_specification ;
qualified_explicit_processor_specification =
”<” *entity_name* **”.”** **”handler”** **”>”** ;
unqualified_explicit_processor_specification =
”<” *entity_name* **”>”** ;

class_name = *name* ;
feature_name = *routine_name* | *entity_name* ;
routine_name = *name* ;
entity_name = *name* | **”Result”** | **”Current”** ;
name = (**”a”** | ... | **”z”** | **”A”** | ... | **”Z”**) {**”a”** | ... | **”z”** | **”A”** | ... | **”Z”**};

A *class* consists of a number of features. A *feature* is either a *routine* – a sequence of instructions – or an *attribute* – a data storage. If a routine returns a result, then it is called a *function*; otherwise, it is called a *procedure*. If a routine is marked as a once routine (**once** keyword), then the routine gets executed only once in a given context. Functions and attributes are also called *queries*; routines are also called *commands*. A routine can define a *precondition* (**require** keyword) and a *postcondition* (**ensure** keyword). The enclosing class can define an *invariant* (**invariant** keyword). Each feature can be exported to a list of classes, so that only these classes can use the feature. A number of procedures are dedicated *creation procedures*. These procedures can be used in creation expression (**create** keyword) to create new objects. A class can be marked as an *expanded class* (**expanded** keyword). Objects of expanded classes get copied when they get passed around; objects of non-expanded classes get aliased.

Formally, a type t is a triple (d, p, c). The component d is the *detachable tag*, p is the *processor tag*, and c is the *class type*. The detachable tag d captures detachability. An entity of attached type (**attached** keyword), i.e., $d = !$, is statically guaranteed to store a value, i.e., to be non-void. An entity of detachable type (**detachable** keyword), i.e., $d = ?$, can be void. As discussed later, the detachable tag is also used for the selective locking mechanism to prevent a request queue from being locked. The processor tag p captures the locality of objects accessed by an entity of the type t. The processor tag p can be separate (**separate** keyword without explicit processor specification), i.e., $p = \top$. The object attached to the entity of the type t is potentially handled by a different processor than the current processor. The processor tag p can be explicit (**separate** keyword with explicit processor specification), i.e., $p = \alpha$. The object attached to the entity of the type t is handled by the processor specified by α. The processor tag p can be non-separate (no **separate** keyword), i.e. $p = \bullet$. The object attached to the entity of the type t is handled by the current processor. The processor tag p can denote no processor, i.e., $p = \bot$. It is used in the type of the void reference. The explicit processor tag either has an unqualified or a qualified specification. An *unqualified explicit processor specification*, i.e., $< p >$, is based on a processor attribute p. The processor attribute p must have the type $(!, \bullet, PROCESSOR)$ and it must be declared in the same class as the explicit processor specification or in one of the ancestors. The processor denoted by this explicit processor specification is the processor stored in p. A *qualified explicit processor specification*, i.e., $< e.handler >$, relies on an entity e occurring in the same class as the explicit processor specification or in one of the ancestors. The entity e must be a non-writable entity of attached type and the type of e must not have a qualified explicit processor tag. The processor denoted by this explicit processor specification is the same processor as the one of the object referenced by e. Explicit processor tags support precise reasoning about object locality. Entities declared with the same processor tag represent objects handled by the same processor. The absence of both the keywords is treated as if there was an **attached** keyword.

3.2 Simplifications

This work makes the following simplifications:

- It does not consider unqualified feature calls. It expects all feature calls to be in the qualified form. This includes accesses to attributes of the current object in expressions.
- It does not consider infix feature calls. It expects all feature calls in the non-infix form. For example, an expression $x > y$ must be transformed into the equivalent form $x.is\_greater(y)$.
- It simplifies the automatic initialization of entities. All entities, except for the current object entity, are initialized with the void reference.
- It neglects exception handling. The exception handling mechanism for SCOOP is still under development.
- It does not consider garbage collection because garbage collection is not refined in the SCOOP model.
- It does not consider agents. From this work's point of view, agents are normal objects.

3.3 Intermediate Representation

For the purpose of the formalization, this work assumes that a program is given in an enriched intermediate representation, where the syntactical elements are replaced with instances of abstract data types. In particular, it assumes ADTs for class types, features, expressions, and instructions. Fig. 1 summarizes these ADTs. The instances of **CLASS_TYPE** are all possible class types, i.e., the types directly defined by all non-generic classes and all possible generic derivations of all possible generic classes. Sec. 4.1 discusses how to get these instances. The ADT **CLASS_TYPE** defines a name query *name*. Each class type can either be a reference class type or an expanded class type. The queries *is_ref* and *is_exp* provide this information. Each class type defines a number of features. These features can be divided into attributes, functions, and procedures. An attribute of an object stores a value. A function performs a computation and

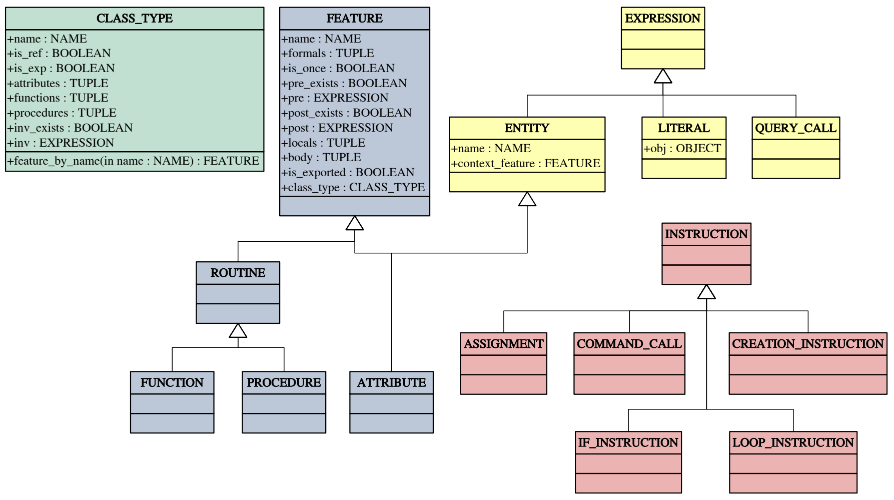

Fig. 1. ADTs for the intermediate representation

returns the result. This computation must not modify the state. A procedure performs a computation that modifies the state. Functions and procedures are also known as routines. For each of these categories, **CLASS_TYPE** defines a query that returns a tuple of features. The query *attributes* returns a tuple of attributes, the query *functions* returns a tuple of functions, and the query *procedures* returns a tuple of procedures. If the name of a feature is known, then the query *feature_by_name* can be used to get the feature with that name. Each class type can have an invariant. The query *inv_exists* indicates whether such an invariant exists. In case an invariant exists, it can be accessed with the query *inv* as an expression. One of the instances of **CLASS_TYPE** is *BOOLEAN*. This class type is expanded and it has an attribute with name *item*. The value of this attribute is the represented boolean value, i.e., an instance of **BOOLEAN**.

In this formalization, a feature is an instance of **FEATURE**. The name of the feature can be retrieved with the query *name* and the formal arguments are given by the query *formals* that returns a tuple with the formal arguments as entities. Whether or not the feature is a once feature can be determined using the query *is_once*. The queries *pre* and *post* return an expression for the precondition respectively the postcondition, provided that the queries *pre_exists* and *post_exists* indicate that the assertions exist. Next, there is the query *locals* that gives the locals of the feature as entities. The query *body* returns the body of the feature as a tuple of instructions. Each feature is either exported or not. A non-exported feature is only available in calls on the current object within the class that declared the feature. An exported feature can be called by other clients as well. The query *exported* returns whether a feature is exported or not. Lastly, each feature has a link to the class it belongs to. This is given by the query *class_type*. This can be used for example to retrieve the invariant that must be preserved by a feature. For each feature category, there is an ADT that inherits from the **FEATURE** ADT.

Expressions can either be entities, literals, or query calls. Every expression is an instance of **EXPRESSION**. For each form of expression, there is one ADT that inherits from the **EXPRESSION** ADT. For entities there is an ADT with a query *name* that returns the name of an entity. A query *context_feature* links an entity to the feature in which the entity is declared. A literal is a character sequence that represents a constant value. As such, literals count as manifest expressions - programming constructs whose values can be deduced by the compiler statically. Literals are instances of an ADT **LITERAL**. This ADT has instances for booleans, integers, characters, and the void literal. Each literal except the void literal can be translated into an object with the query *obj*. This object matches the literal in both type and value. The following notation describes instances of **EXPRESSION**:

$$e \triangleq w \mid b \mid e.f(e,\ldots,e)$$

Here, w is an element of **LITERAL**, b is an instance of **ENTITY**, and f is an instance of **FEATURE**. For instructions, there is an ADT **INSTRUCTION** and an ADT for each kind of instruction. The following notation describes such instances:

```
$h \triangleq$
    $b := e \mid$
    $e.f(e,\ldots,e) \mid$
    create $b.f(e,\ldots,e) \mid$
    if $e$ then $[s\{;s\}*]$ else $[s\{;s\}*]$ end $\mid$
    until $e$ loop $[s\{;s\}*]$ end
```

Here, s stands either for an instance of **INSTRUCTION** or an operation. Instructions are actions that occur in the intermediate representation (user syntax). Operations are actions that do not explicitly occur in the intermediate representation (run-time syntax).

This work builds on an existing type system formalization. It assumes the existence of a typing environment that can be queried for type information.

4 State Formalization

This section provides a formalization of the state of a SCOOP program. This is necessary to describe the effect of SCOOP constructs on the state. The discussion starts with the general approach and continues with the description of the state.

4.1 General Approach

This work considers the state of a SCOOP program to be a data structure that can be created, modified, and queried through features. For the specification of the state, this work uses Liskov's ADT theory [18]. The discussion begins with a justification and the consequences of this choice. The discussion finishes with an explanation on how to get types for elements in the intermediate representation.

Abstract data types Meyer's work on a three-level approach to the description of data structure [20] defines three levels on which a data structure can be described: functional, constructive, and physical. The functional specification is an algebraic approach that uses an implicit characterization of the data structure. The constructive specification provides a means to construct instances of the data structure. The instances constructed like this are mathematical entities. A physical description describes the layout of instances in memory. The constructive specification can be derived from the functional specification and the physical description can be derived from the constructive description.

This work models the state as an ADT instance, on the functional level in the hierarchy described above. This has several reasons. First of all, ADT theory allows us to describe the state on an abstract level without dealing with aspects of the implementation. The constructive and the physical level can be derived from the ADTs on the functional level. Second, ADT theory allows us to modularize the state. Different concerns of the state can be modeled as individual ADTs, while a single ADT can be used to consolidate the individual ADTs. This improves understandability and maintainability of the state description. Lastly, ADT theory is well established and suitable for the task at hand.

An ADT t consists of queries, commands, and constructors. A query of t provides information about an instance of t. The query takes as a first argument the target of type t, which is the instance to be queried. Next to the target, the query can take further arguments with types $t_1, \ldots, t_n$. Finally, the query returns a result of a type t_{n+1}. The declaration of this query is written as $query\colon t \to t_1 \to \ldots \to t_n \to t_{n+1}$. For flexibility reasons, this work uses the curried form (as in Haskell) instead of the equivalent Cartesian form $query\colon t \times t_1 \times \ldots \times t_n \to t_{n+1}$. A command of t returns an updated instance according to the command's semantics. The declaration of a command looks much like the one of a query. However, the result of the command is an instance of t. To simplify the discussions, the following terminology is used: an *update of an ADT instance* is the act of calling a command on the instance; the *updated instance* is the result of the command. A constructor of t creates a new instance of t. In contrast to queries and commands, a constructor does not take the target as the first argument because its purpose is to create a new instance.

To describe an instance of an ADT, one can build an expression that starts with a constructor call. This expression can then be used as the first actual argument of a command call. The resulting expression can then be used as the first actual argument of the next command. This leads to a nested expression, in which the first feature call is in the root of the expression and the last feature call is on the outside of the expression. The instance described in such a way can then be queried. We find this functional notation hard to read. Therefore we use an equivalent object-oriented notation in which the first feature call is on the left and the last feature call is on the right. The main idea is not to write targets as arguments, but to write a target in front of the feature name and to use a dot to separate the two parts from each other. This leads to the following translation between the functional notation and the object-oriented notation:

– The query expression $query(e_0, e_1, \ldots, e_n)$ written in functional notation is equivalent to the expression $e_0.query(e_1, \ldots, e_n)$ written in object-oriented notation.

- The command expression *command*$(e_0, e_1, \ldots, e_n)$ written in functional notation is equivalent to the expression e_0.*command*$(e_1, \ldots, e_n)$ written in object-oriented notation.
- The creation expression *constructor*$(e_1, \ldots, e_n)$ for an instance of an ADT t written in functional notation is equivalent to the expression *new* t.*constructor*$(e_1, \ldots, e_n)$ written in object-oriented notation.

The identity of an ADT instance is given by its query values. Hence, the following holds for all ADTs t: *new* t.*constructor*$(e_1, \ldots, e_n) =$ *new* t.*constructor*$(e_1, \ldots, e_n)$.

Example 1 (Functional notation versus object-oriented notation). The expression in functional notation *is_empty*(*pop*(*push*(*new* **STACK**[**PROC**].*make*, p))) can be written in object-oriented notation as *new* **STACK**[**PROC**].*make*.*push*(p).*pop*.*is_empty*.

Each feature can have a precondition that must be satisfied before the feature gets called. A precondition is expressed as a number of assertions on the target and the arguments. A feature with a precondition is a partial feature. A partial feature is a feature whose domain is restricted. Such a partial feature is indicated with a crossed arrow $\nrightarrow$ after the type of each formal argument that got restricted by the feature's precondition. Non-restricted formal arguments are indicated with a normal arrow $\rightarrow$. The effect of an ADT command is described in a number of axioms. This work deviates from the practice of bundling all axioms for a specific ADT. Instead, all the axioms for a specific feature occur in the feature's declaration. Note that this work does not aspire a sufficiently complete ADT because this would lead to rule explosion. An ADT is sufficiently complete if its axioms make it possible to reduce any query expression to a form that does not involve an instance of the ADT. This requires that the axioms describe the effect of each command on each query. This work follows the practice to describe the effect of each command of an ADT on all the queries of the ADT that have been changed by the command. Unmentioned queries are unchanged.

Example 2 (Command declaration). The following declaration shows a command to set the value of an attribute f of an object o to the value v. The value can either be a reference or a processor. The command takes the object as the target and returns an updated object whose attribute value is set.

```
set_att_val: OBJ → FEATURE ↛ REF ∪ PROC → OBJ
  o.set_att_val(f, v) require
    o.class_type.attributes.has(f)
  axioms
    o.set_att_val(f, v).att_val(f) = v
```

The command states in its precondition that the class type of the target object o must have an attribute f. This is expressed as an assertion after the require keyword. The part in front of the require keyword gives names to the target and the arguments. Note that the precondition makes the command partial. The updated object has the value of its attribute f set to v. This is stated as an axiom after the ensure keyword.

So far the discussion covered queries, commands, and constructors for ADTs. This work extends the ADT theory with the notion of auxiliary features. Auxiliary features are convenience features that are not essential for the definition of the ADT, but nevertheless useful.

The remainder of this work declares various ADTs to model the state of a SCOOP program. Unless it would create confusion, it uses the same name for an instance of an ADT and the corresponding domain element. For example, the instance of the ADT **OBJ** is called an object.

Identifier management This formalization models objects, references, and processors. All of these domain elements have an identity. These identities are automatically managed by the runtime system. The work by Khoshafian and Copeland [16] on different levels of object identity provides good reasons for this decision. They introduce a scale that starts with identities given by the value, goes on with user-supplied identities, and ends with built-in identities. Built-in identities have the advantage that the identities are preserved in case of modifications. According to this hierarchy, our domain elements have a built-in identity. One straightforward way to reflect this, is to model each domain element as an instance of an ADT. However, this direct approach does not properly capture the identities of the domain elements because the identity of an ADT instance is not built-in, but based on the query values. This section describes a way to introduce built-in identities for ADT instances.

To model domain elements with built-in identities, one can define an ADT with an identifier query. A number of ADT instances represent a single domain element over time. Each of the ADT instances has the same value for the identifier query. A modification of the domain element can then be modeled as a new ADT instance where the value of the identity query is preserved and all other queries modulo the modification are preserved.

For this to work, the formalization ensures that no two ADT instances that model different domain elements have the same identity. This is ensured with a fresh identifier for each ADT instance that models a new domain element. For this purpose, the universal stateful query *new_id* returns a fresh identifier. The formalization then preserves the identifier in every modification.

Typing environment Nienaltowski [25] presents a formalization of the SCOOP type system for a core of SCOOP called SCOOP_C. The type system formalization is part of the base for this work. The *typing environment* Γ contains the class hierarchy of a SCOOP program along with all the type definitions of all features and entities. Type rules allow us to derive conclusions.

The notation $\Gamma \vdash e : t$ denotes that expression e is of type t. Based on this derivation, the function $\mathit{type\_of}(\Gamma, e)$ denotes the type of expression e in the typing environment Γ. The type rules can be used to check whether an expression is controlled or not. In a SCOOP program, each processor p that wants to apply a feature f must make sure that all the processors $(q_1, \ldots, q_n)$ of all attached actual arguments of f are exclusively available on behalf of processor p. This guarantees exclusive access on all objects handled by processors $\{p, q_1, \ldots, q_n\}$. Note that processor p is in this set too because p can exclusively access its objects during a feature execution. For safety, the type system only allows feature calls in f on expressions, where the type system can derive that the value of the expression is a reference to an object and this object is handled by one of the processors $\{p, q_1, \dots, q_n\}$. Such an expression is called *controlled*. Whether or not an expression is controlled can be determined through the context in which the expression appears and the type of the expression. The context can either be the enclosing class, in case of expressions in invariants, or it can be the enclosing feature, in case of all other expressions. To be more precise, an expression e of type $t = (d, p, c)$ is controlled if and only if t is attached, i.e., $d = !$, and t satisfies at least one of the following conditions:

- The expression e is non-separate, i.e., $p = \bullet$.
- The expression e appears in a routine f that has an attached formal argument w with the same handler as e, i.e., $p = w.handler$.

The second condition is satisfied if and only if at least one of the following conditions is true:

- The expression e appears as an attached formal argument of f.
- The expression e has a qualified explicit processor specification *w.handler* and w is an attached formal argument of f.
- The expression e has an unqualified explicit processor specification p, and some attached formal argument of f has p as its unqualified explicit processor specification.

The notation $\Gamma \vdash controlled(t)$ denotes that an expression e of type t is controlled. To establish the derivation $\Gamma \vdash controlled(t)$ one has to find an attached formal argument w in the enclosing routine such that the types suggest that w and e are handled by the same processor or one has the establish that the type t is non-separate. One can therefore be sure that whenever an expression e is controlled, either a matching formal argument exists or its type is non-separate. For the first case, the formal argument is the *controlling entity* for e. For the second case, the current entity is the controlling entity. Although not present in Nienaltowski's formalization of the type system, this work introduces a new derivation $\Gamma \vdash y = controlling\_entity(e)$ that returns the controlling entity y for an expression e as an instance of **ENTITY**. This notion is essential to determine the handler of any controlled expression without evaluating the expression. One can simply determine the controlling entity and then determine the handler of the controlling entity.

4.2 Components of the State

The state is divided into three parts: the regions, the heap, and the store. The main purpose of the heap is to keep track of objects and to maintain the mapping of references to objects. It also maintains the once status of once routines, i.e., whether a once routine is fresh on a processor. The regions manage the association between objects and processors. Objects that are handled by the same processor form a region. The regions are also concerned with locking. The store is a map of names to references. It maps names of formal argument, names of local variables, the name of the current object entity, and the name of the result entity to references. A state ADT models the state with one query for each of the three parts.

$regions\colon \mathbf{STATE} \to \mathbf{REG}$

heap: **STATE** $\rightarrow$ **HEAP**

store: **STATE** $\rightarrow$ **STORE**

The next few sections introduce ADTs for each of the parts. A later section presents the state ADT.

4.3 Heap ADT

The *heap* keeps track of the objects and the references associated to them. It also keeps track of the status of once routines. This section first defines an ADT for objects and references. Then it introduces an ADT for the heap.

Objects and references There are two kinds of class types in the SCOOP type system: reference class types and expanded class types. The main difference lies in the semantics of using an instance of the types as the source of an attachment, such as assignment or argument passing. If an object of *reference class type* is the source of an attachment, then the reference to the object gets copied over to the destination of the attachment. The object is then accessible both through the source of the attachment as well as through the destination of the attachment. If an object of *expanded class type* is the source of an attachment, then a copy of the object gets attached to the destination of the attachment. The details can be found in Sec. 7.4 of the Eiffel ECMA standard [9].

This formalization takes a unified view on objects and references that is compatible with the semantics described in the Eiffel ECMA standard. It does not consider objects of expanded class type as sub-objects in other objects or in an environment. Instead it locates expanded objects on the heap, just like objects of reference class type. For each object there is exactly one reference. Assigning references to objects of expanded type has one major advantage for the formalization. If an ADT instance x that models an object gets updated, then one gets a new ADT instance y. If one would model expanded objects as sub-objects stored in other objects or in environments, then such an update might trigger a cascade of ADT instance updates: each ADT instance that has x as a query value would have to be updated with y, and so on. A consequent usage of references avoids this issue. To do the update, one simply alters the reference to x so that it points to y from now on.

The ADT **REF** models references with an identity query *id* and a constructor *make*. The constructor uses the query *new_id* to create a fresh identifier for the newly created reference. The void reference *void* is an instance of this ADT.

The ADT **OBJ** models objects. Each object has a query *id* for its identifier, a query *class_type* for its class type, and a query *att_val* for its attribute values. An object can only have attribute values for attributes that are defined in its class type.

The attribute values of an object can be modified with the command *set_att_val*. Only the attribute values for attributes that are defined in the class type can be modified. The result is an updated object where the attribute value of f is set to v. Note that the value can either be a reference or a processor. Processor values are necessary to support processor attributes.

$\mathit{set\_att\_val}\colon \textbf{OBJ} \to \textbf{FEATURE} \nrightarrow \textbf{REF} \cup \textbf{PROC} \to \textbf{OBJ}$
  $o.\mathit{set\_att\_val}(f,v)$ **require**
    $o.\mathit{class\_type}.\mathit{attributes}.\mathit{has}(f)$
  axioms
    $o.\mathit{set\_att\_val}(f,v).\mathit{att\_val}(f) = v$

The constructor *make* can be used to create a new object. It creates a new object with the given class type. The new object has a new identifier that is given by the query *new_id*. The constructor initializes all the attribute values of the new object with the void reference.

$\mathit{make}\colon \textbf{CLASS\_TYPE} \to \textbf{OBJ}$
  axioms
    $\mathit{make}(c).\mathit{id} = \mathit{new\_id}$
    $\mathit{make}(c).\mathit{class\_type} = c$
    $\forall i \in \{1,\ldots,n\}\colon \mathit{make}(c).\mathit{att\_val}(a_i) = \mathit{void}$
     where
     $\{a_1,\ldots,a_n\} \overset{def}{=} c.\mathit{attributes}$

An object can also be copied with the auxiliary query *copy*. This is important for expanded objects with copy semantics. The copied object has the same class type and the same attribute values as the original object, but it has a new identity. The new identity comes from the call to the constructor *make*.

$\mathit{copy}\colon \textbf{OBJ} \to \textbf{OBJ}$
  axioms
    $o.\mathit{copy} = \mathit{make}(o.\mathit{class\_type})$
      $.\mathit{set\_att\_val}(a_1, o.\mathit{att\_val}(a_1))$
      $....$
      $.\mathit{set\_att\_val}(a_n, o.\mathit{att\_val}(a_n))$
     where
     $n \overset{def}{=} o.\mathit{class\_type}.\mathit{attributes}.\mathit{count}$
     $\{a_1,\ldots,a_n\} \overset{def}{=} o.\mathit{class\_type}.\mathit{attributes}$

Mapping from references to objects The ADT **HEAP** makes use of **OBJ** and **REF** to model the mapping from references to objects. For this purpose, it declares the query *objs* to store all the objects on the heap and it declares the query *refs* to get all the references to these objects. The reference *void* is not part of the reference set. The query *ref_obj* defines the actual mapping. For each reference in *refs* an object in *objs* gets returned. The ADT also declares the query *last_added_obj* to keep track of the last object that has been added to the heap. It uses this query to define the effect of adding an object to the heap.

$\mathit{objs}\colon \textbf{HEAP} \to \textbf{SET}[\textbf{OBJ}]$

$\mathit{refs}\colon \textbf{HEAP} \to \textbf{SET}[\textbf{REF}]$

ref_obj: $\mathbf{HEAP} \to \mathbf{REF} \nrightarrow \mathbf{OBJ}$
  $h.ref\_obj(r)$ **require**
    $h.refs.has(r)$

$last\_added\_obj$: $\mathbf{HEAP} \to \mathbf{OBJ}$
  $h.last\_added\_obj$ **require**
    $\neg h.objs.is\_empty$

A number of commands are responsible for adding objects and for altering the mapping of references to objects. The command *add_obj* takes an object o and adds it to the heap. The result of the command is a new heap with the object o and a new reference that points to o. The newly added object is indicated in the query *last_added_obj*. Note that this command does not create a new object. It simply adds an object that has been provided as an argument. The command requires that the object is not yet part of the heap.

add_obj: $\mathbf{HEAP} \to \mathbf{OBJ} \nrightarrow \mathbf{HEAP}$
  $h.add\_obj(o)$ **require**
    $\forall u \in h.objs: u.id \neq o.id$
    $\forall a \in o.class\_type.attributes$:
      $o.att\_val(a) \in \mathbf{REF} \to (o.att\_val(a) = void \vee h.refs.has(o.att\_val(a)))$
  axioms
    $h.add\_obj(o).objs = h.objs \cup \{o\}$
    $h.add\_obj(o).refs = h.refs \cup \{r\}$
    $h.add\_obj(o).ref\_obj(r) = o$
    $h.add\_obj(o).last\_added\_obj = o$
   where
    $r \overset{def}{=} new\ \mathbf{REF}.make$

If an object that is already part of the heap gets updated, then it is necessary to update the mapping from the reference to the object on the heap. This can be done with the command *update_ref* that takes a reference r and an updated object o and returns a heap where the reference r points to o. The command requires that r is a valid reference and that o is an updated version of the original object. Because the remaining part of the state only deals with references rather than objects directly, a reference update does not require an update of these parts.

$update\_ref$: $\mathbf{HEAP} \to \mathbf{REF} \nrightarrow \mathbf{OBJ} \nrightarrow \mathbf{HEAP}$
  $h.update\_ref(r,o)$ **require**
    $h.refs.has(r)$
    $o.id = h.ref\_obj(r).id$
    $\forall a \in o.class\_type.attributes$:
      $o.att\_val(a) \in \mathbf{REF} \to (o.att\_val(a) = void \vee h.refs.has(o.att\_val(a)))$
  axioms
    $h.update\_ref(r,o).objs.has(o)$
    $o \neq h.ref\_obj(r) \to \neg h.update\_ref(r,o).objs.has(h.ref\_obj(r))$
    $h.update\_ref(r,o).ref\_obj(r) = o$
    $h.last\_added\_obj = h.ref\_obj(r) \to h.update\_ref(r,o).last\_added\_obj = o$

So far **HEAP** covers the mapping from references to objects. Occasionally it is necessary to have the inverse mapping. The commands *add_obj* and *update_ref* ensure that there is exactly one reference for each object on the heap. Thus it is possible to define the inverse query *ref* as an auxiliary query.

```
$ref\colon \mathbf{HEAP} \to \mathbf{OBJ} \nrightarrow \mathbf{REF}$
  $h.ref(o)$ **require**
    $h.objs.has(o)$
  **axioms**
    $h.ref\_obj(h.ref(o)) = o$
```

Once routines A *once routine* can either be a once function or a once procedure. A once routine gets executed at most once in a certain context. If a once routine has been executed in the context, then it is called *non-fresh* in the context. Otherwise it is called *fresh* in the context. The context is either the set of all processors in the system or a single processor. The heap remembers which once routines are fresh. For this purpose, **HEAP** declares the queries *is_fresh* and *once_result*. For any processor p and any once routine f, the query *is_fresh* states whether f is fresh on p or not. For a once function f that is not fresh on a processor p, the query *once_result* returns the result of f on p.

Two commands change the once status of a fresh once routine to non-fresh. One version works for once functions and the other one for once procedures. Both commands take the once routine f and the processor p. The version for once functions also takes a once result r. The two commands implement the semantics for once routines: a once routine has either a once per system or a once per processor semantics. Once functions declared as separate with or without an explicit processor specification have the once per system semantics. In this case, the command *set_once_func_not_fresh* defines f as non-fresh on all processors. Once functions with a non-separate result type have the once per processor semantics. In this case, the command *set_once_func_not_fresh* sets f as non-fresh on p with the once result r. Once procedures have the once per processor semantics. In this case, the command *set_once_proc_not_fresh* sets f as non-fresh on p.

```
$set\_once\_func\_not\_fresh\colon \mathbf{HEAP} \to \mathbf{PROC} \to \mathbf{FEATURE} \nrightarrow \mathbf{REF} \nrightarrow \mathbf{HEAP}$
  $h.set\_once\_func\_not\_fresh(p,f,r)$ **require**
    $f \in \mathbf{FUNCTION} \wedge f.is\_once$
    $r \neq void \to h.refs.has(r)$
  **axioms**
    $(\exists d,c\colon \Gamma \vdash f : (d,\bullet,c)) \to$
      $\neg h.set\_once\_func\_not\_fresh(p,f,r).is\_fresh(p,f) \wedge$
      $h.set\_once\_func\_not\_fresh(p,f,r).once\_result(p,f) = r$
    $(\exists d,c\colon \Gamma \vdash f : (d,p,c) \wedge p \neq \bullet) \to \forall q \in \mathbf{PROC}\colon$
      $\neg h.set\_once\_func\_not\_fresh(p,f,r).is\_fresh(q,f) \wedge$
      $h.set\_once\_func\_not\_fresh(p,f,r).once\_result(q,f) = r$

$set\_once\_proc\_not\_fresh\colon \mathbf{HEAP} \to \mathbf{PROC} \to \mathbf{FEATURE} \nrightarrow \mathbf{HEAP}$
  $h.set\_once\_proc\_not\_fresh(p,f)$ **require**
    $f \in \mathbf{PROCEDURE} \wedge f.is\_once$
  **axioms**
    $\neg h.set\_once\_proc\_not\_fresh(p,f).is\_fresh(p,f)$
```

Creation A new heap can be created with the constructor *make*. A new heap has no objects and no references. All once routines are marked as fresh on all processors.

make: **HEAP**
  axioms
    $make.objs.is\_empty$
    $make.refs.is\_empty$
    $\forall p \in \mathbf{PROC}, f \in \mathbf{FEATURE}\colon\ f.is\_once \rightarrow make.is\_fresh(p, f)$

4.4 Regions ADT

The heap is partitioned into disjoint *regions*, and each region is assigned to exactly one processor. This concept relates to the concept of a ken in Schmidt's work [30]. The processor of a region is the handler of all the objects in the region. Regions are also used to maintain locks. The following discussion first describes an ADT for processor and then describes an ADT for regions.

Processors A *processor* is an autonomous thread of control capable of executing features on objects. Each processor is responsible for a set of objects. As such a processor is called the *handler* of its associated objects. Each object is assigned to exactly one processor that is the authority of feature executions on this object. If a processor q wants to call a feature on an object handled by a different processor p, then q needs to send a feature request to processor p. This is where the request queue of processor p comes into place. The *request queue* keeps track of features to be executed on behalf of other processors. Processor q can add a request to this queue and processor p will execute the request as soon as it executed all previous requests in the request queue. Processor p uses its *call stack* to execute the feature request at the beginning of the request queue. The call stack is responsible for the order of feature executions on the same processor. In a situation of a non-separate call, the call stack ensures that the calling feature execution resumes once the called feature execution terminated. The interaction between the call stack and the request queue is best described with the following loop through which each processor goes:

1. Idle wait. If both the call stack and the request queue are empty, then wait for new requests to be enqueued.
2. Request scheduling. If the call stack is empty but the request queue is not empty, then dequeue an item and push it onto the call stack.
3. Request processing. If there is an item on the call stack, then pop the item from the call stack and process it. If the item is a feature request, then apply the feature. If the item is an operation, then execute the operation.

For each processor there is a *request queue lock* and a *call stack lock*. A lock on the request queue grants permission to add a feature request to the end of the request queue. A lock on the call stack grants permission to add a feature request to the top of the call stack. Before processor q can add a request to p's request queue, it must have a lock on this request queue. Otherwise another processor could intervene. Once processor q

is done with the request queue of processor *p* it can add an unlock operation to the end of the request queue. This makes sure that the request queue lock of *p* will be released after all the previous feature requests have been executed. Similarly, processor *p* must have a lock on its call stack to add features to its call stack. Initially, each processor has a lock on its own call stack and its request queue is not locked.

Processor *q* could also make a synchronous call to *p*. However *q* might be in possession of some locks that are necessary for the execution of the resulting feature request on *p*. In such a situation, *q* is waiting for the synchronous call to terminate and *p* is waiting for locks to be available. According to the conditions given by Coffman et al. [8] a deadlock occurred. This can be avoided if *q* temporarily passes its locks to the *p*. This allows *p* to finish the execution and hence *q* can continue.

Clarification 1 (Request queue locks and call stack locks). The notion of request queue locks and call stack locks was not present in Nienaltowski's [25] definition of SCOOP. He defines one lock for each processor. A lock on a processor means exclusive access to the whole processor. This lock model is not sufficient to describe SCOOP. In particular, this lock model creates a contradiction with respect to separate callbacks. A separate callback is a feature call in which processor *q* made a direct or indirect call to processor *p* and now *p* is calling back processor *q*. The separate callback is only possible if *p* has a lock on *q*. However, *p* does not necessarily have this lock because the lock might be in possession of the processor that locked *q* in the first place. Request queue locks and call stack locks allow us to clarify the situation. Thus we propose a new lock model with request queue locks and call stack locks.

The lock model used in Nienaltowski's work [25] is an abstraction of the new lock model. The abstraction works under the assumption that no processor passes its locks. Under this assumption each processor keeps its call stack lock. In this abstraction, the request queue lock on a processor *p* is called the lock on *p*. As long as the call stack lock on a processor *p* is in possession of *p*, a request queue lock on *p* in possession of a processor *q* means that processor *p* will be executing new feature requests in the request queue exclusively on behalf of *q*. This means that a request queue lock grants exclusive access to all the objects handled by *p*. Transferring this insight to the abstraction, a lock on processor *p* denotes exclusive access to the objects handled by *p*. □

The formalization defines the ADT **PROC** for processors. A processor has an identifier stored in the query *id*.

The constructor *make* returns a new processor with a fresh identifier. The fresh identifier is defined through the query *new_id*.

make: **PROC**
  axioms
    $make.id = new\_id$

The ADT **PROC** is very simple. It neither takes care of the mapping from processors to the handled objects nor does it take care of the locks. These aspects are taken care of by the ADT for regions.

Mapping of processors to objects and locking This section introduces the ADT **REG** for regions. This ADT declares a query *procs* that keeps track of all the processors in the system. The query *handled_objs* defines a set of handled objects for each processor in *procs*. Finally, the query *last_added_proc* denotes the last processor that has been added to *procs*.

$procs$: $\mathbf{REG} \rightarrow \mathbf{SET}[\mathbf{PROC}]$

$handled\_objs$: $\mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{OBJ}]$
  $k.handled\_objs(p)$ **require**
    $k.procs.has(p)$

$last\_added\_proc$: $\mathbf{REG} \nrightarrow \mathbf{PROC}$
  $k.last\_added\_proc$ **require**
    $\neg k.procs.is\_empty$

Next to the queries that are concerned with the mapping from processors to objects, there are a number of queries that deal with locking. The feature *rq_locked* states whether the request queue of a processor in *procs* is locked or not. Similarly, the feature *cs_locked* states whether the call stack is locked.

The remaining queries specify the owners of the locks. For this, the formalization distinguishes between *obtained* and *retrieved* locks. Obtained locks are locks that got acquired by a processor. Retrieved locks are locks that got passed from another processor.

The query *obtained_rq_locks* returns a stack of obtained processor sets for a processor. A stack of sets models the way processors acquire locks: they go through a nested series of feature applications and each feature application requires a set of locks before the feature can be executed. For each feature application the executing processor adds a new set on top of its stack. As soon as the feature application finished, the processor removes the top set from its stack. The query *obtained_cs_lock* returns the acquired call stack lock of a processor. Initially each processor starts with a lock on its own call stack and this call stack lock never changes. Thus this query is only declared for reasons of completeness. If a processor appears in a set of request queue locks, then the processor denotes its request queue lock. If a processor appears in a set of call stack locks, then the processor denotes its call stack lock.

A processor can pass its locks to another processor. There are several queries to formalize this aspect. The features *retrieved_rq_locks* and *retrieved_cs_locks* return the retrieved locks of a processor. Both of these queries return a stack of sets. The stack keeps track of the set of retrieved locks for each feature application. These two stacks grow and shrink in parallel to the stack *obtained_rq_locks*. Once a processor passed its locks, it cannot use them anymore until the locks are revoked. The query *locks_passed* returns whether a processor passed some or all of its locks or not.

The following discussion first goes through the list of commands that add processors and commands that change the association of processors to objects. It then proceeds with the commands that handle locks. The command *add_proc* updates the regions with a new processor. Note that the processor must have been created beforehand. The axioms state that the new processor will be included in *procs* and that it will be stored in *last_added_proc*. The axioms also state how the new processor is initialized. The new processor's request queue is unlocked and its call stack is locked. Apart from the initial lock on the call stack there are no obtained or retrieved locks and hence the processor did not pass its locks.

$add\_proc\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{REG}$
  $k.add\_proc(p)$ **require**
    $\neg k.procs.has(p)$
  axioms
    $k.add\_proc(p).procs.has(p)$
    $k.add\_proc(p).last\_added\_proc = p$
    $k.add\_proc(p).handled\_objs(p).is\_empty$
    $\neg k.add\_proc(p).rq\_locked(p)$
    $k.add\_proc(p).cs\_locked(p)$
    $k.add\_proc(p).obtained\_rq\_locks(p).is\_empty$
    $k.add\_proc(p).obtained\_cs\_lock(p) = p$
    $k.add\_proc(p).retrieved\_rq\_locks(p).is\_empty$
    $k.add\_proc(p).retrieved\_cs\_locks(p).is\_empty$
    $\neg k.add\_proc(p).locks\_passed(p)$

The command *add_obj* takes a processor p in *procs* and an object o that is not handled by a processor in *procs* yet. It returns the updated regions in which o is handled by p.

$add\_obj\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{OBJ} \nrightarrow \mathbf{REG}$
  $k.add\_obj(p,o)$ **require**
    $k.procs.has(p)$
    $\forall q \in k.procs, u \in k.handled\_objs(q)\colon u.id \neq o.id$
  axioms
    $k.add\_obj(p,o).handled\_objs(p).has(o)$

In the opposite direction, the command *remove_obj* removes an object that is handled by a processor in *procs* from the regions.

$remove\_obj\colon \mathbf{REG} \to \mathbf{OBJ} \nrightarrow \mathbf{REG}$
  $k.remove\_obj(o)$ **require**
    $\exists p \in k.procs\colon k.handled\_objs(p).has(o)$
  axioms
    $\neg\exists p \in k.procs\colon k.remove\_obj(o).handled\_objs(p).has(o)$

The following part discusses the commands that deal with the locking aspects of the regions. The command *lock_rqs* locks the request queues of a set of processors $\overline{q}$ on behalf of a processor p. None of these request queues must be locked beforehand.

$lock\_rqs\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}] \nrightarrow \mathbf{REG}$
  $k.lock\_rqs(p,\overline{l})$ **require**
    $k.procs.has(p)$
    $\forall x \in \overline{l}\colon k.procs.has(x)$
    $\forall x \in \overline{l}\colon \neg k.rq\_locked(x)$
  axioms
    $k.lock\_rqs(p,\overline{l}).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).push(\overline{l})$
    $\forall x \in \overline{l}\colon k.lock\_rqs(p,\overline{l}).rq\_locked(x)$

At some point, processor p will not require the obtained request queue locks anymore because p made sure to enqueue all necessary features requests. Processor p uses the command *pop_obtained_rq_locks* to remove his claims on the obtained request queue locks. This requires that processor p is in possession of these locks, i.e., that p did not pass its locks.

```
$pop\_obtained\_rq\_locks\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{REG}$
  $k.pop\_obtained\_rq\_locks(p)$ **require**
    $k.procs.has(p)$
    $\neg k.obtained\_rq\_locks(p).is\_empty$
    $\neg k.locks\_passed(p)$
  **axioms**
    $k.pop\_obtained\_rq\_locks(p).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).pop$
```

Removing the locks from p's obtained request queue locks stack does not mean that these request queues are unlocked. It just means that the request queue locks are not claimed by p anymore and therefore p will not enqueue further feature requests on the respective processors. The request queues remain locked until they get unlocked with a call to the command *unlock_rq*. This happens after the processors whose request queues got locked by p finished all the requested feature applications. The precondition of the command states that a request queue can only be unlocked if it is not claimed by any other processor. This precondition guarantees that the request queue can only be unlocked when it is not used as an obtained or retrieved lock by any other processor anymore. Note that there is no unlock command for call stack locks because the call stack never gets unlocked.

```
$unlock\_rq\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{REG}$
  $k.unlock\_rq(p)$ **require**
    $k.procs.has(p)$
    $k.rq\_locked(p)$
    $\forall q \in k\colon \neg k.obtained\_rq\_locks(q).flat.has(p)$
  **axioms**
    $\neg k.unlock\_rq(p).rq\_locked(p)$
```

The request queues remain locked until explicitly unlocked with a call to *unlock_rq*. Between the call to *pop_obtained_rq_locks* and the call to *unlock_rq*, the owner of these locks is undefined. In some situations this is not satisfactory. A different solution must be found if another processor wants to claim the locks until they are unlocked. The command *delegate_obtained_rq_locks* serves this purpose. It takes a processor p and a number of processors $\bar{l}$ and makes p the owner of the request queue locks of all processors in $\bar{l}$ by adding these locks to the obtained request queue locks stack of p. This can only work if there is no current owner and the request queues are indeed locked.

$delegate\_obtained\_rq\_locks\colon \mathbf{REG} \to \mathbf{PROC} \not\to \mathbf{SET}[\mathbf{PROC}] \not\to \mathbf{REG}$

$\quad k.delegate\_obtained\_rq\_locks(p, \overline{l})$ **require**

$\quad\quad k.procs.has(p)$

$\quad\quad \forall x \in \overline{l}\colon k.procs.has(x)$

$\quad\quad \forall x \in \overline{l}\colon \neg\exists y \in k.procs\colon k.obtained\_rq\_locks(y).flat.has(x)$

$\quad\quad \forall x \in \overline{l}\colon k.rq\_locked(x)$

$\quad$ **axioms**

$\quad\quad k.delegate\_obtained\_rq\_locks(p, \overline{l}).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).push(\overline{l})$

Delegation is different from lock passing: *delegation* is the permanent transfer of ownership and *lock passing* is the temporary transfer of the right to use the locks. The following discussion looks at the commands to pass and revoke locks. The command *pass_locks* takes a processor p and a processor q as well as a set of request queue locks $\overline{l_r}$ along with a set of call stack locks $\overline{l_c}$. The result is an updated instance of **REG** in which $\overline{l_r}$ and $\overline{l_c}$ have been passed from p to q. As a precondition for this task, processor p must be in possession of all these locks. This means that all the locks in $\overline{l_r}$ and $\overline{l_c}$ must be obtained or retrieved locks of p and the locks must not be passed. The updated result must reflect that some or all of p's locks have been passed. However, because the two sets of locks can potentially be empty, p's locks must only be marked as passed if at least one of the two sets of locks is non-empty. Lastly, the command must take care of one special case of the lock passing operation. If a processor q different from processor p passed its locks in a previous lock passing operation and now the command passes these locks back to q, then the command has to mark the locks of processor q as not passed. This case is important to handle separate callbacks.

$pass\_locks\colon \mathbf{REG} \to \mathbf{PROC} \not\to \mathbf{PROC} \not\to \mathbf{TUPLE}[\mathbf{SET}[\mathbf{PROC}], \mathbf{SET}[\mathbf{PROC}]] \not\to \mathbf{REG}$

$\quad k.pass\_locks(p, q, (\overline{l_r}, \overline{l_c}))$ **require**

$\quad\quad k.procs.has(p) \wedge k.procs.has(q)$

$\quad\quad \forall x \in \overline{l_r}\colon k.procs.has(x) \wedge \forall x \in \overline{l_c}\colon k.procs.has(x)$

$\quad\quad \forall x \in \overline{l_r}\colon k.obtained\_rq\_locks(p).flat.has(x) \vee k.retrieved\_rq\_locks(p).flat.has(x)$

$\quad\quad \forall x \in \overline{l_c}\colon x = k.obtained\_cs\_lock(p) \vee k.retrieved\_cs\_locks(p).flat.has(x)$

$\quad\quad \neg k.locks\_passed(p)$

$\quad$ **axioms**

$$k.pass\_locks(p, q, (\overline{l_r}, \overline{l_c})).locks\_passed(p) = \begin{cases} true & \text{if} \neg\overline{l_r}.is\_empty \vee \neg\overline{l_c}.is\_empty \\ false & otherwise \end{cases}$$

$$k.pass\_locks(p, q, (\overline{l_r}, \overline{l_c})).retrieved\_rq\_locks(q) = k.retrieved\_rq\_locks(q).push(\overline{l_r})$$

$$k.pass\_locks(p, q, (\overline{l_r}, \overline{l_c})).retrieved\_cs\_locks(q) = k.retrieved\_cs\_locks(q).push(\overline{l_c})$$

$$\begin{pmatrix} p \neq q \wedge \\ k.locks\_passed(q) \wedge \\ k.obtained\_rq\_locks(q).flat \subseteq \overline{l_r} \wedge \\ k.retrieved\_rq\_locks(q).flat \subseteq \overline{l_r} \wedge \\ k.obtained\_cs\_lock(q) \in \overline{l_c} \wedge \\ k.retrieved\_cs\_locks(q).flat \subseteq \overline{l_c} \end{pmatrix} \to \neg k.pass\_locks(p, q, (\overline{l_r}, \overline{l_c})).locks\_passed(q)$$

The command *revoke_locks* takes a processor p and a processor q. It reverses the effect of a lock passing operation from a processor p to q and returns an updated instance of **REG**. This is only allowed if processor p passed locks to q in a preceding lock passing operation. Note that the lock passing operation from p to q potentially marked the locks of q as not passed. Revoking the locks from q to p requires the reverse action. If p has retrieved locks in common with the locks of q, even after the retrieved locks from p have been removed from q, then q's locks must be marked as passed because they are now in possession of p.

```
$revoke\_locks\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{PROC} \nrightarrow \mathbf{REG}$
  $k.revoke\_locks(p,q)$ require
    $k.procs.has(p) \land k.procs.has(q)$
    $\neg k.retrieved\_rq\_locks(q).is\_empty \land \neg k.retrieved\_cs\_locks(q).is\_empty$
    $k.retrieved\_rq\_locks(q).top \subseteq k.obtained\_rq\_locks(p).flat \cup k.retrieved\_rq\_locks(p).flat$
    $k.retrieved\_cs\_locks(q).top \subseteq \{k.obtained\_cs\_lock(p)\} \cup k.retrieved\_cs\_locks(p).flat$
    $k.retrieved\_rq\_locks(q).top \cup k.retrieved\_cs\_locks(q).top \neq \{\} \to k.locks\_passed(p)$
    $\neg k.locks\_passed(q)$
  axioms
    $\neg k.revoke\_locks(p,q).locks\_passed(p)$
    $k.revoke\_locks(p,q).retrieved\_rq\_locks(q) = k.retrieved\_rq\_locks(q).pop$
    $k.revoke\_locks(p,q).retrieved\_cs\_locks(q) = k.retrieved\_cs\_locks(q).pop$
    $\left(\begin{array}{l} p \neq q \land \\ \left(\begin{array}{l} \exists x \in k.retrieved\_rq\_locks(p).flat\colon ( \\ \quad k.obtained\_rq\_locks(q).flat.has(x) \lor \\ \quad k.retrieved\_rq\_locks(q).pop.flat.has(x) \\ ) \lor \\ \exists x \in k.retrieved\_cs\_locks(p).flat\colon ( \\ \quad x = k.obtained\_cs\_lock(q) \lor \\ \quad k.retrieved\_cs\_locks(q).pop.flat.has(x) \\ ) \end{array}\right) \end{array}\right)$
      $\to k.revoke\_locks(p,q).locks\_passed(q)$
```

These commands wrap up the mapping of processors to objects and the locking aspects. The discussion continues with a number of auxiliary queries to simplify access to the presented queries. The command *add_obj* makes sure that a processor is assigned to each object that gets added. This mapping is available through the query *handled_objs*. Thus it is possible to define an auxiliary query *handler* that is inverse to the query *handled_objs*.

```
$handler\colon \mathbf{REG} \to \mathbf{OBJ} \nrightarrow \mathbf{PROC}$
  $k.handler(o)$ require
    $\exists p \in k.procs\colon k.handled\_objs(p).has(o)$
  axioms
    $k.handled\_objs(k.handler(o)).has(o)$
```

There are four different categories of locks that each processor can have. For both the request queue locks and the call stack locks, there are queries for obtained and retrieved locks. In some situations it is easier to just work with request queue locks and call stack locks without splitting them into obtained and retrieved locks. The auxiliary queries *rq_locks* and *cs_locks* serve this purpose. The auxiliary query *rq_locks* returns a set that contains all the obtained and the retrieved request queue locks of a processor p. Similarly, the auxiliary query *cs_locks* returns all the call stack locks of a processor p.

$rq\_locks\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}]$
  $k.rq\_locks(p)$ **require**
    $k.procs.has(p)$
  axioms
    $k.rq\_locks(p) = k.obtained\_rq\_locks(p).flat \cup k.retrieved\_rq\_locks(p).flat$

$cs\_locks\colon \mathbf{REG} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}]$
  $k.cs\_locks(p)$ **require**
    $k.procs.has(p)$
  axioms
    $k.cs\_locks(p) = \{k.obtained\_cs\_lock(p)\} \cup k.retrieved\_cs\_locks(p).flat$

Creation The constructor *make* creates a new instance of **REG**. The new instance has no processors.

$make\colon \mathbf{REG}$
  axioms
    $make.procs.is\_empty$

4.5 Store ADT

Each processor in the system has a call stack to execute features. Every time a processor executes a feature, a new call stack frame gets created on top of the call stack. The new call stack frame stores the values of formal arguments, local variables, the current object entity, and the result entity for the current feature execution. The call stack is also responsible for the order of feature executions on the same processor. This formalization separates the two concerns of the call stack. The *store* only models the values in each stack frame. A store has a stack of environments for each processor, where each *environment* maps names to values. This section first presents an ADT for environments and then presents an ADT for the store.

Environments The ADT **ENV** has a query *names* that stores all the defined names. The query *val* can then be used to get the value for each such name.

$names\colon \mathbf{ENV} \to \mathbf{SET}[\mathbf{NAME}]$

$val\colon \mathbf{ENV} \to \mathbf{NAME} \nrightarrow \mathbf{REF} \cup \mathbf{PROC}$
  $e.val(n)$ **require**
    $e.names.has(n)$

The command *update* takes a name and a value and returns an updated environment. Note that it does not matter whether the name is already defined in the environment or not. In any case, the name will be defined in the updated environment and the name will be mapped to the value. The value can either be a reference or a processor. Environments with processor values are not strictly needed to describe SCOOP, however they make it possible to have a unified view on attribute values and environment values.

$update\colon$ **ENV** $\rightarrow$ **NAME** $\rightarrow$ **REF** $\cup$ **PROC** $\rightarrow$ **ENV**
  axioms
    $e.update(n,v).names = e.names \cup \{n\}$
    $e.update(n,v).val(n) = v$

The constructor *make* returns an empty environment.

$make\colon$ **ENV**
  axioms
    $make.names.is\_empty$

Mapping from processors to environments The ADT **STORE** has a single query *envs* that stores a stack of environments for each processor.

$envs\colon$ **STORE** $\rightarrow$ **PROC** $\rightarrow$ **STACK**[**ENV**]

The command *push_env* pushes a given environment on top a processor's stack of environments. The command *pop_env* pops the top environment from a non-empty stack of environments.

$push\_env\colon$ **STORE** $\rightarrow$ **PROC** $\rightarrow$ **ENV** $\rightarrow$ **STORE**
  axioms
    $s.push\_env(p,e).envs(p) = s.envs(p).push(e)$

$pop\_env\colon$ **STORE** $\rightarrow$ **PROC** $\nrightarrow$ **STORE**
  $s.pop\_env(p)$ **require**
    $\neg s.envs(p).is\_empty$
  axioms
    $s.pop\_env(p).envs(p) = s.envs(p).pop$

The constructor *make* creates an empty store.

$make\colon$ **STORE**
  axioms
    $\forall p \in$ **PROC**$\colon make.envs(p).is\_empty$

4.6 State ADT

The ADT **STATE** models the *state* with three queries to retrieve the different parts of the ADT.

$regions\colon$ **STATE** $\rightarrow$ **REG**

$heap\colon$ **STATE** $\rightarrow$ **HEAP**

$store\colon$ **STATE** $\rightarrow$ **STORE**

The command *set* sets the regions, the heap, and the store at the same time. A precondition specifies consistency criteria between the parts of the state. The first precondition clause states that a processor can handle an object if and only if the object is on the heap. The second precondition clause states that if the heap declares a feature as non-fresh on a processor p, then the regions must know about this processor. The third precondition clause requires that all processors stored in attribute values are known by the regions. Note that **HEAP** already requires that the references stored in attribute values are known. The forth precondition clause states that each non-empty environment in the store must belong to a processor that is known by the regions. The fifth precondition clause states that each value in the store must either be a known reference or a known processor.

```
set: STATE → REG ⇸ HEAP ⇸ STORE ⇸ STATE
  σ.set(k,h,s) require
    $\exists p \in k.procs, \exists o \in \mathbf{OBJ}\colon k.handled\_objs(p).has(o) \leftrightarrow h.objs.has(o)$
    $\exists p \in \mathbf{PROC}, f \in \mathbf{FEATURE}\colon \neg h.is\_fresh(p,f) \rightarrow k.procs.has(p)$
    $\forall o \in h.objs, a \in o.class\_type.attributes\colon o.att\_val(a) \in \mathbf{PROC} \rightarrow k.procs.has(o.att\_val(a))$
    $\forall p \in \mathbf{PROC}, e \in s.envs(p)\colon \neg e.names.is\_empty \rightarrow k.procs.has(p)$
    $\forall p \in k.procs, e \in s.envs(p), x \in e.names\colon$
      $(e.val(x) \in \mathbf{REF} \rightarrow e.val(x) = void \vee h.refs.has(e.val(x)))\wedge$
      $(e.val(x) \in \mathbf{PROC} \rightarrow k.procs.has(e.val(x)))$
  axioms
    $\sigma.set(k,h,s).regions = k$
    $\sigma.set(k,h,s).heap = h$
    $\sigma.set(k,h,s).store = s$
```

Creation To create a state, one has to create the three parts of the state. This is done with the constructor *make*.

```
make: STATE
  axioms
    $make.regions = new\ \mathbf{REG}.make$
    $make.heap = new\ \mathbf{HEAP}.make$
    $make.store = new\ \mathbf{STORE}.make$
```

Facade It is too cumbersome to work with **STATE** as it is. For example, the following expression defines a new state σ' in which a new processor has been added to the state σ: $\sigma' \overset{def}{=} \sigma.set(\sigma.regions.add\_proc(new\ \mathbf{PROC}.make), \sigma.heap, \sigma.store)$. This expression is too long for this simple task, especially if the expression is used multiple times. It would be easier to have an auxiliary command that does this job for us. The *facade* is an abstraction with auxiliary features that provide easy access to the state functionality. The facade is divided into different aspects. The following discussion dedicates one section to each aspect. It starts with the mapping of processors to objects and the mapping of references to objects. It continues with a section on how to set values, followed by a section on how to get values. It concludes with a section on locking.

Mapping of processors to objects and mapping of references to objects The regions and the heap manage the references, the objects, the processors, and the mapping between them. The facade unifies all related features in one aspect. This section first defines a number of auxiliary queries for the mapping of processors to objects. Next, it defines auxiliary queries for the mapping of references to objects. It then defines auxiliary commands that work on both aspects.

The two auxiliary queries *procs* and *last_added_proc* give access to all the processors and the last added processor.

The auxiliary query *handler* gives the handler of an object referenced by *r*. The auxiliary query uses the heap to get the referenced object and then gives this object to the regions to get the handler. In contrast to the corresponding auxiliary query in **REG**, the version here takes a reference instead of an object. The version in **REG** deals directly with objects rather than references because it does not know about the heap and thus the mapping from references to objects is not available. The facade, however, has access to both the regions and the heap and thus it can use the preferred way of identifying objects: references.

The auxiliary query *new_proc* is a shorthand for processor creation. The auxiliary query *last_added_obj* returns the object that has been added last to the heap. The auxiliary query *ref_obj* returns the object that is associated to a given reference. In the other direction, the auxiliary query *ref* returns the reference to a given object. The auxiliary query *new_obj* is a shorthand for object creation; it returns a new object with a given class type.

The discussion continues with the auxiliary commands that modify the mapping of processors to objects and the mapping of references to objects. Before an object can be added to the set of handled objects of a processor, the processor must exist. If the processor does not exist yet, the command *add_proc* can be used to update a state with a new processor.

$add\_proc\colon$ **STATE** $\rightarrow$ **PROC** $\nrightarrow$ **STATE**
  $\sigma.add\_proc(p)$ **require**
    $\neg\sigma.regions.procs.has(p)$
  axioms
    $\sigma.add\_proc(p) = \sigma.set(\sigma.regions.add\_proc(p), \sigma.heap, \sigma.store)$

The auxiliary command *add_obj* can then be used to add an object to the processor and the heap. The auxiliary command takes a processor *p* and an object *o* and it returns a state in which object *o* is part of the heap and handled by processor *p*.

$add\_obj\colon$ **STATE** $\rightarrow$ **PROC** $\nrightarrow$ **OBJ** $\nrightarrow$ **STATE**
  $\sigma.add\_obj(p,o)$ **require**
    $\sigma.regions.procs.has(p)$
    $\forall u \in \sigma.heap.objs\colon u.id \neq o.id$
    $\forall a \in o.class\_type.attributes\colon$
      $(o.att\_val(a) \in \mathbf{REF} \rightarrow o.att\_val(a) = void \vee \sigma.heap.refs.has(o.att\_val(a)))\wedge$
      $(o.att\_val(a) \in \mathbf{PROC} \rightarrow \sigma.regions.procs.has(o.att\_val(a)))$
  axioms
    $\sigma.add\_obj(p,o) = \sigma.set(\sigma.regions.add\_obj(p,o), \sigma.heap.add\_obj(o), \sigma.store)$

The auxiliary command *update_ref* updates a reference with an updated object. It takes a reference r on the heap and an object o and it returns a state in which o replaced the object u referenced by r on the heap and in the regions. Note that o must indeed be an updated version of the object referenced by r. The auxiliary command first removes u from the set of handled objects and then adds o to the set of handled objects of u's handler. Then it updates the heap with the command *update_ref*, which is declared in **HEAP**.

```
$update\_ref : \mathbf{STATE} \to \mathbf{REF} \nrightarrow \mathbf{OBJ} \nrightarrow \mathbf{STATE}$
  $\sigma.update\_ref(r,o)$ require
    $\sigma.heap.refs.has(r)$
    $o.id = \sigma.heap.ref\_obj(r).id$
    $\forall a \in o.class\_type.attributes:$
      $(o.att\_val(a) \in \mathbf{REF} \to o.att\_val(a) = void \lor \sigma.heap.refs.has(o.att\_val(a)))\land$
      $(o.att\_val(a) \in \mathbf{PROC} \to \sigma.regions.procs.has(o.att\_val(a)))$
  axioms
    $\sigma.update\_ref(r,o) = \sigma.set(k,h,s)$
     where
     $u \overset{def}{=} \sigma.heap.ref\_obj(r)$
     $k \overset{def}{=} \sigma.regions.remove\_obj(u).add\_obj(\sigma.regions.handler(u),o)$
     $h \overset{def}{=} \sigma.heap.update\_ref(r,o)$
     $s \overset{def}{=} \sigma.store$
```

Setting values This section takes a look at how to set values. To start, it looks at a prerequisite for this task: the deep import operation. Setting values includes setting values of formal arguments, values of local variables, the value of the current object entity, the value of the result entity, and attribute values of the current object. All of these values can be written and read without a feature call. This section concludes with auxiliary commands to set the status of once routines. The SCOOP validity rules exclude other types of value setting operations.

Deep Import Operation Expanded objects have a copy semantics: if an object o of expanded class type is the source of an attachment, then a copy u gets attached to the destination of the attachment. However, a shallow copy is not sufficient if o's handler p is different from u's handler q. If o has an attached non-separate entity, then u now has a non-separate entity to which a separate object is attached. This would result in a *traitor* – a non-separate entity that points to a separate object. The SCOOP model, as defined by Nienaltowski [25], introduces the *import operation* to solve this issue. Applied to o the import operation creates a copied object structure that mirrors the original object structure in a way that o and all the objects reachable from o through non-separate references are replaced with copied objects that are handled by q. This data structure then gets attached to the destination of the attachment. The import operation computes the non-separate version of an object structure.

Clarification 2 (Deep import operation). The import operation potentially results in a copied object structure that contains both copied and original objects. This can be an issue in case one of the copied objects has an invariant over the identities of objects, as shown in example 3.

Example 3 (Invariant violation as a result of the import operation). Imagine two ob-

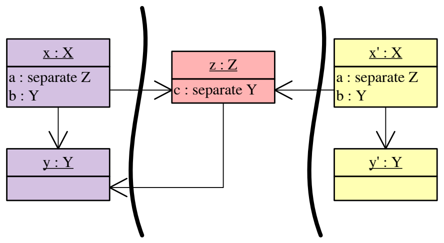

Fig. 2. Invariant violation as a result of the import operation

jects x and y handled by one processor and another object z handled by another processor. Object x has a separate entity a that points to z and a non-separate entity b that points to y. Object z has a separate entity c that points to y. Object x has an invariant with a query $a.c = b$. An import operation on x executed by a third processor will result in two new objects x' and y' on the third processor. The reference a of object x' will point to the original z. The reference b of object x' will point to the new object y'. This situation is illustrated in Fig. 2. Now object x' is inconsistent, because $a.c$ and b identify different objects, namely y and y'.

The *deep import operation* is a variant of the import operation that does not mix the copied and the original objects. □

Instead of copying only the objects that are reachable through non-separate references, the deep import operation makes a full copy of the object structure. The deep importing processor handles all the copies of the objects that are non-separate with respect to the object to be imported. Each other separate object is handled by the processor of the respective original object. The deep import operation does not show the issue with invariants. The drawback of the deep import operation is that more objects must be copied. Nevertheless, we use the deep import operation in our formalization because we cannot tolerate violated invariants. Once routines complicate the deep import operation a bit. Consider a processor p that wants to deep import an object o handled by a different processor q. For each non-separate once function f of each copied object the following must be done: if a non-separate once function f is fresh on p and non-fresh on q, then f must be marked as non-fresh on p and the value of f on q must be used as the value of f on p. If a once procedure f is fresh on p and non-fresh on q, then f must be marked as non-fresh on p. In all other cases, nothing must be done.

The auxiliary command *deep_import* implements the deep import operation. The command takes an importing processor p and a reference r to be imported. The command returns a state in which the copied object structure exists on the heap and the objects are associated to the respective processors. The copied object structure is accessible through the auxiliary query *last_imported_ref*.

```
$deep\_import$: $\mathbf{STATE} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{REF} \nrightarrow \mathbf{STATE}$
  $\sigma.deep\_import(p,r)$ require
    $\sigma.regions.procs.has(p)$
    $\sigma.heap.refs.has(r)$
  axioms
    $\sigma.deep\_import(p,r) = \sigma'$
    $\sigma.deep\_import(p,r).last\_imported\_ref = r'$
     where
      $w \overset{def}{=} new\ \mathbf{MAP}[\mathbf{REF},\mathbf{REF}].make$
      $(r',w',\sigma') \overset{def}{=} deep\_import\_rec\_with\_map(p,\sigma.handler(r),r,w,\sigma)$
```

The auxiliary command *deep_import* is based on *deep_import_rec_with_map*. This auxiliary function takes a tuple containing an importing processor p, a processor q that handles the root of the object structure to be imported, a reference r to be deep imported, and a state σ to be modified. Note that the object referenced by r is not necessarily handled by q because this object might be on a different processor than the handler of the root of the object structure to be deep imported. The function returns another tuple with a reference r'' to the copied object structure and an updated state σ''. The auxiliary function *deep_import_rec_with_map* works hand in hand with the auxiliary function *deep_import_rec_without_map*. They have the same signature and together they recursively traverse the object structure and make a deep copy of it. The functions must ensure that no object gets copied twice. For this purpose the functions take as an additional argument a map w that maps references to objects in the input data structure to references in the copied data structure. A mapping from one reference x to another reference y means that the object referenced by y is the copy of the object referenced by x. An updated map is returned as part of the result tuple. The auxiliary command *deep_import* starts the recursion with an empty map. The auxiliary function *deep_import_rec_with_map* uses the map to determine whether the object referenced by r has already been copied. In such a case, the result of the function comes from the map. Otherwise the auxiliary function *deep_import_rec_with_map* returns the result of the auxiliary function *deep_import_rec_without_map*. The auxiliary function *deep_import_rec_without_map* creates a copy of the object referenced by r and handles once routines. Finally, it returns a new reference r', an updated map w' in which r is mapped to r', and an updated state σ'.

```
$deep\_import\_rec\_with\_map(p,q,r,w,\sigma) = (r'',w'',\sigma'')$
 where
  $(r',w',\sigma') \overset{def}{=} deep\_import\_rec\_without\_map(p,q,r,w,\sigma)$
  $r'' \overset{def}{=} \begin{cases} w.val(r) & if\, w.keys.has(r) \\ r' & if \neg w.keys.has(r) \end{cases}$
  $w'' \overset{def}{=} \begin{cases} w & if\, w.keys.has(r) \\ w' & if \neg w.keys.has(r) \end{cases}$
  $\sigma'' \overset{def}{=} \begin{cases} \sigma & if\, w.keys.has(r) \\ \sigma' & if \neg w.keys.has(r) \end{cases}$
```

The auxiliary function *deep_import_rec_without_map* is divided into several steps: a copy step, an attribute values update step, a clients update step, a once status update step, and a result generation step. Each of the steps has several definitions associated to it and each set of definitions depends on the definitions of the previous step. The following discussion goes through each of these steps in more details.

The copy step includes the definitions of o, o'_0, σ'_0 and w'_0. The definition o is the object referenced by r, and the definition o'_0 makes a copy of o. In the next step, the function defines an updated state σ'_0 that includes the copy o'_0. There are two cases to be differentiated at this point. If o is handled by q, then o'_0 must be handled by p. Otherwise o'_0 must be handled by the handler of o. The definition w'_0 is the updated map.

The attribute values update step recursively uses *deep_import_rec_with_map* to import all the non-void reference attribute values of o using the updated map. This leads to an updated object with the deep imported values. This step includes the definition of $\{a_1,\ldots,a_n\}$, as well as the definitions of $\{r'_1,\ldots,r'_n\}$, $\{w'_1,\ldots,w'_n\}$, $\{\sigma'_1,\ldots,\sigma'_n\}$, and $\{o'_1,\ldots,o'_n\}$. The set $\{a_1,\ldots,a_n\}$ contains each attributes of o whose value is a non-void reference. The function defines (r'_i, w'_i, σ'_i) for $i = 1\ldots n$ as a sequence of tuples. Each of the tuples is responsible for a single recursive deep import operation for one of the attributes in $\{a_1,\ldots,a_n\}$. Each such operation results in an updated map and an updated state that must be used in the next deep import operation. The result of this is an updated map w'_n, and updated state σ'_n, and references $r_1,\ldots,r_n$ to deep imported data structures. Finally, the function defines a sequence of updated objects $\{o'_1,\ldots,o'_n\}$ that ends with the updated object o'_n. The updated object has the values of the attributes $\{a_1,\ldots,a_n\}$ set to the deep imported data structures referenced by $r_1,\ldots,r_n$.

Until now, the function has an updated state σ'_n that contains the initial copy o'_0. In the client update step, the function updates σ'_n such that the reference to o'_0 points to the updated object o'_n. This is done in the clients update step. This step includes the definition σ'_x. Note that σ'_n is derived from the state σ'_0, which includes the object o'_0.

In a next step, the function takes care of the once routines of the imported object. For this, it defines a new state σ'_y based on the state σ'_x. It defines $\{f_1,\ldots,f_w\}$ as the set of all non-separate once functions of o that are fresh on the processor $\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n))$, which handles the copied object, but non-fresh on the processor $\sigma'_x.\mathit{handler}(r)$, which handles the object referenced by r. Note that the two processor can be the same, in which case the set $\{f_1,\ldots,f_w\}$ is empty. Similarly, it defines the set $\{f_{w+1},\ldots,f_m\}$ for once procedures. For each once routine defined in this way, it updates the state σ'_x such that the once status is taken over to the handler of the copied object. These definitions deal with the case where a once routine is fresh on the handler of the copied object, but non-fresh on the handler of the object referenced by r. Note that the remaining cases are implicitly taken care of because no change to the state is necessary. The result is the state σ'_y.

The last step defines the result of the function, based on the definitions of the preceding steps. The result generation step defines the resulting reference r' to the imported object o'_n, the resulting map w', and the resulting state σ'.

$$
\begin{array}{l}
\mathit{deep\_import\_rec\_without\_map}(p,q,r,w,\sigma) = (r',w',\sigma') \\
\quad \textbf{where} \\
\quad o \overset{def}{=} \sigma.\mathit{ref\_obj}(r) \\
\quad o'_0 \overset{def}{=} o.\mathit{copy} \\
\quad \sigma'_0 \overset{def}{=} \begin{cases} \sigma.\mathit{add\_obj}(p,o'_0) & \mathit{if}\,\sigma.\mathit{handler}(r) = q \\ \sigma.\mathit{add\_obj}(\sigma.\mathit{handler}(r),o'_0) & \mathit{otherwise} \end{cases} \\
\quad w'_0 \overset{def}{=} w.\mathit{add}(r,\sigma'_0.\mathit{ref}(o'_0)) \\
\quad \{a_1,\dots,a_n\} \overset{def}{=} \{a \mid o.\mathit{att\_val}(a) \in \textbf{REF} \wedge o.\mathit{att\_val}(a) \neq \mathit{void}\} \\
\quad \forall i \in \{1,\dots,n\}\colon (r'_i,w'_i,\sigma'_i) \overset{def}{=} \mathit{deep\_import\_rec\_with\_map}(p,q,o.\mathit{att\_val}(a_i),w'_{i-1},\sigma'_{i-1}) \\
\quad \forall i \in \{1,\dots,n\}\colon o'_i \overset{def}{=} o'_{i-1}.\mathit{set\_att\_val}(a_i,r'_i) \\
\quad \sigma'_x \overset{def}{=} \sigma'_n.\mathit{update\_ref}(\sigma'_n.\mathit{ref}(o'_0),o'_n) \\
\quad \sigma'_y \overset{def}{=} \sigma'_x \\
\qquad .\mathit{set\_once\_func\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)),f_1,\sigma'_x.\mathit{once\_result}(\sigma'_x.\mathit{handler}(r),f_1)) \\
\qquad .\dots \\
\qquad .\mathit{set\_once\_func\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)),f_w,\sigma'_x.\mathit{once\_result}(\sigma'_x.\mathit{handler}(r),f_w)) \\
\qquad .\mathit{set\_once\_proc\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)),f_{w+1}) \\
\qquad .\dots \\
\qquad .\mathit{set\_once\_proc\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)),f_m) \\
\quad\; \textbf{where} \\
\qquad \{f_1,\dots,f_w\} \overset{def}{=} \begin{array}{l} \{x \in o.\mathit{class\_type}.\mathit{functions} \mid x.\mathit{is\_once} \wedge \exists c,d\colon \Gamma \vdash x : (d,\bullet,c) \wedge \\ \qquad \sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)),x) \wedge \\ \qquad \neg\sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(r),x)\} \end{array} \\
\qquad \{f_{w+1},\dots,f_m\} \overset{def}{=} \begin{array}{l} \{x \in o.\mathit{class\_type}.\mathit{procedures} \mid x.\mathit{is\_once} \wedge \\ \qquad \sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)),x) \wedge \\ \qquad \neg\sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(r),x)\} \end{array} \\
\quad r' \overset{def}{=} \sigma'_y.\mathit{ref}(o'_n) \\
\quad w' \overset{def}{=} w'_n \\
\quad \sigma' \overset{def}{=} \sigma'_y
\end{array}
$$

Setting values of formal arguments and the value of the current object entity The deep import operation is used in two ways. It is used when an expanded object handled by one processor gets used as an actual argument for a formal argument on another processor. The deep import operation also gets used when an expanded object handled by one processor gets returned to another processor. This section focuses on the argument passing aspect.

The auxiliary command *push_env_with_feature* defines a state in which a processor p receives a new environment. The new environment is initialized for the execution of the feature f with target reference r_0 and actual argument references $(r_1,\dots,r_n)$. Actual arguments of expanded type must either be copied or they must be deep imported.

$\mathit{push\_env\_with\_feature}\colon \textbf{STATE} \to \textbf{PROC} \nrightarrow \textbf{FEATURE} \to \textbf{REF} \to \textbf{TUPLE} \nrightarrow \textbf{STATE}$

$\quad \sigma.\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n))$ **require**

$\qquad \sigma.\mathit{regions.procs.has}(p)$

$\qquad f.\mathit{formals.count} = n$

$\qquad \forall i \in \{0, \ldots, n\}\colon\ r_i \neq \mathit{void} \to \sigma.\mathit{heap.refs.has}(r_i)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n)) =$

$\qquad\quad \sigma'_n.\mathit{set}(\sigma'_n.\mathit{regions}, \sigma'_n.\mathit{heap}, \sigma'_n.\mathit{store.push\_env}(p, e))$

$\qquad$ **where**

$$\sigma'_0 \overset{def}{=} \sigma$$

$$\forall i \in \{1, \ldots, n\}\colon (\sigma'_i, r'_i) \overset{def}{=} \begin{cases} \mathit{if}\,\exists d,q,c\colon \Gamma \vdash f.\mathit{formals}(i) : (d,q,c) \wedge c.\mathit{is\_exp} \wedge r_i \neq \mathit{void} \wedge \sigma'_{i-1}.\mathit{handler}(r_i) \neq p \\ \quad (\sigma_x, \sigma_x.\mathit{last\_imported\_ref}) \\ \quad\ \textbf{where} \\ \quad\quad \sigma_x \overset{def}{=} \sigma'_{i-1}.\mathit{deep\_import}(p, r_i) \\ \mathit{if}\,\exists d,q,c\colon \Gamma \vdash f.\mathit{formals}(i) : (d,q,c) \wedge c.\mathit{is\_exp} \wedge r_i \neq \mathit{void} \wedge \sigma'_{i-1}.\mathit{handler}(r_i) = p \\ \quad (\sigma_x, \sigma_x.\mathit{last\_added\_obj}) \\ \quad\ \textbf{where} \\ \quad\quad \sigma_x \overset{def}{=} \sigma'_{i-1}.\mathit{add\_obj}(p, \sigma'_{i-1}.\mathit{heap.ref\_obj}(r_i).\mathit{copy}) \\ \mathit{otherwise} \\ \quad (\sigma'_{i-1}, r_i) \end{cases}$$

$$\begin{aligned} w \overset{def}{=} &\ \mathit{new}\ \textbf{ENV}.\mathit{make} \\ &\ .\mathit{update}(f.\mathit{formals}(1).\mathit{name}, r'_1)\ \ldots\ .\mathit{update}(f.\mathit{formals}(n).\mathit{name}, r'_n) \\ &\ .\mathit{update}(f.\mathit{locals}(1).\mathit{name}, \mathit{void})\ \ldots\ .\mathit{update}(f.\mathit{locals}(f.\mathit{locals.count}).\mathit{name}, \mathit{void}) \\ &\ .\mathit{update}(\textbf{Current}, r_0) \end{aligned}$$

$$e \overset{def}{=} \begin{cases} w & \mathit{if}\, f \in \textbf{PROCEDURE} \\ w.\mathit{update}(\textbf{Result}, \mathit{void}) & \mathit{if}\, f \in \textbf{FUNCTION} \end{cases}$$

In a first step, the auxiliary command defines an updated state, in which p gets a new initialized environment e. The updated state is based on an intermediate state σ'_n, which gets defined in a cascade of state updates with the goal of either copying or deep importing the actual arguments of expanded type. The cascade starts with the definition of a starting state σ'_0. For each formal argument, the cascade defines a tuple (σ'_i, r'_i) with an updated state and a reference. If the corresponding actual argument is of reference class type, nothing needs to be done. If the actual argument is of expanded class type and the referenced object is not handled by p, then p must deep import the object structure. This results in an updated state and a new reference to the deep imported object structure. If the actual argument is of expanded class type and the referenced object is handled by p, then the expanded object must be copied. This results in an updated state and a new reference to the copy. The resulting state σ'_n contains all the deep imported and copied objects. The resulting references $r'_1, \ldots, r'_n$ will be used for values of the formal argument names.

In a next step, the command defines the environment w as a new environment that gets updated to map formal argument names, local variable names, the current entity name, and the result entity name to the respective values. The names of the formal arguments get mapped to the references $r'_1, \ldots, r'_n$. Names of local variables are mapped to the void reference. The current entity name is mapped to the target reference.

The environment w is the final environment e in which the result name gets mapped to the void reference. This environment and the updated state σ'_n define the result of the command. The auxiliary command *push_env* pushes e onto p's stack of environments. The auxiliary command *push_env* takes a processor p and an environment e. It returns a state in which e is pushed on top of p's environment stack.

The effect of a call to *push_env_with_feature* or a call to *push_env* can be undone with a call to the auxiliary command *pop_env*. This auxiliary command takes a processor p and removes the top environment from p's stack of environments.

Setting values of local variables and the value of the result entity The values of local variables and the value of the result entity are maintained in the store. The auxiliary command *set_env_val* sets a value v for the name n in processor p's top environment. For this, it defines an updated environment e in which n is set to v. It then defines an updated store s by first removing the top environment and then adding the updated environment e. The updated store is then used to define an updated state. The updated state becomes the result of the auxiliary command.

Setting attribute values of the current object The auxiliary command *set_att_val* takes an object o, a name n, and a value v. It returns an updated state in which the attribute with name n of object o is set to the value v. In a first step, the auxiliary command defines an updated object with a call to *set_att_val*. This updated object is then used to update the existing reference to o in the state.

Setting values of local variables, the value of the result entity, and attribute values of the current object in a unified way The auxiliary command *set_val* attaches a value v to an entity with name n. The entity can either be a local variable or the result entity in the top environment of p. It can also be an attribute of the current object on p. In either case, the update affects an entity on p.

The definition of the resulting state is based on the auxiliary definitions o, σ', and v'. The definition o defines the current object, as defined by the top environment of processor p. The precondition makes sure that there is always such an environment on p where the current object is defined. If v is a reference and the referenced object is an object of reference class type, then v can be attached directly to the entity with name n. If the object is an expanded object handled by processor p, then the referenced object must first be copied. Expanded objects handled by a processor different than p must be deep imported. However, this is done right when the object gets returned from another processor to p. The definitions σ' and v' define a state and a value that are potentially updated according to these rules.

The state σ' must be updated with the value v'. The update can either affect the current object on p or it can affect the top environment of p. Attribute names of the current object, local variable names, and formal argument names are distinct. Therefore it is safe to first check whether the current object o has an attribute with name n, in which case the current object gets updated with a v'. If the current object does not have such an attribute, then it is safe to assume that the top environment contains an entity with name n, in which case the top environment gets updated.

```
$set\_val$: $\textbf{STATE} \rightarrow \textbf{PROC} \nrightarrow \textbf{NAME} \nrightarrow \textbf{REF} \cup \textbf{PROC} \nrightarrow \textbf{STATE}$
  $\sigma.set\_val(p,n,v)$ require
    $\sigma.regions.procs.has(p)$
    $\neg\sigma.store.envs(p).is\_empty \wedge \sigma.store.envs(p).top.names.has(\textbf{Current})$
    $v \in \textbf{REF} \wedge v \neq void \rightarrow \sigma.heap.refs.has(v)$
    $v \in \textbf{PROC} \rightarrow \sigma.regions.procs.has(v)$
  axioms
    $\sigma.set\_val(p,n,v) = \begin{cases} if\, \exists a \in o.class\_type.attributes\colon a.name = n \\ \quad \sigma'.set\_att\_val(o,n,v') \\ otherwise \\ \quad \sigma'.set\_env\_val(p,n,v') \end{cases}$
      where
        $o \overset{def}{=} \sigma.heap.ref\_obj(\sigma.store.envs(p).top.val(\textbf{Current}))$
        $(\sigma',v') \overset{def}{=}$
          $\begin{cases} if\, v \in \textbf{REF} \wedge v \neq void \wedge \sigma.heap.ref\_obj(v).class\_type.is\_exp \wedge \sigma.handler(v) = p \\ \quad (\sigma_x, \sigma_x.last\_added\_obj) \\ \quad\; \textbf{where} \\ \quad\quad \sigma_x \overset{def}{=} \sigma.add\_obj(p, \sigma.heap.ref\_obj(v).copy) \\ otherwise \\ \quad (\sigma, v) \end{cases}$
```

Setting values of once functions Values can also be stored in the status of once functions. A once function can be fresh or non-fresh. If the once function is non-fresh on a processor p, then there is a once result for the once function on p. A once function is set as non-fresh during the execution of the once function. The following discussion takes a look at how a processor can set the status of once routines in general, i.e., it considers both once functions and once procedures.

The auxiliary command *set_once_func_not_fresh* takes a processor p, a once function f, and a value r. It returns an updated state in which f is set as non-fresh with the once result r. If f is declared as non-separate, then f is set as non-fresh on p with the once result r. If f is declared as separate with or without an explicit processor specification, then f is set as non-fresh on all processors.

The auxiliary command *set_once_proc_not_fresh* does the same for once procedures. It takes a processor p and a once procedure f and it returns a state in which f is set as non-fresh on p.

Getting values This section takes a look at how a processor can read a value that got written with one of the mechanisms from Sec. 4.6.

Getting values of formal arguments, the value of the current object entity, values of local variables, and the value of the result entity The auxiliary query *envs* takes a processor p and returns the stack of environments for p. The auxiliary query *env_val* is more specialized. It takes a processor p and a name n and it returns the value stored under n in the top environment of p.

Getting attribute values of the current object The auxiliary query *att_val* takes an object o and a name n and returns the attribute value for the attribute with name n of object o.

Getting values of formal arguments, the value of the current object entity, values of local variables, the value of the result entity, and attribute values of the current object in a unified way The auxiliary queries *env_val* and *att_val* define a new auxiliary query *val* that deals both with values in the top environment as well as with values stored in attributes of the current object. The auxiliary query *val* takes a processor p and a name n and it returns the value of n in p's current feature execution context. This context consists of the top environment and its reference to the current object. The auxiliary query requires that the execution context of processor p is setup properly, i.e., there is a top environment with a reference to the current object. The precondition also states that either the top environment has the name n registered or the current object has an attribute with name n. In any valid SCOOP program, any environment variable has a name that is distinct from the attribute names of the current object. This allows us to define the result of the auxiliary query in a simple way. If the name exists in the top environment, then the result is the value given by *env_val*. Otherwise the name must be the name of an attribute of the current object, in which case the result is given by *att_val*.

$$
\begin{array}{l}
val\colon \textbf{STATE} \to \textbf{PROC} \nrightarrow \textbf{NAME} \nrightarrow \textbf{REF} \cup \textbf{PROC} \\
\quad \sigma.val(p,n)\ \textbf{require} \\
\qquad \sigma.regions.procs.has(p) \\
\qquad \neg\sigma.store.envs(p).is\_empty \\
\qquad e.names.has(\textbf{Current}) \\
\qquad e.names.has(n) \vee \exists a \in o.class\_type.attributes\colon\ a.name = n \\
\qquad\ \ \textbf{where} \\
\qquad\quad e \overset{def}{=} \sigma.store.envs(p).top \\
\qquad\quad o \overset{def}{=} \sigma.heap.ref\_obj(e.val(\textbf{Current})) \\
\quad \textbf{axioms}
\end{array}
$$

$$
\sigma.val(p,n) = \begin{cases}
\begin{array}{l}
if\, e.names.has(n) \\
\quad \sigma.env\_val(p,n) \\
\ \ \textbf{where} \\
\quad e \overset{def}{=} \sigma.store.envs(p).top \\
if\, \exists a \in o.class\_type.attributes\colon\ a.name = n \\
\quad \sigma.att\_val(o,n) \\
\ \ \textbf{where} \\
\quad e \overset{def}{=} \sigma.store.envs(p).top \\
\quad o \overset{def}{=} \sigma.heap.ref\_obj(e.val(\textbf{Current}))
\end{array}
\end{cases}
$$

Getting values of once functions The auxiliary query *is_fresh* takes a processor p and a once routine f. It returns whether f is fresh on p or not.

For non-fresh once functions, the auxiliary query *once_result* returns the once result of f on p.

Locking This section explores the aspect of the facade that deals with locking. The auxiliary query *rq_locked* states whether a processor *p*'s request queue is locked or not. There are no auxiliary queries to distinguish between obtained and retrieved locks. Instead, the auxiliary queries *rq_locks* and *cs_locks* return the set of all request queue locks, respectively the set of all call stack locks of a processor *p*. These locks are only usable if they are not passed. This information can be retrieved with a call to the auxiliary query *locks_passed*.

The facade provides auxiliary commands for locking request queues, removing obtained request queue locks, unlocking request queues, delegating obtained request queue locks, passing locks, and revoking locks.

5 Formalization of Execution

This section formalizes the execution of a SCOOP program. It explains the general approach, defines the starting point of the execution, and explains the rules that drive the execution. The rules are divided into rules for mechanisms and rules for code elements.

5.1 General Approach

The formalization is based on *structural operational semantics* [29], combined with parts of the terminology from Ostroff et al. [28]. The idea behind a structural operational semantics is to define the behavior of a program in terms of its parts, i.e., the syntactical elements of the program. Such a semantics is intuitive because it talks directly about elements in the code. It is a very powerful semantics because it allows us to apply structural induction as a proof technique.

Computations A *computation* models the execution of a SCOOP program. It is a sequence of configurations, where each non-initial configuration is derived from a previous configuration through a transition. Each configuration defines a state and a list of statements for each processor. Each transition is described by an inference rule that maps one configuration to another. The transition from one configuration to the next models an atomic step of one processor. The concurrent execution of a SCOOP program is modeled by the interleaved transitions taken by different processors.

Example 4 (Modeling of parallel execution). Suppose there are two processors p and q. Processor p executes the following sequence of statements: $s_{p,1};s_{p,2}$. In parallel, processor q executes the following sequence of statements: $s_{q,1}$. This execution is modeled by any of the following simplified computations: $s_{p,1};s_{p,2};s_{q,1}$ or $s_{p,1};s_{q,1};s_{p,2}$ or $s_{q,1};s_{p,1};s_{p,2}$.

Configurations A *configuration* models a snapshot in the execution of a SCOOP program. A configuration consists of a state and a set of processors, each with a queue of statements. The state is an instance of **STATE**. A *schedule* models the processors and the associated queues, called *action queues*. Each processor must execute the statements in its action queue in a FIFO order. The beginning of the action queue contains the statements for the features that are being executed at the moment. The order of these statements models the way the call stack orders feature executions. The tail of the action queue is the request queue of the processor. A call stack lock is the right to add a feature request to the beginning of the action queue and a request queue lock is the right to add a feature request to the end of the action queue. The notation for a configuration with processors $p_1,\dots,p_n$, respective action queues $s_1,\dots,s_n$, and state σ is:

$\langle p_1 :: s_1 \mid \dots \mid p_n :: s_n, \sigma\rangle$

The processor separator $\mid$ is commutative and associative, i.e., $p_1 :: s_1 \mid p_2 :: s_2 = p_2 :: s_2 \mid p_1 :: s_1$ and $p_1 :: s_1 \mid (p_2 :: s_2 \mid p_3 :: s_3) = (p_1 :: s_1 \mid p_2 :: s_2) \mid p_3 :: s_3$. Within an action queue, ; separates statements. The configuration is *well-defined* if and only if $\neg\exists i,j \in \{1,\dots,n\}\colon\ p_i = p_j$.

Statements A *statement* is an element of the action queue. A statement is either an instruction or an operation. An *instruction* is user syntax, i.e. an action that occurs explicitly in the SCOOP program. An *operation* is run-time syntax, i.e. an action that does not explicitly occur in a SCOOP program. For example, locking of request queues is not an action that is explicit in a SCOOP program. Instead, locking is based on the formal argument list. It is done implicitly before a feature gets executed.

Transitions A *transition* takes a system in a start configuration and leaves it in a result configuration. The following shows the general form of a transition definition that declares a start configuration $\langle P,\sigma\rangle$ with schedule $P \overset{def}{=} p_1 :: s_1 \mid \dots \mid p_n :: s_n$ and a result configuration $\langle P',\sigma'\rangle$ with schedule $P' \overset{def}{=} p'_1 :: s'_1 \mid \dots \mid p'_m :: s'_m$:

$\Gamma \vdash \langle P,\sigma\rangle \to \langle P',\sigma'\rangle$

The typing environment Γ can be used in the transition definition to access static information about the SCOOP program.

Inference rules An *inference rule* describes the circumstances under which a transition can be used. The inference rule has a premise and a conclusion. The *conclusion* is the transition and the *premise* describes the circumstances under which the transition can be used. The premise consists of a number of transitions and a side condition. The premise is satisfied if all transitions in the premise can be taken and if the side condition is true. In this formalization, most of the rules have no transition in the premise. The following *simplified inference rule template* takes this into account:

Simplified Inference Rule Template

$$\frac{\begin{array}{l}\textit{condition}\\ \textit{new state}\ \sigma'\ \textit{definition}\\ \textit{fresh channels definitions}\end{array}}{\Gamma \vdash \langle P,\sigma\rangle \to \langle P',\sigma'\rangle}$$

The side condition has three parts. The first part defines a *condition* that is based on the typing environment and the start configuration. The second part is the *new state definition* that defines the state of the result configuration. This new state is based on the state in the start configuration. The last part consists of the *fresh channels definitions*. Auxiliary definitions can be used in the condition, the new state definition, and the fresh channels definitions. The side condition can mention features of **STATE**. The preconditions of these features serve as additional conditions in the side condition.

The following inference rule generalizes transitions by adding processors both to the start configuration and to the result configuration. These additional processors run in parallel but do not take any actions during the generalized transition.

Parallelism

$$\frac{\Gamma \vdash \langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle}{\Gamma \vdash \langle P \mid Q, \sigma \rangle \rightarrow \langle P' \mid Q, \sigma' \rangle}$$

Scheduling Before a processor can execute a feature it must acquire locks and it must wait until the wait condition is satisfied. A locking request encapsulates these two requirements; it consists of the requested locks and the wait condition. At every moment, multiple processors can have conflicting locking requests. The scheduler is the arbiter for these conflicts. The scheduler takes locking requests and stores them in a queue. It then approves locking requests according to a certain scheduling algorithm.

The model permits a number of possible scheduling algorithms. The algorithms differ in their level of fairness and their performance. This formalization does not focus on a particular scheduling algorithm. Instead, it uses the conditions of the inference rules to express locking requests. If more than one processor satisfies the conditions, then any of these processors can proceed.

5.2 Initial Configuration

The initial configuration is defined by the SCOOP program. Each SCOOP program defines a root class type c and a root procedure f. The root procedure is a creation procedure of the root class type that has no formal arguments and no precondition.

In the beginning, the runtime generates a bootstrap processor p and root processor q with a root object of the root class type. The request queue of the root processor is locked on behalf of the bootstrap processor. This defines our initial state σ:

$\sigma_x \overset{def}{=} new\ \mathbf{STATE}.make$

$\sigma_y \overset{def}{=} \sigma_x.add\_proc(\sigma_x.new\_proc)$

$p \overset{def}{=} \sigma_y.last\_added\_proc$

$\sigma_z \overset{def}{=} \sigma_y.add\_proc(\sigma_y.new\_proc)$

$q \overset{def}{=} \sigma_z.last\_added\_proc$

$\sigma_w \overset{def}{=} \sigma_z.add\_obj(q, \sigma_z.new\_obj(c))$

$r \overset{def}{=} \sigma_w.ref(\sigma_w.last\_added\_obj)$

$\sigma \overset{def}{=} \sigma_w.lock\_rqs(p, \{q\})$

The bootstrap processor first asks the root processor to execute the root procedure on the root object and then asks the root processor to unlock its request queue as soon as it finished the execution. The bootstrap processor can do this because it has the request queue lock on the root processor. Finally, the bootstrap processor removes the request queue lock from its stack of obtained request queue locks. This is shown in the following initial configuration:

$$\begin{array}{l}
\langle \\
\quad p :: \texttt{call}(r, f, (), ()); \\
\quad\quad \texttt{issue}(q, \texttt{unlock}); \\
\quad\quad \texttt{pop\_obtained\_rq\_locks} \mid \\
\quad q :: \\
, \\
\quad \sigma \\
\rangle
\end{array}$$

The statements `call`, `issue`, `unlock`, and `pop_obtained_rq_locks` are operations. In a nutshell, the $\texttt{call}(r, f, (), ())$ operation asks the handler of the target r to make a call to the feature f on target r. The `unlock` operation unlocks the request queue of the processor that executes the operation. The $\texttt{issue}(q, \texttt{unlock})$ operation adds the `unlock` operation to q's action queue. The `pop_obtained_rq_locks` operation removes the top element from the stack of obtained request queue locks.

5.3 Mechanisms

Mechanisms are the machinery for the execution of code elements. This section studies these mechanisms.

Issuing mechanism With the issuing mechanism, a processor p can add statements to the action queue of a processor q. It uses the `issue` operation to get a result configuration in which a processor's action queue is extended with the new statements. There are two main cases: p adds the statements to its own action queue, i.e., $p = q$, or p adds the statements to the action queue of a different processor, i.e., $p \neq q$. The first case is the non-separate case and the second one is the separate case.

For the non-separate case p puts the statements to the beginning of q's action queue, which is the same as putting the statements on top of the call stack. This requires that p is in possession of its own call stack lock.

Issue Operation – Non-Separate

$$\frac{\begin{array}{l} q = p \\ \neg\sigma.locks\_passed(p) \\ \sigma.cs\_locks(p).has(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p, \sigma\rangle \rightarrow \langle p :: s_w; s_p, \sigma\rangle}$$

For the separate case there is a difference between a normal and a callback case. In the normal case, p adds the statements to the end of q's action queue. This case requires that p is in possession of q's request queue lock. To distinguish the normal case from the callback case, this case also requires that q does not have a lock on p.

Issue Operation – Separate

$$\frac{\begin{array}{l} q \neq p \\ \neg\sigma.locks\_passed(p) \\ \sigma.rq\_locks(p).has(q) \\ \neg(\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma \rangle \rightarrow \langle p :: s_p \mid q :: s_q; s_w, \sigma \rangle}$$

The callback case occurs if q has a lock on p. In this situation, p could issue a statement s_w on q and then wait for q to complete. On the other side, processor q could already be waiting for p to complete. Processor q would be waiting for p to finish and p would be waiting for q to finish. However, since s_w would be at the end of q's action queue and q would be waiting there cannot be any progress. This type of deadlock can be prevented by adding s_w not to the of q's action queue but to the beginning. This will make sure that q can execute the statement right away and hence p can continue. This in return will enable q to continue. As a prerequisite, p must possess q's call stack lock.

Issue Operation – Separate Callback

$$\frac{\begin{array}{l} q \neq p \\ \neg\sigma.locks\_passed(p) \\ \sigma.cs\_locks(p).has(q) \\ \sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma \rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma \rangle}$$

Delegated execution mechanism This section discusses how a processor q can delegate the execution of statements to a different processor p. This mechanism is useful for the evaluation of asynchronous postconditions. Processor q must make sure that the statements make sense in the context of processor p. The names that occur in these statements must be defined in the top environment of p and p must have the necessary locks to execute the statements. Statements that fulfill the following conditions can be delegated:

– All names that occur in the statements are defined in q's top environment.
– Their execution only requires the top set of q's stack of obtained request queue locks.

These conditions exclude statements that involve non-separate calls or separate callbacks because such calls require a call stack lock. If these conditions are met, q can transfer its top environment and the top of its obtained request queue locks to p. Given this context, p can then execute the delegated statements instead of q.

The $\texttt{execute\_delegated}(s_w, x, \{q_1, \dots, q_m\})$ operation sets up a new context on p with an environment x and obtained request queue locks $\{q_1, \dots, q_m\}$. To set up the new context, the operation uses a combination of the commands *push_env* and *delegate_obtained_rq_locks*. The command *delegate_obtained_rq_locks* requires that the request queue locks $\{q_1, \dots, q_m\}$ are not in possession of another processor anymore. It also requires that the request queues of $\{q_1,\ldots,q_m\}$ are locked. Once the context is set up, processor p executes the statements s_w and then gets rid of the context, using the `leave_delegated` operation.

To delegate the execution of the statements s_w, processor q must make sure that its top environment x is set up correctly and it must make sure that the top set of its obtained request queue locks contains all locks $\{q_1,\ldots,q_m\}$ that are necessary for the execution of s_w. Processor q must then issue a $\texttt{execute\_delegated}(s_w,x,\{q_1,\ldots,q_m\})$ operation to processor p. Processor q must then remove $\{q_1,\ldots,q_m\}$ from its stack of obtained request queue locks so that the *delegate_obtained_rq_locks* operation can take place.

Execute Delegated Operation

$$\frac{\begin{array}{l}\forall x \in \{q_1,\ldots,q_m\}\colon \neg\exists y \in \sigma.procs\colon \sigma.rq\_locks(y).has(x)\\ \forall x \in \{q_1,\ldots,q_m\}\colon \sigma.rq\_locked(x)\\ \sigma' \overset{def}{=} \sigma.push\_env(p,x).delegate\_obtained\_rq\_locks(p,\{q_1,\ldots,q_m\})\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{execute\_delegated}(s_w,x,\{q_1,\ldots,q_m\}); s_p, \sigma\rangle \rightarrow\\ \quad \langle p :: s_w; \texttt{leave\_delegated}; s_p, \sigma'\rangle\end{array}}$$

Leave Delegated Execution Operation

$$\frac{\begin{array}{l}\neg\sigma.envs(p).is\_empty\\ \neg\sigma.obtained\_rq\_locks(p).is\_empty\\ \sigma' \overset{def}{=} \sigma.pop\_env(p).pop\_obtained\_rq\_locks(p)\end{array}}{\Gamma \vdash \langle p :: \texttt{leave\_delegated}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

Notification mechanism Processors can notify each other. A notification can optionally include a value. The formalization uses channels to describe such communication. Channels are described in Milner's π-calculus [23]. In the π-calculus, the expression $c(x).P$ denotes a process that is waiting for a notification sent on a channel c. Once the notification has been received, the value of the notification is bound to the variable x and the process continues with the expression P. The notification comes from a process that executes $\bar{c}y.Q$ to emit the value y on the channel c before executing Q.

The formalization reuses the channel idea in two flavors: once as a notification mechanism with a value and once as a notification mechanism without a value. A processor sends a notification with a value r over a channel a as it executes the operation $\texttt{result}(a,r)$. Similarly, the process sends a notification without a value over a channel a by executing the operation $\texttt{notify}(a)$. For both cases, any processor can wait for a notification by executing the operation $\texttt{wait}(a)$. In case a notification on a channel a carries a value, the value can be accessed with $a.data$. This way of accessing the value of a channel is different from the way it is done in the π-calculus. In the π-calculus, each value is bound to a variable. This formalization does not define a new variable for the value. Instead, it uses $a.data$ to identify the value of a channel a.

A number of inference rules describe the interaction between a processor that sent a notification over a channel and a processor that is waiting for a notification over the same channel. Two main cases can be distinguished: either a processor sends a notification to itself or it sends a notification to a different processor. The first case is the non-separate case and the latter case is the separate case. In each of these two main cases, the channel carries a notification with or without a value. For each of these sub cases, there is one inference rule.

In the non-separate case, one processor has a `result`(a,r) operation or a `notify`(a) operation at the beginning of its action queue and a `wait`(a) operation on the same channel later in the action queue. In this case, the `wait`(a) operation can be removed along with the `result`(a,r) operation, respectively the `notify`(a) operation. If the channel carries a value, then the value must be installed on the processor, by substituting all occurrences of $a.data$ with the posted value in all the statements s_p after the `wait`(a) operation.

Wait and Result Operation – Non-Separate

$$\frac{}{\Gamma \vdash \langle p :: \texttt{result}(a,r);s_w;\texttt{wait}(a);s_p,\sigma\rangle \to \langle p :: s_w;s_p[r/a.data],\sigma\rangle}$$

Wait and Notify Operation – Non-Separate

$$\frac{}{\Gamma \vdash \langle p :: \texttt{notify}(a);s_w;\texttt{wait}(a);s_p,\sigma\rangle \to \langle p :: s_w;s_p,\sigma\rangle}$$

In the separate case, one processor has a `result`(a,r) or a `notify`(a) operation at the beginning of its action queue and a different processor has a `wait`(a) somewhere in its action queue. In this situation, the `wait`(a), `result`(a,r), and `notify`(a) can be removed from the action queues. In case the notification has a value, the value can be installed in the statements s_p, after the `wait`(a) operation.

Wait and Result Operation – Separate

$$\frac{}{\Gamma \vdash \langle p :: s_w;\texttt{wait}(a);s_p \mid q :: \texttt{result}(a,r);s_q,\sigma\rangle \to \langle p :: s_w;s_p[r/a.data] \mid q :: s_q,\sigma\rangle}$$

Wait and Notify Operation – Separate

$$\frac{}{\Gamma \vdash \langle p :: s_w;\texttt{wait}(a);s_p \mid q :: \texttt{notify}(a);s_q,\sigma\rangle \to \langle p :: s_w;s_p \mid q :: s_q,\sigma\rangle}$$

The operations presented here must be used so that each `wait` operation can be resolved with exactly one `result` or `notify` operation. To define this condition more precisely, we define that one statement s_1 weakly precedes a statement s_2 if and only if s_1 occurs earlier than s_2 in the same action queue or s_1 and s_2 occur in different action queues. One statement s_1 strongly precedes a statement s_1 if and only if s_1 occurs earlier than s_2 in the same action queue. With these definitions, the condition says:

- For each `wait`(a) operation there must be either exactly one `result`(a,r) or exactly one `notify`(a) operation.
- For each `result`(a,r) or `notify`(a) operation there must be exactly one `wait`(a) operation.
- Each `result`(a,r) or `notify`(a) operation weakly precedes the `wait`(a) operation.

Expression evaluation mechanism An expression can either be a literal, an entity, or a query call. The query call can contain actual arguments that are expressions themselves. This section discusses the general mechanism to evaluate expressions. It focuses on the general approach and defers the evaluation of particular expressions to later sections.

The operation `eval`(a, e) takes a channel a and an expression e. Each `eval`(a, e) operation determines the value r of the expression e and then sends a notification with value r on channel a. This means that each `eval`(a, e) operation creates a `result`(a, r) operation in the action queue. It is therefore important to follow each `eval`(a, e) operation with exactly one `wait`(a) to receive the notification with the value.

Locking and unlocking mechanism A processor p that wants to execute a feature must first obtain the request queue locks of a number of processors $\{q_1, \ldots, q_n\}$. For this, p adds $\{q_1, \ldots, q_n\}$ on top of its obtained request queue locks stack. Only then can p issue statements to these processors. The `lock`$(\{q_1, \ldots, q_n\})$ operation serves this purpose. The operation requires that none of the request queues is already locked.

Lock Operation

$$\frac{\begin{array}{l}\neg\exists q_i \in \{q_1, \ldots, q_m\}\colon \sigma.rq\_locked(q_i) \\ \sigma' \overset{def}{=} \sigma.lock\_rqs(p, \{q_1, \ldots, q_m\})\end{array}}{\Gamma \vdash \langle p :: \texttt{lock}(\{q_1, \ldots, q_m\}); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

Once p is done with the execution of the feature, it asks $\{q_1, \ldots, q_n\}$ to unlock their request queues once they are done with the issued statements. For this purpose, the `unlock` operation unlocks the request queue. Processor p issues the `unlock` operation to processors $\{q_1, \ldots, q_n\}$. This operation requires that the request queue is indeed locked and that no processor possesses the request queue lock.

Unlock Operation

$$\frac{\begin{array}{l}\sigma.rq\_locked(p) \\ \forall q \in \sigma.procs\colon \neg\sigma.rq\_locks(q).has(p) \\ \sigma' \overset{def}{=} \sigma.unlock\_rq(p)\end{array}}{\Gamma \vdash \langle p :: \texttt{unlock}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

After p issued the `unlock` operations, it can remove $\{q_1, \ldots, q_n\}$ from its stack of obtained request queue locks using the `pop_obtained_rq_locks` operation. This ensures that the `unlock` operations can proceed.

Pop Obtained Request Queue Locks

$$\frac{\sigma' \overset{def}{=} \sigma.pop\_obtained\_rq\_locks(p)}{\Gamma \vdash \langle p :: \texttt{pop\_obtained\_rq\_locks}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

Brooke, Paige, and Jacob [5] noticed that `unlock` operations are not optimal. In essence, it could be possible to unlock the request queue of a processor q_i directly after p issued all statements. The request queue lock is important to guarantee exclusive access on q_i's request queue. However, as soon as p issued all statements on q_i, this lock is no longer needed. Unlocking the request queue right away could improve the performance in some situations because q_i's request queue could be locked again earlier and hence another processor that is waiting for this lock could proceed earlier.

Write and read mechanism A processor p can use the `write`(x,v) operation to set a value v of an entity with name x. This operation uses the *set_val* command. Hence, p can both set attribute values of its current object and values of entities in its top environment.

Write Value Operation

$$\frac{\sigma' \overset{def}{=} \sigma.set\_val(p,x,v)}{\Gamma \vdash \langle p :: \texttt{write}(x,v);s_p,\sigma\rangle \to \langle p :: s_p,\sigma'\rangle}$$

Similarly, processor p can execute the `read`(x,a) operation to read a value of an entity with name x and send the value over channel a. The `read` operation does not present its result in a `result` operation because, unlike an `eval` operation, a `read` operation always produces a result for the surrounding action queue. It is easier to do the substitution of the channel access directly. A later section introduces the `eval` operation for entity expressions. This variant of the `eval` operation makes use of the `read` operation and presents the result in a `result` operation.

Read Value Operation

$$\frac{}{\Gamma \vdash \langle p :: \texttt{read}(x,a);s_p,\sigma\rangle \to \langle p :: s_p[\sigma.val(p,x)/a.data],\sigma\rangle}$$

Finally, there is the `set_not_fresh` operation in a variant for once functions and in a variant for once procedures. This operation sets the once status of a once routine. The variant `set_not_fresh`(f,r) sets the once status of a once function f to non-fresh with value r. If f is of separate type, then the once function becomes non-fresh on all processors in the system. If f has a non-separate type, then f becomes non-fresh only on processor p. The variant `set_not_fresh`(f) sets the once status of a once procedure f to non-fresh on processor p.

Set Once Routine Not Fresh Operation – Function

$$\frac{\begin{array}{l} f \in \textbf{FUNCTION} \wedge f.is\_once \\ \sigma' \overset{def}{=} \sigma.set\_once\_func\_not\_fresh(p,f,r)\end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f,r);s_p,\sigma\rangle \to \langle p :: s_p,\sigma'\rangle}$$

Set Once Routine Not Fresh Operation – Procedure

$$\frac{\begin{array}{l} f \in \textbf{FUNCTION} \wedge f.is\_once \\ \sigma' \overset{def}{=} \sigma.set\_once\_proc\_not\_fresh(p,f)\end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f);s_p,\sigma\rangle \to \langle p :: s_p,\sigma'\rangle}$$

Flow control mechanism In addition to flow control instructions in the user code, there are flow control operations, which implement flow control in the inference rules. This way, fewer inference rules are required because multiple variants can be handled in one inference rule.

The `provided` x `then` s_t `else` s_f `end` operation takes the condition x as an argument. The operation either executes s_t if x indicates that the condition is true or s_f if x

indicates that the condition is false. For each possibility there is one inference rule. The condition x can either be an instance of **BOOLEAN** or it can be a reference that points to an object of class type *BOOLEAN*. To decide which branch to take, the operation must evaluate x. If x is an instance of **BOOLEAN**, then it can determine which instance x is, i.e., *true* or *false*. If x is a reference, then it must get the referenced object and see which boolean value it represents. For this purpose, it evaluates the attribute *item* of the referenced object.

If Operation – True

$$\frac{y \overset{def}{=} \begin{cases} x & if\, x \in \mathbf{BOOLEAN} \\ \sigma.att\_val(\sigma.ref\_obj(x), item) & if\, x \in \mathbf{REF} \wedge \sigma.ref\_obj(x).class\_type = BOOLEAN \\ false & otherwise \end{cases} \qquad y = true}{\Gamma \vdash \langle p :: \texttt{provided}\, x\, \texttt{then}\, s_t\, \texttt{else}\, s_f\, \texttt{end}; s_p, \sigma\rangle \to \langle p :: s_t; s_p, \sigma\rangle}$$

If Operation – False

$$\frac{y \overset{def}{=} \begin{cases} x & if\, x \in \mathbf{BOOLEAN} \\ \sigma.att\_val(\sigma.ref\_obj(x), item) & if\, x \in \mathbf{REF} \wedge \sigma.ref\_obj(x).class\_type = BOOLEAN \\ true & otherwise \end{cases} \qquad y = false}{\Gamma \vdash \langle p :: \texttt{provided}\, x\, \texttt{then}\, s_t\, \texttt{else}\, s_f\, \texttt{end}; s_p, \sigma\rangle \to \langle p :: s_f; s_p, \sigma\rangle}$$

The `provided` x `then` s_t `else` s_f `end` operation has two branches. Sometimes it is necessary to only have one branch. The `nop` operation can be executed without an effect. It can be used in the conditional operation to define an empty branch. The `nop` operation can also be used to indicate that an action queue is empty.

No Operation

$$\frac{}{\Gamma \vdash \langle p :: \texttt{nop}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma\rangle}$$

5.4 Code Elements

This section explains the semantics of code elements: entity expressions, literal expressions, feature calls, feature applications, creation instructions, flow control instructions, and assignment instructions.

Entity expressions A variant of the `eval`(a,e) operation evaluates entity expressions. The operation uses the `read` operation to send a notification with the value of the entity over a new channel a'. It then uses the value of this channel to define the result of the `eval` operation.

Entity Expression

$$\frac{e \in \mathbf{ENTITY} \qquad a' \text{ is fresh}}{\Gamma \vdash \langle p :: \texttt{eval}(a,e); s_p, \sigma\rangle \to \langle p :: \texttt{read}(e.name, a'); \texttt{result}(a, a'.data); s_p, \sigma\rangle}$$

Literal expressions Another variant of the $\texttt{eval}(a,e)$ operation evaluates literal expressions. To evaluate a non-void literal expression, the operation creates a new object of the literal class type so that the new object represents the literal value. For this purpose, it uses the query *obj* of **LITERAL**. Since the type of every literal is non-separate, it creates the new object on the processor that evaluates the literal expression. The reference r to the new object is the result of the evaluation. To evaluate a void literal, the operation takes the void reference.

Literal Expression

$$\frac{\begin{array}{l} e \in \textbf{LITERAL} \\ \sigma' \overset{def}{=} \begin{cases} \sigma & \textit{if } e = \textbf{Void} \\ \sigma.\textit{add\_obj}(p, e.\textit{obj}) & \textit{otherwise} \end{cases} \\ r \overset{def}{=} \begin{cases} \textit{void} & \textit{if } e = \textbf{Void} \\ \sigma'.\textit{ref}(\sigma'.\textit{last\_added\_obj}) & \textit{otherwise} \end{cases} \end{array}}{\Gamma \vdash \langle p :: \texttt{eval}(a,e); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a,r); s_p, \sigma'\rangle}$$

Feature calls A feature call can occur in two ways. First, a feature call can be a call to a command in a command instruction. Second, a feature call can be a call to a query in an expression. This section studies both variants. A processor p that executes a feature call $e_0.f(e_1,\ldots,e_n)$ goes through the following steps:

1. Target evaluation. Evaluate the target expression e_0 and let q denote the handler of the target.
2. Argument passing. Evaluate the actual arguments expressions $(e_1,\ldots,e_n)$.
3. Lock passing. Determine which locks to pass to q.
   - Take all request queue locks and call stack locks if a controlled actual argument gets attached to an attached formal argument of reference type.
   - Take all request queue locks and call stack locks if the feature call is a separate callback, i.e., q has a lock on p.
   - Otherwise, take no locks.
4. Feature request.
   - Ask q to apply f to the target immediately and wait until the execution terminates if any of the following conditions holds:
     - The feature call is non-separate, i.e., $p = q$.
     - The feature call is a separate callback, i.e., q has a lock on p.
   - Otherwise, ask q to apply f to the target after the previous feature requests.
5. Wait by necessity. If f is a query, then wait for the result.
6. Lock revocation. If lock passing happened, then wait for the locks to come back.

A command instruction is a statement in the action queue. A query is an expression on the right hand side of an assignment, a condition in a flow control instruction, or an actual argument in a feature call. Whenever a query occurs in one of these constructs, the inference rule of the construct encloses the query in an `eval` operation. To handle feature calls, there is an inference rule for command instructions and a variant of the `eval` operation for query calls.

In each case, the statement first evaluates the target expression and all actual argument expressions. For each of these expressions e_i, it uses one $\texttt{eval}(a_{e_i}, e_i)$ operation and a corresponding $\texttt{wait}(a_{e_i})$ operation with a fresh channel a_{e_i}. Each of the channel values gets used in the subsequent `call` operation. With this, the statement handled the target evaluation and the argument passing step. It defers the attachment of the actual arguments to the formal arguments to the point where the called feature gets applied. The reason for this is simple: at this point the context for the feature application does not exist yet.

The `call` operation takes care of the remaining steps. The operation exists in two variants, one for command instructions and one for queries. The variant for queries takes a channel a' and uses it for the result of the query. Since a call to a command does not produce a result, such a channel is not required for command instructions. Both `call` variants take the reference to the target a_{e_0}, the feature f to be called, the references to the actual arguments $(a_{e_1}.data, \ldots, a_{e_n}.data)$, and the actual argument expressions $(e_1, \ldots, e_n)$). The actual argument expressions are used to check whether there is a controlled actual argument. This information determines whether the locks should be passed.

Command Instruction

$$\frac{\forall i \in \{0, \ldots, n\}: a_{e_i}\ is\ fresh}{\begin{array}{l} \Gamma \vdash \langle p :: e_0.f(e_1, \ldots, e_n); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a_{e_0}, e_0); \texttt{eval}(a_{e_1}, e_1); \ldots; \texttt{eval}(a_{e_n}, e_n); \\ \qquad \texttt{wait}(a_{e_0}); \texttt{wait}(a_{e_1}); \ldots; \texttt{wait}(a_{e_n}); \\ \qquad \texttt{call}(a_{e_0}.data, f, (a_{e_1}.data, \ldots, a_{e_n}.data), (e_1, \ldots, e_n)); \\ \qquad s_p, \sigma\rangle \end{array}}$$

Query Expression

$$\frac{\begin{array}{l}\forall i \in \{0, \ldots, n\}: a_{e_i}\ is\ fresh \\ a'\ is\ fresh\end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{eval}(a, e_0.f(e_1, \ldots, e_n)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a_{e_0}, e_0); \texttt{eval}(a_{e_1}, e_1); \ldots; \texttt{eval}(a_{e_n}, e_n); \\ \qquad \texttt{wait}(a_{e_0}); \texttt{wait}(a_{e_1}); \ldots; \texttt{wait}(a_{e_n}); \\ \qquad \texttt{call}(a', a_{e_0}.data, f, (a_{e_1}.data, \ldots, a_{e_n}.data), (e_1, \ldots, e_n)); \\ \qquad \texttt{result}(a, a'.data); \\ \qquad s_p, \sigma\rangle \end{array}}$$

Both variants of the `call` operation take the reference to the target r_o, the feature f to be called, the references to the actual arguments $(r_1, \ldots, r_n)$, and the actual argument expressions $(e_1, \ldots, e_n)$. The variant for queries takes an additional channel a to be used for the result of the query. In a first step, the operation must evaluate the handler q of the target. The handler is used in an `issue` operation to issue a feature request on the responsible processor. The feature request comes in the form of an `apply` operation. The `apply` operation takes a channel a for the communication between p and q, the target reference r_0, the called feature f, the references to the actual arguments $(r_1, \ldots, r_n)$, the caller processor p, and the passed locks $\bar{l}$.

Clarification 3 (Lock passing). Processor p passes all its request queue locks and all its call stack locks either if there is a controlled actual argument that will get attached to an attached formal argument of reference type or if the feature call is a separate callback. An attached formal argument of reference type means that the request queue lock or the call stack lock on the actual argument's handler is required during the application of f. A controlled actual argument means that p has a request queue lock or a call stack lock on the handler of the actual argument. In short, p has a lock that is required by q and thus p has to pass the locks. A separate callback occurs if q has a lock on p. In this situation, p can issue a statement to q and then wait for q to complete. However, processor q could already be waiting for p to complete. To handle this case, the `issue` operation in the `call` operation triggers an immediate execution by adding the `apply` to the beginning of q's action queue. The `issue` operation requires that p has the call stack lock of q. To enable q to perform an immediate execution, p has to give back q's call stack lock.

In both cases, p has to wait for the locks to come back. Thus it does not hurt to pass all the locks in both cases. In contrast to Nienaltowski's [25] description of SCOOP, p only passes the locks that it really has. In particular, p does not pass its own request queue lock in situations where p does not possess this lock, such as when the processor that called p possesses p's request queue lock. □

In the cases where the operation passes the locks, $\bar{l}$ is $(\sigma.rq\_locks(p), \sigma.cs\_locks(p))$. In all other cases there is no lock passing and thus $\bar{l} = (\{\}, \{\})$. The operation just determines which locks to pass. The actual lock passing action will be executed by q. Similarly, the actual lock revocation action will be executed by q.

For command calls, lock passing is the only reason to wait. In this case, the operation creates a fresh channel a to wait for a notification from q. The notification arrives when q is ready to return the locks. For query calls, the operation has to wait for the result. The operation uses the given channel a to wait for the result. This has the advantage that once the result arrives, it will be substituted after the `call` operation, i.e. in the `result` operation of the `eval` operation.

Call Operation – Command

$$\frac{\begin{array}{l} q \overset{def}{=} \sigma.handler(r_0) \\ \bar{l} \overset{def}{=} \begin{cases} \textit{if} \\ \quad q \neq p \wedge \exists i \in \{1, \dots, n\} : \Gamma \vdash e_i : t \wedge controlled(t) \wedge \Gamma \vdash f.formals(i) : (!, g, c) \wedge c.is\_ref \\ \textit{then} \\ \quad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \textit{if} \\ \quad q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \textit{then} \\ \quad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \textit{otherwise} \\ \quad (\{\}, \{\}) \end{cases} \\ a\ is\ fresh \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{call}(r_0, f, (r_1, \dots, r_n), (e_1, \dots, e_n)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{issue}(q, \texttt{apply}(a, r_0, f, (r_1, \dots, r_n), p, \bar{l})); \\ \qquad \texttt{provided}\ \bar{l} \neq (\{\}, \{\})\ \texttt{then}\ \texttt{wait}(a)\ \texttt{else}\ \texttt{nop}\ \texttt{end}; \\ \qquad s_p, \sigma\rangle \end{array}}$$

Call Operation – Query

$$
\frac{
\begin{array}{l}
q \overset{def}{=} \sigma.handler(r_0) \\
\bar{l} \overset{def}{=} \begin{cases}
\textit{if} \\
\quad q \neq p \wedge \exists i \in \{1,\dots,n\} : \Gamma \vdash e_i : t \wedge controlled(t) \wedge \Gamma \vdash f.formals(i) : (!, g, c) \wedge c.is\_ref \\
\textit{then} \\
\quad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\
\textit{if} \\
\quad q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\
\textit{then} \\
\quad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\
\textit{otherwise} \\
\quad (\{\}, \{\})
\end{cases}
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{call}(a, r_0, f, (r_1, \dots, r_n), (e_1, \dots, e_n)); s_p, \sigma \rangle \rightarrow \\
\quad \langle p :: \texttt{issue}(q, \texttt{apply}(a, r_0, f, (r_1, \dots, r_n), p, \bar{l})); \texttt{wait}(a); s_p, \sigma \rangle
\end{array}
}
$$

Feature applications A feature call by a client processor q results in a feature request for a supplier processor p. A *feature application* is the serving of the feature request. This section discusses how p applies a feature f on a target referenced by r_0. Processor p takes the following steps:

1. Once status update. If f is a once routine, then set its status to non-fresh.
2. Lock passing. Pass the locks from q to p.
3. Argument passing. Bind the actual arguments to the formal arguments. Arguments of expanded type that are handled by a different processor than p must be deep imported by p.
4. Synchronization. Involve the scheduler to wait until the following synchronization conditions are satisfied atomically:
   - Processor p owns the request queue lock of each processor q such that:
     - Processor q handles an actual argument of f and the corresponding formal argument has an attached reference type.
     - Processor p and processor q are different.
     - Processor p does not have q's request queue lock.
     - Processor q does not have p's request queue lock.
   - The precondition of f holds.
5. Execution.
   - If f is a non-once routine or a fresh once routine, then run its body.
   - If f is a non-fresh procedure, then do nothing. If f is a non-fresh function, then take its once value as the result.
   - If f is an attribute, then evaluate it.
6. Postcondition evaluation. Evaluate the postcondition if any of the following conditions is satisfied:
   - A feature call in the postcondition requires a lock that was not obtained in the synchronization step.
   - The evaluation of the postcondition involves lock passing.

   Otherwise ask any processor whose request queue lock was obtained in the synchronization step to evaluate the postcondition.

7. Lock releasing. Ask each processor whose request queue has been locked in the synchronization step to unlock its request queue after it is done with the feature requests issued by p.
8. Invariant evaluation. If f is a routine, then evaluate the invariant.
9. Result returning. If f is a query, then return the result to q. If the result is of expanded type and $p \neq q$, then the result must be deep imported by q.
10. Lock revocation. Return the passed locks from p to q.

Each feature application starts with an operation `apply`$(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l})$ in the action queue of processor p. The channel a is used to communicate with the client processor q. If the called feature f is a procedure and the caller processor q passed some locks, then a is used to signal that the locks returned. If f is query, then a is used to return the value. The reference r_0 points to the target of the call. The references $(r_1, \ldots, r_n)$ point to the actual arguments. The tuple $\bar{l}$ contains the locks to be passed from q to p.

If one takes a look at the execution step, one can differentiate three cases:

– The feature f is a non-once routine or a fresh once routine.
– The feature f is a non-fresh once routine.
– The feature f is an attribute.

For each of these cases, there is one inference rule. Each inference rule covers one variant of the `apply` operation. The discussion continues with the most involved case: the feature f is a non-once routine or a fresh once routine.

The condition of the inference rule states that each processor can only apply a feature on one of its own objects. The condition also states the p must not have passed its locks. This part of the condition is always given because p waits whenever it passes its locks. In a first step, the operation defines an updated state σ' to set f's once status to non-fresh, in case f is a once routine. The operation does this before deep importing the actual arguments to avoid the following contradiction.

Clarification 4 (When to change the status of a fresh once routine). Assume f is either a once procedure or a non-separate once routine. The feature f was fresh at the beginning of the `apply` operation. Assume that the caller passed an expanded actual argument that is handled by a processor $g \neq p$. Therefore p has to deep import the actual argument. Assume furthermore that the class type of the actual argument has the once routine f and that f is non-fresh on g. If the operation would deep import before setting f as non-fresh on p, then the deep import operation would take over the once status of f from processor g to processor p. But then the `apply` operation on p would not make much sense anymore because f would now be non-fresh on p. If the operation sets f as non-fresh at the beginning of the `apply` operation, then the deep import operation does not take over the once status from g because f is already non-fresh on p. □

The operation defines an updated state σ'' in which the locks are passed from q to p and in which there is a new environment with the actual arguments $(r_1, \ldots, r_n)$. The call to the *push_env_with_feature* feature takes care of copying and deep importing actual arguments of expanded type. The caller processor q can also pass an empty tuple $(\{\}, \{\})$ which simply means that q did not pass any locks.

Application Operation – Non-Once Routine or Fresh Once Routine

$f \in \textbf{ROUTINE} \wedge f.is\_once \rightarrow \sigma.is\_fresh(p, f)$
$\sigma.handler(r_0) = p$
$\neg\sigma.locks\_passed(p)$

$$\sigma' \overset{def}{=} \begin{cases} \sigma.set\_once\_func\_not\_fresh(p, f, void) & if\, f \in \textbf{FUNCTION} \wedge f.is\_once \\ \sigma.set\_once\_proc\_not\_fresh(p, f) & if\, f \in \textbf{PROCEDURE} \wedge f.is\_once \\ \sigma & otherwise \end{cases}$$

$\sigma'' \overset{def}{=} \sigma'.pass\_locks(q, p, \bar{l}).push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n))$

$\overline{g}_{required\_locks} \overset{def}{=} \{p\} \cup$
$\quad \{x \in \textbf{PROC} \mid \exists i \in \{1, \ldots, n\}, g, c\colon \Gamma \vdash f.formals(i) : (!, g, c) \wedge c.is\_ref \wedge x = \sigma''.handler(r_i)\}$

$\overline{g}_{required\_cs\_locks} \overset{def}{=}$
$\quad \{x \in \overline{g}_{required\_locks} \mid x = p \vee (x \neq p \wedge (\sigma''.rq\_locks(x).has(p) \vee \sigma''.cs\_locks(x).has(p)))\}$

$\overline{g}_{required\_rq\_locks} \overset{def}{=} \overline{g}_{required\_locks} \setminus \overline{g}_{required\_cs\_locks}$

$\overline{g}_{missing\_rq\_locks} \overset{def}{=} \{x \in \overline{g}_{required\_rq\_locks} \mid \neg\sigma''.rq\_locks(p).has(x)\}$

$\forall x \in \overline{g}_{required\_cs\_locks} \colon \sigma''.cs\_locks(p).has(x)$

a_{inv} is fresh $\wedge\, a'$ is fresh

```
$\Gamma \vdash \langle p :: $ apply$(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l}); s_p, \sigma\rangle \rightarrow$
    $\langle p ::$ check_pre_and_lock_rqs$(\overline{g}_{missing\_rq\_locks}, f)$;
        provided $f \in \textbf{FUNCTION} \wedge f.is\_once$ then
            $f.body$
                [$result := y$; read(Result, $a_r$); set_not_fresh($f, a_r.data$) where $a_r$ is fresh/
                    $result := y$]
                [create $result.y$; read(Result, $a_r$); set_not_fresh($f, a_r.data$) where $a_r$ is fresh/
                    create $result.y$]
        else
            $f.body$
        end;
        check_post_and_unlock_rqs$(\overline{g}_{missing\_rq\_locks}, f)$;
        provided $f.class\_type.inv\_exists \wedge f.exported$ then
            eval$(a_{inv}, f.class\_type.inv)$; wait$(a_{inv})$
        else
            nop
        end;
        provided $f \in \textbf{FUNCTION}$ then
            read(Result, $a'$); return$(a, a'.data, q)$
        else
            return$(a, q)$
        end;
        $s_p, \sigma''\rangle$
```

In the next step, the operation synchronizes. For each target expressions in the body of f, the operation can get the controlling entity. Each of these controlling entities is mapped to an object and each of these objects is handled by a processor. For each of these processors the operation must either get a request queue lock or a call stack lock. There are three types of calls: non-separate calls, separate calls, and separate callback. Non-separate calls and separate callbacks require a call stack lock. Separate calls require a request queue lock. This leads to two sets of required locks: one set with required request queue locks and another set with required call stack locks. The set of required call stack locks is composed of p that will lead to a non-separate call and all the processors that will lead to separate callbacks. The set of required request queue locks is composed of the processors that will lead to separate calls. The operation defines two sets for these two categories: $\overline{g}_{required\_cs\_locks}$ and $\overline{g}_{required\_rq\_locks}$.

Each processor initially has its own call stack lock as its obtained call stack lock. This call stack never gets unlocked. This means that other call stack locks cannot be obtained; they must be retrieved through lock passing. The condition of the inference rule expresses this: $\forall x \in \overline{g}_{required\_cs\_locks} : \sigma''.cs\_locks(p).has(x)$. The operation can be assured that p did not pass its own call stack lock because otherwise p would be waiting. The remaining required call stack locks are the ones for the processors that will lead to separate callbacks. Note that the lock passing conditions are not sufficient to guarantee that the call stack locks for separate callbacks are always available.

As for the request queue locks, the operation calculates $\overline{g}_{missing\_rq\_locks}$ as the required request queue locks minus the already owned request queue locks. The already owned request queue locks are the previously obtained request queue locks and the retrieved request queue locks. In the synchronization step, the operation must obtain the difference. If this is not possible because some of the missing request queue locks are not available, then the operation must wait. The `check_pre_and_lock_rqs` operation takes care of this; it takes $\overline{g}_{missing\_rq\_locks}$ and the feature f. Once the execution succeeds, p has the request queue locks of $\overline{g}_{missing\_rq\_locks}$ and the precondition of f holds.

The `apply` operation can be assured that each processor g, whose obtained request queue lock the operation got in the synchronization step, must be in possession of its call stack lock. If g was not in possession of its call stack lock, it must have passed its locks. This means that g is executing a feature call and still waiting for the locks to return. In order to execute the feature call, there must have been a lock on g's request queue lock so that its action queue can contain the feature call. The request queue must still be locked because g is still executing the feature call. Hence, it would not have been possible to obtain g's request queue lock. The only exception is the bootstrap processor. However this processor only plays a role in the system setup and it never passes its own call stack lock.

Once the operation got all the required locks, it can execute the body. For once functions it must update the once status whenever it writes to the result entity as part of an assignment instruction or as part of a creation instruction. For this purpose it adds a `read` operation and a `set_not_fresh` operation after each assignment instruction or creation instruction. For each assignment instruction or creation instruction it has to use a fresh channel.

After the execution of the body, the operation has to evaluate the postcondition and it has to make sure that the locked request queues get unlocked at the right time. These two steps are performed by another operation `check_post_and_unlock_rqs` that takes the missing request queue locks $\overline{g}_{missing\_rq\_locks}$ and the feature f. This operation evaluates the postcondition either synchronously or asynchronously. After the evaluation of the postcondition, the operation enqueues an `unlock` operation to each request queue in $\overline{g}_{missing\_rq\_locks}$.

SCOOP relies on the Eiffel invariant mechanism. This mechanism is described in Sec. 7.5 and Sec. 8.9.16 of the Eiffel ECMA standard [9]. On one hand, Sec. 7.5 describes the semantics of invariants: invariants must be satisfied after the execution of every exported routine and after the execution of every creation procedure. On the other hand, Sec. 8.9.16 describes the runtime monitoring of invariants: invariants get evaluated on both start and termination of a qualified call to a routine and after every call to a creation procedure. We had to decide whether to rely on the semantics of invariants or on the runtime monitoring of invariants. We decided to rely on the semantics of invariants for two reasons. First, the runtime invariant monitoring mechanism is only one possible implementation of the invariant semantics. Second, the runtime invariant monitoring mechanism relies on the notion of unqualified calls. However, for simplicity this work assumes feature calls to be in the canonical qualified form. The `apply` operation reflects this decision: the operation evaluates the invariant whenever f is exported. Note that the invariant can only contain non-separate target expressions. Hence, each call in the invariant will only require p's call stack lock.

Finally, the operation has to return the locks and it has to return the result if f is a function. The `return` operation takes care of this. It comes in a variant for queries and in a variant for commands. Both variants take the channel a and the caller processor q in order to communicate with q. The variant for queries additionally takes the value to be returned to q.

Before explaining the variants of the `apply` operation for non-fresh once routines and attributes, the discussion continues with the operations that have not been discussed in details so far, namely `check_pre_and_lock_rqs`, `check_post_and_unlock_rqs`, and `return`.

Check Precondition and Lock Request Queues Operation

```
                              $a\ is\ fresh$
$\Gamma \vdash \langle p ::$ check_pre_and_lock_rqs$(\{q_1,\dots,q_m\},f);s_p,\sigma\rangle \rightarrow$
    $\langle p ::$ lock$(\{q_1,\dots,q_m\})$;
        provided $f.pre\_exists$ then
            eval$(a,f.pre)$;
            wait$(a)$
        else
            nop
        end;
        provided $\neg f.pre\_exists \vee a.data$ then
            nop
        else
            issue$(q_1,$unlock$)$;
            $\dots$
            issue$(q_m,$unlock$)$;
            pop_obtained_rq_locks;
            check_pre_and_lock_rqs$(\{q_1,\dots,q_m\},f)$
        end;
        $s_p,\sigma\rangle$
```

The `check_pre_and_lock_rqs`$(\{q_1,\dots,q_m\},f)$ operation, executed by processor p, takes a processor set $\{q_1,\dots,q_m\}$ whose request queues must be locked on behalf of p and it takes a feature f whose precondition must be satisfied. The operation treats the precondition as a wait condition. It goes through a number of iterations. Each iteration obtains the request queue locks and then evaluates the precondition. If the precondition is satisfied, then the `check_pre_and_lock_rqs` operation finishes. Otherwise it unlocks the request queues and then starts a new iteration. If the `check_pre_and_lock_rqs` operation finishes, p can be assured that it obtained all the request queue locks and the precondition holds.

Check Postcondition and Unlock Request Queues Operation

$$\overline{q} \overset{def}{=} \{q_1, \dots, q_m\}$$

$$p \notin \overline{q}$$

$$targets(e) \overset{def}{=} \begin{cases} \{e_0\} \cup \bigcup_{i=0\dots n} targets(e_i) & if\, e = e_0.w(e_1, \dots, e_n) \\ \{\} & otherwise \end{cases}$$

$$args(e) \overset{def}{=} \begin{cases} \bigcup_{i=1\dots n} \{(e_i, w, i)\} \cup args(e_i) & if\, e = e_0.w(e_1, \dots, e_n) \\ \{\} & otherwise \end{cases}$$

$$g_0 \overset{def}{\in} \begin{cases} if \\ \quad \overline{q} \neq \{\} \wedge \\ \quad \forall x \in targets(f.post) : (\Gamma \vdash \sigma.handler(\sigma.val(p, controlling\_entity(x).name)) \in \overline{q}) \wedge \\ \quad \neg\exists (x, y, z) \in args(f.post), t, h, c : \\ \quad\quad (\Gamma \vdash x : t \wedge controlled(t) \wedge y.formals(z) : (!, h, c) \wedge c.is\_ref) \\ then \\ \quad \overline{q} \\ otherwise \\ \quad \{p\} \end{cases}$$

$$\{g_1, \dots, g_j\} \overset{def}{=} \overline{q} \setminus g_0$$

$$a\ is\ fresh$$

```
$\Gamma \vdash \langle p ::$ check_post_and_unlock_rqs$(\{q_1, \dots, q_m\}, f); s_p, \sigma\rangle \rightarrow$
    $\langle p ::$ provided $f.post\_exists \wedge g_0 \neq p$ then
            issue(
                $g_0$,
                execute_delegated(
                        eval$(a, f.post)$;wait$(a)$;
                        issue$(g_1,$unlock$)$;$\dots$;issue$(g_j,$unlock$)$
                    ,
                        $\sigma.envs(p).top, \{q_1, \dots, q_m\}$
                );
                unlock
            );
            pop_obtained_rq_locks
        else
            provided $f.post\_exists$ then
                eval$(a, f.post)$;wait$(a)$
            else
                nop
            end;
            issue$(q_1,$unlock$)$;$\dots$;issue$(q_m,$unlock$)$;
            pop_obtained_rq_locks
        end;
        $s_p, \sigma\rangle$
```

The `check_post_and_unlock_rqs` operation also takes a processor set $\{q_1,\ldots,q_m\}$ and a feature f. The processor set is the same as for the `check_pre_and_lock_rqs` operation, i.e., the set of processors whose request queues got locked in the synchronization step. The operation first determines whether the postcondition should be evaluated synchronously or asynchronously. Then the operation starts the evaluation. Finally, the operation enqueues an `unlock` operation to each request queue in $\{q_1,\ldots,q_m\}$.

Clarification 5 (Asynchronous postcondition evaluation). The postcondition can be evaluated asynchronously if every feature call in the postcondition only requires a request queue lock that was obtained in the synchronization step and if the postcondition does not involve lock passing. If the postcondition has a feature call that requires a lock different from the obtained request queue locks, then p cannot delegate its obtained request queue lock and then continue because the required lock would be required in another feature execution context as well. Hence the postcondition must be evaluated synchronously in this case. If the postcondition involves lock passing, then one of p's lock might be necessary for the evaluation of the postcondition. Hence, p must pass its locks and cannot proceed until the postcondition is evaluated and the passed locks returned. Once again, the postcondition must be evaluated synchronously. In Nienaltowski's description of SCOOP [25] a postcondition can be evaluated asynchronously if the current processor is not involved in the postcondition evaluation. This rule permits configurations in which the evaluating processor does not have the necessary locks for the evaluation. □

If the postcondition can be evaluated asynchronously, then the operation can take one of the processors in $\{q_1,\ldots,q_m\}$. This set does not contain processor p because processor p never obtains its own request queue lock. Each processor in this set is exclusively available in the current execution context and can thus be used to evaluate the postcondition asynchronously. The `check_post_and_unlock_rqs` operation defines g_0 to be the evaluating processor according to the rule just presented. It also defines $\{g_1,\ldots,g_j\}$ to be the set $\{q_1,\ldots,q_m\}$ minus the request queue lock of g_0. If p is the evaluating processor, then this set is the same as $\{q_1,\ldots,q_m\}$. As a result of these definitions, the postcondition can be evaluated asynchronously if $g_0 \neq p$. Otherwise, the postcondition must be evaluated synchronously.

In the synchronous case, processor p evaluates the postcondition, enqueues `unlock` operations to each request queue in $\{q_1,\ldots,q_m\}$, and then removes the corresponding locks from its stack of obtained request queue locks. The `unlock` operations will not proceed until the locks have been removed from p's stack of obtained request queue locks. In the asynchronous case, processor p must delegate the postcondition evaluation to processor g_0. For this purpose, p enqueues an `execute_delegated` operation to g_0. The workload involves the postcondition evaluation along with the subsequent issuing of `unlock` operations to all processor in $\{g_1,\ldots,g_j\}$. Processor g_0 unlocks its own request queue after the delegated execution. The evaluation of the postcondition on g_0 requires the environment that defines the values of the entities in the postcondition. Furthermore, the evaluation requires the request queue locks $\{q_1,\ldots,q_m\}$. These locks are sufficient because the postcondition only gets evaluated asynchronously if the evaluation only requires these locks. To satisfy these two requirements, p gives its top environment and $\{q_1,\dots,q_m\}$ to g_0. After g_0 performed the delegated execution, it can unlock its own request queue. In the meantime, processor p removes $\{q_1,\dots,q_m\}$ from its obtained request queue locks to enable g_0 to proceed with the delegated execution.

The `return` operation comes in two variants: one for queries and one for commands.

Return Operation – Query

$$\dfrac{\begin{array}{l}(\sigma',r') \overset{def}{=} \begin{cases} \mathit{if}\, r \neq \mathit{void} \wedge \sigma.\mathit{ref\_obj}(r).\mathit{class\_type}.\mathit{is\_exp} \wedge \sigma.\mathit{handler}(r) \neq q \\ \quad (\sigma_x, \sigma_x.\mathit{last\_imported\_ref}) \\ \quad\ \ \textbf{where} \\ \quad\quad \sigma_x \overset{def}{=} \sigma.\mathit{deep\_import}(q,r) \\ \mathit{otherwise} \\ \quad (\sigma, r) \end{cases} \\ \sigma'' \overset{def}{=} \sigma'.\mathit{pop\_env}(p).\mathit{revoke\_locks}(q,p)\end{array}}{\Gamma \vdash \langle p :: \texttt{return}(a,r,q); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a,r'); s_p, \sigma''\rangle}$$

Return Operation – Command

$$\dfrac{\sigma' \overset{def}{=} \sigma.\mathit{pop\_env}(p).\mathit{revoke\_locks}(q,p)}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{return}(a,q); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ \sigma.\mathit{locks\_passed}(q)\ \texttt{then notify}(a)\ \texttt{else nop end}; s_p, \sigma'\rangle\end{array}}$$

The variant for queries returns the result and the locks. The variant for commands only returns the locks. Both variants take a channel a and the caller processor q. For queries, the channel is used to return the result. For this purpose, the operation takes a reference r that points to the result. Processor q is waiting for this result on channel a. This can be seen in the `call` operation, which issues an `apply` operation and a subsequent `wait`(a) operation. The `apply` operation calls the `return` operation with the same channel a. To return the result to q, processor p executes a `result` on a. The value to be returned is not always r directly. If r points to an object of expanded class type and $q \neq p$, then q must deep import the object. In all other cases, q can take r as the return value. An explanation why the deep import operation is necessary can be found in Sec. 4.6. For commands, the channel is used to signal to q that the locks have been returned in case q passed its locks. This can be determined by looking at the state: $\sigma.\mathit{locks\_passed}(q)$. In both variants of the `return` operation, p removes the passed locks from the stacks of retrieved locks. In case q did not pass any locks, the removed entries might be the empty set. Processor p also removes its top environment because this environment is no longer needed. In case of an asynchronous postcondition evaluation, this environment temporarily gets delegated to the evaluating processor.

Until now, the discussion left out the non-fresh once routines and the attributes. Non-fresh once functions already have a result. The `apply` operation just needs to get this result from the state and return it. For non-fresh once procedures it does not even have to do this. The only obligation is the evaluation of the invariant. The evaluation of the invariant requires the call stack lock of p. This lock is given if the condition $\neg\sigma.\mathit{locks\_passed}(p)$ holds. For attributes, note that an instance of **ATTRIBUTE** is also an instance of **EXPRESSION**. Hence, the operation evaluates the attribute expression and returns the result of the evaluation. The invariant does not have to be evaluated in this case.

Application Operation – Non-Fresh Once Routine

$$\frac{\begin{array}{l} f \in \textbf{ROUTINE} \wedge f.is\_once \wedge \neg\sigma.is\_fresh(p,f) \\ \sigma.handler(r_0) = p \\ \neg\sigma.locks\_passed(p) \\ \sigma' \overset{def}{=} \sigma.pass\_locks(q,p,\bar{l}).push\_env\_with\_feature(p,f,r_0,(r_1,\dots,r_n)) \\ a\ is\ fresh \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{apply}(a,r_0,f,(r_1,\dots,r_n),q,\bar{l}); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ f.class\_type.inv\_exists \wedge f.exported\ \texttt{then} \\ \qquad\quad \texttt{eval}(a, f.class\_type.inv); \texttt{wait}(a) \\ \qquad \texttt{else} \\ \qquad\quad \texttt{nop} \\ \qquad \texttt{end}; \\ \qquad \texttt{provided}\ f \in \textbf{FUNCTION}\ \texttt{then} \\ \qquad\quad \texttt{return}(a, \sigma'.once\_result(p,f), q) \\ \qquad \texttt{else} \\ \qquad\quad \texttt{return}(a,q) \\ \qquad \texttt{end}; \\ \qquad s_p, \sigma'\rangle \end{array}}$$

Application Operation – Attribute

$$\frac{\begin{array}{l} f \in \textbf{ATTRIBUTE} \\ \sigma.handler(r_0) = p \\ \neg\sigma.locks\_passed(p) \\ \sigma' \overset{def}{=} \sigma.pass\_locks(q,p,\bar{l}).push\_env\_with\_feature(p,f,r_0,()) \\ a'\ is\ fresh \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{apply}(a,r_0,f,(),q,\bar{l}); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a', f); \\ \qquad \texttt{wait}(a'); \\ \qquad \texttt{return}(a, a'.data, q); \\ \qquad s_p, \sigma'\rangle \end{array}}$$

Creation instructions A creation instruction has the form **create** $b.f(e_1,\dots,e_n)$ where b is the target entity, f is the creation procedure, and $e_1,\dots,e_n$ are the actual arguments. Assume that b is of type (d,g,c). A processor p that executes this instruction takes the following steps:

1. Processor q creation.
   - If b is separate, i.e., $g = \top$, then create a new processor.
   - If b has an explicit processor specification, i.e., $g = \alpha$, then
     - take the processor denoted by α if it already exists.
     - create a new processor if the processor denoted by α does not exist yet.
   - If b is non-separate, i.e., $g = \bullet$, then take p.
2. Locking. Lock the request queue of q if the following conditions hold:
   - Processor p and processor q are different.
   - Processor p does not have q's request queue lock.
   - Processor q does not have p's request queue lock.

3. Object creation. Ask q to create a new instance with class type c using the creation procedure f. Attach the newly created object to b.
4. Invariant evaluation. If f is not exported, then ask q to evaluate the invariant.
5. Lock releasing. If q's request queue has been locked in the locking step, then ask q to unlock its request queue after it is done with the feature request.

There are four cases in the processor creation step:

– The entity b has a separate type.
– The entity b has an explicit processor specification and the denoted processor already exists.
– The entity b has an explicit processor specification and the denoted processor does not yet exist.
– The entity b has a non-separate type.

For each of these cases, there is one inference rule. The discussion starts with the variant where b has a separate type. In this case, the instruction defines q as a new processor and o as a new object of class type c. The reference r points to this object. First the instruction acquires a request queue lock on the new processor q so that it can issue statements on q. Next, it writes the value r into the entity b. To make a call to the creation procedure, it executes a command instruction. Once this is done, it checks whether there is an invariant to evaluate. If f is exported, then the invariant will be evaluated as part of f's feature application. In this case the instruction does nothing. However, if f is not exported, then it must issue the invariant evaluation to q. After this step, it can issue an `unlock` operation to q and remove the request queue lock from p's obtained request queue locks.

Create Instruction – Top

$$\frac{\begin{array}{l}(d,h,c) \overset{def}{=} type\_of(\Gamma,b)\\ h = \top\\ q \overset{def}{=} \sigma.new\_proc\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_proc(q).add\_obj(q,o)\\ r \overset{def}{=} \sigma'.ref(o)\\ a \textit{ is fresh}\end{array}}{\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1,\ldots,e_n);s_p,\sigma\rangle \rightarrow}$$

```
⟨$p$ :: lock($\{q\}$);
     write($b.name, r$);
     $b.f(e_1,\ldots,e_n)$;
     provided $\neg f.class\_type.inv\_exists \vee f.exported$ then
        nop
     else
        issue($q$, eval($a, f.class\_type.inv$); wait($a$))
     end;
     issue($q$, unlock);
     pop_obtained_rq_locks;
     $s_p \mid q$ :: nop, $\sigma'$⟩
```

The following discussion looks at the two variants for the cases where b has an explicit processor specification. There are two forms of explicit processor specifications: unqualified and qualified. An unqualified explicit processor specification, i.e., $< x >$, is based on a processor attribute x with an attached type. The processor denoted by this explicit processor specification is the processor stored in x. A qualified explicit processor specification, i.e., $< y.handler >$, is based on a non-writable entity y of attached type. The processor denoted by this explicit processor specification is the same processor as the one handling the object referenced by y. A qualified explicit processor specification always denotes an existing processor because this specification is based on an attached entity. This means that there is already an object attached to this entity and thus its handler must exist. This insight helps to write the conditions for the two inference rule variants.

Create Instruction – Existing Explicit Processor

$$(d,h,c) \overset{def}{=} type\_of(\Gamma,b)$$
$$h = < x > \vee h = < y.handler >$$
$$q \overset{def}{=} \begin{cases} \sigma.val(p,x) & if\, t = (d, < x >, c) \\ \sigma.handler(\sigma.val(p,y)) & if\, t = (d, < y.handler >, c) \end{cases}$$
$$\sigma.procs.has(q)$$
$$\overline{g}_{required\_cs\_locks} \overset{def}{=} \begin{cases} \{q\} & if\, q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \{\} & otherwise \end{cases}$$
$$\forall x \in \overline{g}_{required\_cs\_locks} : \neg\sigma.locks\_passed(p) \wedge \sigma.cs\_locks(p).has(x)$$
$$o \overset{def}{=} \sigma.new\_obj(c)$$
$$\sigma' \overset{def}{=} \sigma.add\_obj(q,o)$$
$$r \overset{def}{=} \sigma'.ref(o)$$
$$a \text{ is } fresh$$

```
$\Gamma \vdash \langle p ::$ create $b.f(e_1,\dots,e_n); s_p, \sigma\rangle \rightarrow$
    $\langle p ::$ provided $q \neq p \wedge \neg\sigma'.rq\_locks(p).has(q) \wedge \neg\sigma'.rq\_locks(q).has(p)$ then
            lock($\{q\}$)
        else
            nop
        end;
        write($b.name, r$);
        $b.f(e_1,\dots,e_n)$;
        provided $\neg f.class\_type.inv\_exists \vee f.exported$ then
            nop
        else
            issue($q$, eval($a, f.class\_type.inv$); wait($a$))
        end;
        provided $q \neq p \wedge \neg\sigma'.rq\_locks(p).has(q) \wedge \neg\sigma'.rq\_locks(q).has(p)$ then
            issue($q$, unlock);
            pop_obtained_rq_locks
        else
            nop
        end;
        $s_p, \sigma'\rangle$
```

The variant that handles existing processors states that the specified processor must exist. To check this, one must consider both the qualified and the unqualified possibility.

For the qualified option, one can simply lookup the value of the attribute x. For the unqualified option, one first looks up the value of the entity y and then determines the handler of the referenced object. In either case, the result q is either the denoted processor or the void value. One then checks whether q is in the set of processors of our system. The overall idea of this inference rule is the same as in the case where b has a separate type. The difference is in the processor creation, locking, and lock releasing steps. Instead of creating a new processor, the instruction takes the existing processor q. If $q = p$, then the call to the creation procedure will be a non-separate call. In this case, the instruction requires p's call stack lock. This lock is given because otherwise p would be waiting. If $p \neq q$ and q has a lock on p, then the call to the creation procedure will be a separate callback. In this case, the instruction requires q's call stack lock. This is expressed in the condition with the help of the set $\overline{g}_{required\_cs\_locks}$. If $p \neq q$ and q does not have p's request queue lock, then the call to the creation procedure will be a separate call. In this case, the instruction must obtain q's request queue lock, provided it does not already have this lock. Only when it obtained q's request queue lock, does the instruction have to issue an `unlock` operation and remove q from p's stack of obtained request queue locks.

Create Instruction – Non-Existing Explicit Processor

$$\frac{\begin{array}{l} (d,h,c) \overset{def}{=} type\_of(\Gamma,b) \\ h = <x> \\ \neg\sigma.procs.has(\sigma.val(p,x)) \\ q \overset{def}{=} \sigma.new\_proc \\ o \overset{def}{=} \sigma.new\_obj(c) \\ \sigma' \overset{def}{=} \sigma.add\_proc(q).add\_obj(q,o) \\ r \overset{def}{=} \sigma'.ref(o) \\ a\ is\ fresh \end{array}}{\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1,\ldots,e_n);s_p,\sigma\rangle \rightarrow}$$

```
⟨p :: write($x.name, q$);
    lock($\{q\}$);
    write($b.name, r$);
    $b.f(e_1,\ldots,e_n)$;
    provided $\neg f.class\_type.inv\_exists \vee f.exported$ then
        nop
    else
        issue($q$, eval($a, f.class\_type.inv$); wait($a$))
    end;
    issue($q$, unlock);
    pop_obtained_rq_locks;
    $s_p \mid q$ :: nop, $\sigma'$⟩
```

For the variant that handles non-existing processors, one has to verify that the specified processor does not exist. To do so, one considers only unqualified processor specifications. In this case, the instruction creates a new processor q with a new object o and reference r. The steps in this variant are similar to those in the variant where b has a separate type. However, the instruction has to set the value of the processor attribute x to the newly created processor. This ensures that the denoted processor will be found to exist in the future.

Lastly, there is a variant for the case where b has a non-separate type. In this case, the instruction creates the object on p. Processor creation, locking, and lock releasing is not necessary. The required call stack lock on p is given because otherwise p would be waiting.

Create Instruction – Non-Separate

$$\frac{\begin{array}{l}(d,h,c) \overset{def}{=} type\_of(\Gamma,b)\\ h = \bullet\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_obj(p,o)\\ r \overset{def}{=} \sigma'.ref(o)\\ a \text{ is } fresh\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \mathbf{create}\ b.f(e_1,\ldots,e_n);s_p,\sigma\rangle \rightarrow\\ \quad \langle p :: \texttt{write}(b.name,r);\\ \qquad b.f(e_1,\ldots,e_n);\\ \qquad \texttt{provided}\ \neg f.class\_type.inv\_exists \vee f.exported\ \texttt{then}\\ \qquad\quad \texttt{nop}\\ \qquad \texttt{else}\\ \qquad\quad \texttt{eval}(a,f.class\_type.inv);\texttt{wait}(a)\\ \qquad \texttt{end};\\ \qquad s_p,\sigma'\rangle\end{array}}$$

Flow control instructions The **if** e **then** s_t **else** s_f **end** instruction executes s_t if the expression e evaluates to true. Otherwise the instruction executes s_f. There is one inference rule for this instruction. In a first step, the instruction evaluates the expression e using a fresh channel a and then waits for a notification on a. In a second step, it uses the `provided` operation to either execute s_t or s_f, depending on the value of the expression.

If Instruction

$$\frac{a \text{ is } fresh}{\begin{array}{l}\Gamma \vdash \langle p :: \mathbf{if}\ e\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end};s_p,\sigma\rangle \rightarrow\\ \quad \langle p :: \texttt{eval}(a,e);\\ \qquad \texttt{wait}(a);\\ \qquad \texttt{provided}\ a.data\ \texttt{then}\\ \qquad\quad s_t\\ \qquad \texttt{else}\\ \qquad\quad s_f\\ \qquad \texttt{end};\\ \qquad s_p,\sigma\rangle\end{array}}$$

The **until** e **loop** s_l **end** instruction executes a sequence of s_l instructions until the expression e evaluates to true. If e is true initially, then s_l never gets executed. There is one inference rule for this instruction. First, the instruction evaluates e using a fresh channel a. Then it waits for a notification on a. Next, it uses the `provided` operation to check whether e evaluates to true or false. If e is true, then it is done. Otherwise, it executes s_l followed by another **until** e **loop** s_t **end** operation.

Loop Instruction

$$\frac{a \textit{ is fresh}}{\Gamma \vdash \langle p :: \textbf{until } e \textbf{ loop } s_l \textbf{ end}; s_p, \sigma\rangle \rightarrow}$$

```
⟨$p$ :: eval($a, e$);
     wait($a$);
     provided $a.data$ then
        nop
     else
        $s_l$; until $e$ loop $s_l$ end
     end;
     $s_p, \sigma$⟩
```

Assignment instructions An assignment instruction $b := e$ assigns the value of the expression e to the entity b. The instruction first evaluates the expression e and then waits for a notification on a fresh channel a. Once it gets this notification, it uses the `write` operation to set the value to the entity b.

Assignment

$$\frac{a \textit{ is fresh}}{\Gamma \vdash \langle p :: b := e; s_p, \sigma\rangle \rightarrow \langle p :: \texttt{eval}(a, e); \texttt{wait}(a); \texttt{write}(b.name, a.data); s_p, \sigma\rangle}$$

5.5 Termination

The system terminates when it reaches a configuration where all action queues are empty, i.e., when there is no more work to do.

6 Conclusion

In this paper we have presented a formal specification of the SCOOP model, based on operational semantics. We have demonstrated that this level of rigor is necessary if the specification is to be used as a guideline for an implementation. In particular, we were able to clarify a number of omissions and ambiguities in the available informal specification, which had gone undetected in other formalizations:

– Are processor locks fine-grained enough? We require request queue locks and call stack locks.
– Which locks must be passed? Which locks can be passed? We pass all the locks we actually have. We pass these locks both for normal lock passing and for separate callbacks.
– How do we move object structures from one processor to another processor without violating the invariant? The deep import operation must be used.
– When do we set the status of a fresh once routine to non-fresh? The status of the once routine must be set to non-fresh before deep importing.

– When can a postcondition be evaluated asynchronously? The postcondition can be evaluated asynchronously if every feature call in the postcondition only requires a lock that was obtained in the synchronization step and if the postcondition does not involve lock passing.

Because of the complexity of the SCOOP model, our resulting specification is large, the management of which is a challenge for a fully formal development. To address this problem, we used abstract data types and a notation with an object-oriented flavor, which made the specification more readable and more easily extendable, without sacrificing any of the rigors of operational semantics. Furthermore, we introduced a distinction between two kinds of statements, namely instructions (user syntax) and operations (run-time syntax). This made it possible to treat within one inference system both the actual language elements and the implementation details of the runtime system, and to distinguish clearly between them.

The main application of this work is to guide the implementation of the SCOOP model. This has led to a successful implementation of SCOOP on top of the Eiffel language, which supersedes the previous prototype implementation and is publicly available [31]. The SCOOP model can however be implemented on top of any object-oriented language (support for contracts, as offered by Java or Spec#, is beneficial), and our work also facilitates such future implementation efforts. In the case of Java, first steps towards such an implementation have been taken [33], which could certainly be supported by our work.

A number of other applications of our semantics can be envisioned. First, the semantics can be used to prove correct various properties of the model which have so far only been postulated, such as absence of object-level data races and type safety (absence of traitors). In light of the complexity of the full model, these properties are no longer obvious. For example, as processor locks serve as an abstraction only, it must be shown that locks are not misused in situations such as separate callbacks, which involve call stack locks. Second, our operational semantics can also be used to prove correct an axiomatic semantics for the SCOOP model, which is planned for future work. In the case of sequential Eiffel, a similar development is documented in [26]. Third, we feel our semantics is detailed enough that its rules can directly be implemented as an interpreter for SCOOP programs. Such an interpreter could serve as a true reference implementation, which could in turn be used for conformance checking of real implementations.

Acknowledgments. We thank Stephan van Staden for interesting discussions on the lock model. This work is part of the SCOOP project at ETH Zurich, which has benefited from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award) and ETH (ETHIIRA).

References

1. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: A compositional operational semantics for JavaMT. Lecture Notes in Computer Science 2772, 290–303 (2003)
2. Allen, E., Chase, D., Luchangco, V., Maessen, J.W., Jr., G.L.S.: Object-oriented units of measurement. In: Conference on Object Oriented Programming Systems Languages and Applications. pp. 384–403 (2004)

3. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. ACM Transactions on Programming Languages and Systems 26(5), 269–804 (2004)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. ACM SIGPLAN Notices 30(8), 207–216 (1995)
5. Brooke, P.J., Paige, R.F., Jacob, J.L.: A CSP model of Eiffel's SCOOP. Formal Aspects of Computing 19(4), 487–512 (2007)
6. Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: An event-based structural operational semantics of multi-threaded Java. Lecture Notes in Computer Science 1523, 157–200 (1999)
7. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Conference on Object Oriented Programming Systems Languages and Applications. pp. 519–538 (2005)
8. Coffman, E.G., Elphick, M.J., Shoshani, A.: System deadlocks. ACM Computing Surveys 3(2), 67–78 (1971)
9. ECMA: ECMA-367 Eiffel: Analysis, design and programming language 2nd edition. Tech. rep., ECMA International (2006)
10. Ericsson Erlang website: http://www.erlang.org/ (2011)
11. Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 372–385 (1996)
12. Gelernter, D., Carriero, N., Chandran, S., Chang, S.: Parallel programming in Linda. In: International Conference on Parallel Processing. pp. 255–263 (1985)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
14. International Organization for Standardization: ISO/IEC 8652:1995 Ada. Tech. rep., International Organization for Standardization (1995)
15. Joyner, M., Chamberlain, B.L., Deitz, S.J.: Iterators in Chapel. In: International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium (2006)
16. Khoshafian, S., Copeland, G.P.: Object identity. In: Conference on Object Oriented Programming Systems Languages and Applications. pp. 406–416 (1986)
17. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
18. Liskov, B., Zilles, S.: Programming with abstract data types. ACM SIGPLAN Notices 9(4), 50–59 (1974)
19. Lochbihler, A.: Type safe nondeterminism – A formal semantics of Java threads. In: International Workshop on Foundations of Object-Oriented Languages (2008)
20. Meyer, B.: A three-level approach to data structure description, and notational framework. In: ACM-NBS Workshop on Data Abstraction, Databases and Conceptual Modelling. pp. 164–166 (1981)
21. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2nd edn. (1997)
22. Microsoft Axum website: http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx/ (2011)
23. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press (1999)
24. Morandi, B., Nanz, S., Meyer, B.: A comprehensive operational semantics of the SCOOP programming model. http://arxiv.org/abs/1101.1038v1 (2011)
25. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich (2007)
26. Nordio, M., Calcagno, C., Müller, P., Meyer, B.: A sound and complete program logic for Eiffel. In: Oriol, M., Meyer, B. (eds.) TOOLS-EUROPE. Lecture Notes in Business and Information Processing, vol. 33 (2009)
27. Odersky, M.: The Scala language specification version 2.8. Tech. rep., Swiss Federal Institute of Technology Lausanne (2010)

28. Ostroff, J.S., Torshizi, F.A., Huang, H.F., Schoeller, B.: Beyond contracts for concurrency. Formal Aspects of Computing 21(4), 319–346 (2008)
29. Plotkin, G.D.: A structural approach to operational semantics. The Journal of Logic and Algebraic Programming 60–61, 17–139 (2004)
30. Schmidt, H.W., Chen, J.: Reasoning about concurrent objects. Asia-Pacific Software Engineering Conference 0, 86 (1995)
31. SCOOP website: http://scoop.origo.ethz.ch/ (2011)
32. SGS-THOMSON Microelectronics Limited: occam 2.1 reference manual. Tech. rep., SGS-THOMSON Microelectronics Limited (1995)
33. Torshizi, F., Ostroff, J.S., Paige, R.F., Chechik, M.: The SCOOP concurrency model in Java-like languages. In: Communicating Process Architectures. pp. 155–178. IOS (2009)