Falcon 1.0 Organic virtual machine specification

The Falcon Committee

Giancarlo Niccolai

Overview

Status of this document

This document is an evolving draft (0.4).

All the basic aspects of the FOM are illustrated in this version of the document.

Notice that actual implementation of the Falcon programming language presents several extensions, which are not indicated in this specification as they fall under the category of implementation-specific behavior.

Scope of this document

This document is a technical description of the specifications of the Falcon Organic Virtual Machine (FOM).

The FOM is a Virtual Machine explicitly designed to run Falcon 1.0 language programs, but it can be generally used as any virtual processor to execute a wide range of code, that go beyond the specific implementation of a single programming language.

Usually, a virtual machine specification includes a finite list of atomic operations that the virtual machine is able to perform, but this is unpractical in the case of the FOM. In fact, the FOM is dynamically analyzing and executing syntactic trees, each node of which has a precise symbolic meaning in the programming language to which it belongs. Instructions for the virtual machine are provided from the outside, according to some rules that are hereby described, and hence are not strictly part of the general specification.

For this reason, the description of the “instructions” that the FOM executes are not in the scope of this document. A brief list of basic general purpose instruction is given in an appendix for reference and example of the instruction implementation.

A separate document describing the instructions that are provided to the virtual machine by each language shall be separately provided.

Premise about the environment of the FOM

The Falcon Organic Machine is defined as an entity living in an *environment*. Be it implemented as a software code or wired in an hardware structure, the FOM requires a supporting environment in which it can operate.

In this regards, this specifications differ from usual virtual machine specifications, where the virtual machine is generally defined as a closed entity or a whole theoretical computation unit (a CPU that may be a stand-alone system).

This specifications include the minimal requirements for the FOM supporting environment and the protocol through which the environment and the FOM exchange information.

Concretely, the environment might be one of the following:

- A complex software using the FOM for any purpose. This goes under the name of “embedding”.
- An operating system specifically designed to run FOM as its sole kind of process, or as one of the possible kind of processes.

- An hardware system designed to start and manage at least one FOM.

Graphic and linguistic conventions

This specification adopts the following conventions:

- Every paragraph in the subsequent characters, up to the appendix, is to be considered a mandatory norm, unless prefixed with the words *rationale* or *notice.*
- Paragraphs prefixed with the word *rationale* explain the reason why some norm is given in the previous paragraph(s). Rationales are given especially when the reason is not of immediate understanding, or when a wider comprehension of the reason behind a given norm is considered relevant for its application.
- Paragraphs prefixed with the word *notice* give some exemplification, explicit warning or context information that concerns the previous paragraph(s).
- Minor considerations relative to norm explanation are given as footnotes, when they aren't important enough to be written in paragraphs prefixed with the *notice* indication.
- The word **“must”** written in bold characters indicates a mandatory behavior that is required for the implementation to complain this specification.
- The word **“must not”** written in bold characters indicates a behavior that is explicitly forbidden for the implementation to complain this specification.
- The word “**can**” written in bold characters indicates an optional behavior.
- The locution “the implementation is free to” indicates a behavior that is not part of the specification, but that is explicitly excluded from the scope of this specification now and in future.
- The locution **“if** … **then** ...” written in bold characters indicates a mandatory norm which must be followed only in the case that some pre-condition apply.

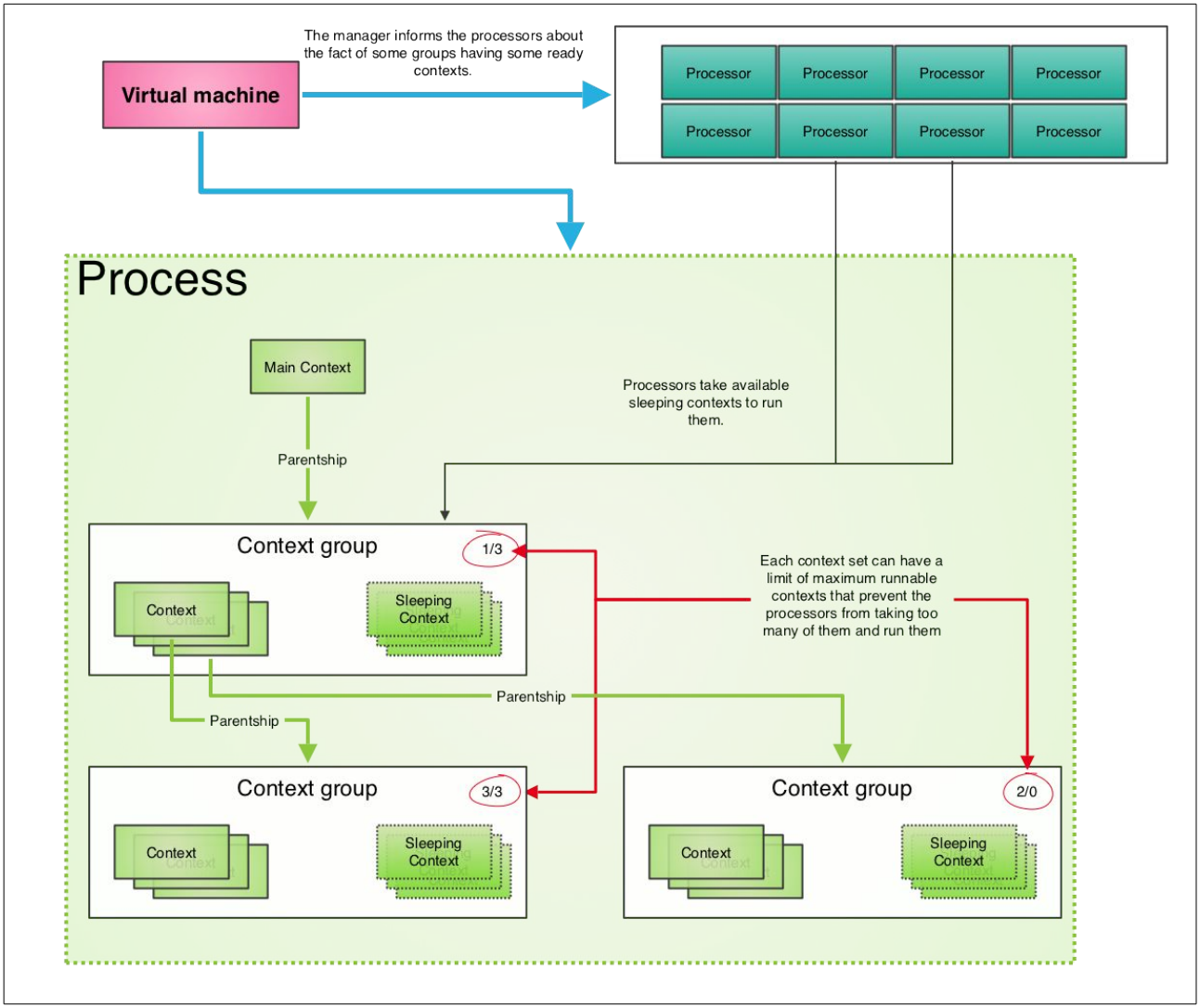

Figure 1: Falcon Organic Machine overall structure

High level structure of the organic virtual machine

The FOM has knowledge of five fundamental entities that superintend the execution of code[1]:

- **Context**: A consistent unit of execution status, which encapsulates any aspect of a sequential execution.
- **Processor**: An entity that receives a context and a code to be executed and is capable to process the code making the required changes on the context.
- **Process**: A execution unit that manages the current status of execution of one or more contexts through one or more processors.
- **Context group**: A set of contexts that are related to a parent context which launches a parallel execution. The status of the group is a function of the status of each context in the group, and the parent context is notified about events that are relevant to the group as a

[1] Notice that we don't need to define the “code” entity yet. At this level, the exact structure of what the FOM is supposed to execute can still be opaque.

whole.

- **Shared resource** (or simply *shared*): a synchronization device used as a base for several synchronization operations.

The relationship between these entities is summarized in Figure 1: *Falcon Organic Machine overall structure.*

As the virtual machine is requested to execute some code, it creates a *process* which has one *main context* representing the status of the execution. The code to be executed and the *main context* are passed to one of the available *processors*, which will perform its task of executing each step of the code and applying the results on the context. If the code requires a parallel processing to be started, a new *context group* is created. The group is parented to the context in which the parallel execution is started, and it holds as many contexts as required. Again, each context in the group, together with the code that must be executed in that context, is passed to a free processor. Contexts in the group can start parallel processing, creating more subgroups that are parented to them.

Internal FOM architecture

A Falcon organic virtual machine is internally composed of the following components:

- A *time slicer*, which is a simple clock that keeps track of some events and performs some timed services for the FOM and its processors.
- A *context manager*, which has the duty of keeping track of unscheduled contexts and rescheduling them in due time or when some condition occur.
- A *garbage collector* which periodically inspects contexts for garbage and keeps track of unused memory.

Other useful but common structures that implementers will need, as sets of contexts, set of context groups, set of processors and so on are willfully left out from this specification. Their implementation and interface is left to the final implementer.

Optionally, the FOM can provide a *streamline processor*, which is a special processor that doesn't execute code asynchronously.

Process lifecycle

The process through which a FOM performs a consistent set of computations, (in generic terms, *a program*), is defined as a set of data shared between the FOM and its *environment*.

A *process* is created through one of the following operations:

- Request of execution of a given *function*[2].
- Request of execution of a given *code*.

In both cases, a Process entity is created and returned to the calling *environment*. The process presents to the environment an interface through which the environment can:

- interrupting the process (invalidating it definitively);
- wait for the process completion (either for a given time or indefinitely);
- retrieve the final computation result once the process is complete.

[2] Left undefined at this time; the concept will be explained in the chapter related to the structure of the code.

Process entities are **necessarily** reference counted, as they are shared at least between the FOM and the invoking *environment*.

If the FOM provides a *streamline processor*, then it **must** provide an interface to execute a function or a code without returning a Process entity to the *environment*, and the invocation **must** be blocking until the process executed in streamline is complete. However, the FOM itself may create or recycle a process entity for its internal usage.

Once a *process* is started, the following operations are performed:

1. The *main process* is sent to a *processor.* There can be only one *main context* running at a certain time for a given *process*[3].
2. When starting a parallel execution, the initiator *context* is suspended. A *context group* containing a number of contexts equal to the number of code entities to be executed in parallel is created and parented with the initiator.
3. When all the contexts in a group have completed their execution[4], the parent context is notified and its execution resumed; then it will be given the opportunity to retrieve the final result of the computations performed in each context.
4. In case one of the contexts in a group is terminated with error, the all the other contexts of the group are interrupted. The parent context is resumed and the error is re-thrown[5] there.
5. In case two or more contexts in a group are terminated with error before the FOM is able to stop all the contexts in the group, the error happening first in time only is notified upstream; all the other errors are silently discarded.
6. If the main context terminates with error, the process itself terminates and the error is re-thrown at the environment level[6]; a current or subsequent wait operation on the Process is bound to throw the given error.

By this rules, it can be noticed that an error happening deep in the hierarchy of running contexts can terminate a whole process if not caught at some level (which is a desirable behavior).

When the *main context* is terminated either cleanly or with error, the *process* is disengaged from the virtual machine and its reference count is decremented so that the last reference is left to the FOM user.

On the FOM user side, it is valid not to wait for a process termination, and hence release the reference prior to its completion. This means that when the reference count hits 0, the process entity must be able to cleanly release any system resource it held autonomously.

Context lifecycle

Internally to the FOM, the performing of a *process* is a cooperative and iterative operation that involves the *processors*, the *time slicer*, the *context manager* and the *garbage collector*, as indicated in Figure 2.

---

3 Obviously, the FOM must be capable of running more processes at the same time.

4 Notice that we're talking of execution of context for brevity. Actually, it's the code that's related to a context that is executed, while the context just records the execution status. In general, when we speak of execution of a context, we mean the processing of the code whose status is recorded by a certain context.

5 Concepts of FOM error and throw are still left opaque at this time.

6 How an error can be “thrown” in the environment is outside the scope of this specification.

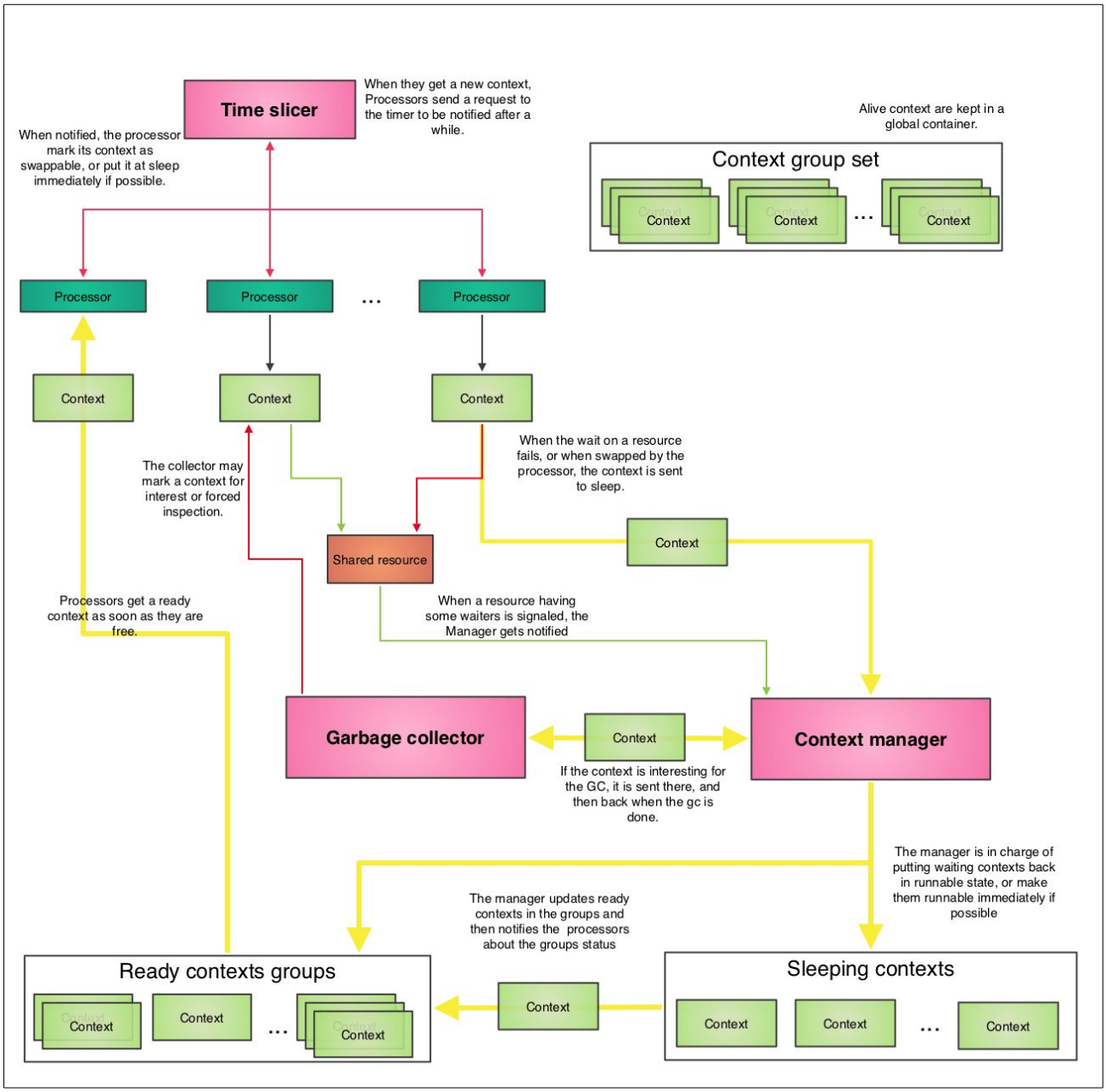

Figure 2: Context life cycle

Processors are notified about new contexts being ready to run; initially, this corresponds to the main context in a processor only, in a typical producer-consumer model where the queued "work" is a context ready to be executed[7].

Once a context is being run in a processor, three things can happen:

- The context may cleanly terminate.
- The context may terminate due to an error that was raised and not caught by any handler.
- The context might get swapped out and sent to the context manager for the following reasons:
    - The garbage collector is requiring to inspect the context memory.
    - The assigned time-slice on the processor is terminated.

[7] Again, we're using the concept of the context to indicate also the code that the context is bound to just for brevity. Strictly speaking, the context just represents the status of the code that is being run.

- A timed wait on shared resources has failed.

These events are kept in a globally visible, atomically access *context status register* (CSR) which reports what events have happened. The implementation of the register is not specified (see appendix A for advised implementation details). The CSR **can** be used in the final implementation to present other implementation-relevant events that might have happened, as long as this usage doesn't break this specification and is confined to the internal working of the implementation.

Rationale: The given states are a minimal indication of what the FOM must be prepared to handle and to communicate to the Environment, but there are other events that an implementation might be interested to audit for. The raising of soft exceptions and the request to return from partial computations are two examples. Having to check for this fact in a place different than the CSR and the CSR itself to decide to exit a computation might be cumbersome.

Context completion

As a context cleanly terminates, because the code it superintends to explicitly returns out of the topmost function, or because all the instructions are performed, it is naturally swapped out of the processor on which it was running.

If it belongs to a *context group*, the group gets updated; when all the other contexts are complete, the parent context is resumed (re-sent to the ready contexts queue), and it gets the chance to inspect the group to determine the results.

Also, if context belongs to a *context group*, and if the *context group* has a *runnable context count*, a notification is sent to the context manager.

If the context is not part of a group, this means that it is a *main context* for a process, and the *process* is thus terminated (eventually waking up physical threads engaged in the waiting methods of that process).

Context error termination

When some code raises an error and there isn't any handler suitable to intercept it through the context in which the error is thrown, the context is said to be *terminated with error*.

If the context is part of a group, all the contexts in the same group are terminated. When this happens, proper flags are set in the contexts and adequate messages are sent to the processors and to the context manager so that the contexts can get swapped out or removed from the ready queue or sleeping context set as soon as possible.

When all the contexts of the group are cleared from their status (i.e. removed from execution or wait queues), the context that is owning the group gets notified and put in the ready contexts queue. As soon as it reaches a processor, the error throw is repeated as if it was thrown by the code itself (as if it emerged by the very call that started the parallel processing), and there it can be caught or again emerge uncaught to the top-level, and from there, to the environment.

If a context terminated with error is not part of a group, this means it's a main context for a process; the process is disengaged from the FOM and dereferenced, and the error is passed to the process structure. As a wait method is invoked (or immediately if the FOM environment was already engaged in a wait). The exact interface through which the error is notified to the environment is obviously environment-specific, and hence it's not specified here[8].

If the process was already dereferenced by the *environment*, as the process is disengaged and

[8] Just as an example, on a minimal hardware it might just be an error code stored in a register with a special significance, on a O/S it might be a system signal, on an embedding program it should be a language-specific exception.

dereferenced it is destroyed, and the error is silently ignored.

Notice: there is a possibility that more than a context in a group terminates with error; this might happen because a processor is not able to honor the termination request issued after the first error soon enough to avoid the second one, or simply because an error can happen just at the same time in two different contexts. In that case, the errors generated after the first one are silently ignored and destroyed. If it's not possible to determine which error came first, one is selected at random and the others are destroyed.

Rationale: On one side, there is no guarantee that the subsequent errors are not generated by a misbehaving of the first failing context; on the other side, a context terminating with error is a critical problem that has to be specifically addressed, usually by fixing a problem in the code by the original programmers. Addressing that problem may well resolve the other concurrently generated problems; if not, they require a separate and specific intervention. Also, providing a consistent mechanism to signal and deal with multiple errors occurring at the same time seems hard and complex enough to be difficult to implement and error-prone to use. It's our judgment that the extended information that might be obtained by dealing with more uncaught errors occurring in the same group at the same time is not worth the effort that a final programmer should deploy to correctly handle this information and make good use of it.

Context swap-out

Besides the completion and termination cases, there are exactly four other reasons why a context can be swapped out from a processor:

1. The processor sends a request to the time slicer to be notified after a given time. When notified, it checks if the context is in a swappable state and if there is some context ready for execution; if both this conditions are met, the context is sent to the manager for later rescheduling.
2. The garbage collector keeps track of the allocation activity performed by each context, and when it "suspects[9]" that a context might have some garbage in need to be collected, it marks it for inspection. When the need becomes urgent, for example, because of memory shortage, it might mark the context for immediate check, and instruct the processor to swap it out ASAP.
3. The code related to the context fails to acquire any *shared* resource in a wait request.
4. The code of a context asks to be swapped out through an explicit sleep request.

In the first case, the processor contextually gets another context to be run, while in the others it peeks for ready contexts and if none is found, it simply put itself in wait for new work to be performed.

A context cannot be swapped out (it is *non-swappable*) while it holds an *acquirable shared* resource; it is said to be in a *critical section*[10]. More detail about this aspect are seen later when we describe the shared resources.

Shared resources

Shared resources are synchronization devices that are meant to allow contexts to wait for a certain event to happen, or in general, for the *program* to have reached a certain consolidated state.

[9] The precise algorithm used to determine the need for garbage collection in contexts is beyond the scope of this document.

[10] Pay attention to the fact that this term is used with various non-overlapping and even conflicting meanings in literature and in practice. We use the term critical section in the original meaning of a region of code that must be run exclusively by a single processor and that cannot be interrupted.

Shared resources **can** be exposed as entities that the environment can directly handle, by putting a manageable part of itself in wait, or polling their status, or through any mean that gives a consistent representation of a synchronization device in the given environment.

They **must** support the following basic operations:

- **signal**: indicates that the state has been reached.
- **wait**: checks if the given program state is reached, and if not, wait for that to happen up to a given time or forever. If the wait time is zero the operation is said to be a *try-wait*, and the invoker is not blocked; instead, a failure is immediately notified to the invoker. If the wait time is -1, the operation is said to be *waiting forever*, and the invoking context won't be made runnable again unless the resource is signaled.
- **clear**: remove all the signals currently posted to the shared resource. This indicates that the condition that was previously signaled is not present anymore.

The initial status of a shared resource is *not signaled*. The implementation **can** provide means to initialize the shared resource status differently.

Notice: despite the fact that the all the shared resources must implement this operations, it is not necessary for all the shared resources to expose this the **clear** operation to their final users (either the environment, the final language implementation or both).

Shared resources **must** behave accordingly with one of the following *signal consumption semantic*:

- *Consume single signal in wait*: as a wait is fulfilled, a single signal is consumed. In other words, each signal operation grants a single waiting entity to proceed. Waiting entities might be already in wait, or file a waiting operation at a later time[11].
- *Consume all signals in wait*: as a wait is fulfilled, the signaled state of the resource is reset. Any subsequent wait operation will fail unless one or more signal operations have occurred in the meanwhile[12].
- *Consume no signal in wait*: wait operations won't alter the signaled status of the resource.

If the shared resource is exposed to the *environment*, then it **must** present an interface so that it can be determined if the invoker of the wait operation was a *context* or the *environment*. In this case, when a timed wait performed by the environment fails, the environment **must** be provided with a *environment wait handler* by which it can wait and receive notifications about the resource being signaled.

Environment wait handlers can be shared across different environment entities (i.e. one per shared resource), or be uniquely created/initialize as a wait operation is performed. This detail is left to the implementation.

Shared resources **must** implement a list of contexts or *environment wait handlers* waiting for a signal to be received, called *notification queue*[13].

Rationale: The FOM doesn't necessarily share signal-wait semantics provided by the host environment. The handler bridges this difference, allowing an environment that is capable of various degrees of native multi-threading to participate to the wait-signal processes that are at work inside the FOM. Forcing the environment to wait on shared resources directly would cause the model to be defective either on the environment side or on the FOM side in various implementation

[11] In short, this semantic is the same as a classical semaphore semantic as defined by Dijkstra.

[12] This is the same semantic as the MS-Windows system self-resetting events.

[13] It might be useful to manage just wait handlers in this queue, with the wait handlers eventually referring to a FOM context they're attached to.

contexts.

The *environment handler*, **if** provided, **must** implement the following operations:

- Try-wait on the attached shared resource.

The *environment wait handler*, **if** provided, **can** implement the following optional operations:

- Timed wait
- Infinte wait
- Cancel wait

Wait operations on *environment wait handlers* must provide at least one of the following result:

- Signal received / wait succeed.
- Host resource destroyed.
- Program terminated.
- Wait canceled.

Rationale: Minimally, the environment must be able to poll over multiple handlers to know if it acquired any of the resources it was waiting for. On environments that are able to give some control on the multi-threading O/S layer, it's ideally better to wait for resources to be acquired through proper O/S wait devices. A couple of events that aren't interesting for programs run by the FOM are actually important for the environment; for instance, if the program is terminated, or if a shared resource goes out of scope and garbage collected, the host must be notified or it would hang in case it's waiting on that resource.

Notice: Environment wait handlers share part of the behavior of a context. They can be stored in the shared resource wait list and handled by the context manager for timed wake up. This might suggest a common base class in an OOP implementation, or at least a common entity used in the environment wait handler and context structure which presents a consistent interface to shared resources and to the context manager.

Shared resources **must** also expose the following operations exclusively to the FOM engine:

- **Enqueue**: stores a *context* or an *environment handler* in the *notification queue*, disregarding the current signal status.
- **Signal queue:** while signals can be consumed accordingly with the signal consumption semantic of the resource, dequeue entities stored in the notification queue and notifies them about they having consumed a signal on this resource. This operation **can** return a list of notified entities for the implementation to use it.

Environment handlers scope

If the implementation provides an environment handler, **then** the receiving environment **must** ensure that the receiving code stays valid for the whole of the duration of an environment wait request. This means that the environment is not allowed to destroy an environment handler while engaged in a wait.

Wait operation process

As a wait operation is performed, the following operations **must** be performed atomically (seen as an indivisible operation from outside):

- Check the notification queue; if it's not empty, the wait fails.
- Check the signal status or count; if the resource is in *not signaled status*, the wait fails.
- If the two checks above succeed, change the signal count or status accordingly with the *signal consumption semantic,* and report success.
- If any of the check fails, and if the wait request is not a try-wait operation, add the calling entity to the notification queue.

Rationale: entities waiting for the resource to be signaled will add themselves to that list, while wake ups are exclusively performed by the *context manager*; so the queue operation can be concurrent from multiple system threads, while the dequeue operation will be performed by the *manager* thread only.

Context waits on multiple resources

A context or the environment can wait on an undefined count of shared resources, on a hierarchical preference order. The multiple resource wait operation is an operation which is not strictly referred to the shared resource entities. From an object oriented programming point of view, it's an operation of the context, or in case of the environment, it's either a function or requires a waiter object which has this sole purpose. How multiple resource waits are exposed to the environment, if at all, is left to the implementation. This specification describes how to implement multiple resource waits performed by a FOM context.

A try-wait operation on multiple shared resources is performed as follows:

- The context tries to wait on the first resource in the list.
- In case of success, the resource is returned and the operation ends successfully.
- In case of failure, the next resource is tried.
- If none of the try-wait operations on the required resources was successful, the failure is reported.

A timed operation on multiple shared resources is performed as follows:

- The context tries to wait on the first resource in the list.
- In case of success, the resource is returned and the operation ends successfully.
- In case of failure, the next resource is tried.
- If none of the try-wait operations on the required resources was successful, the following operations are performed, in any order:
    - The context status register is set to *wait failed*.
    - The context invokes the **enqueue** operation on all the resources it was waiting for.

Also, in case of failure the context **must** subsequently be de-scheduled from the processor it's running on and sent to the context manager as soon as possible, and anyhow before performing any other operation in the processor. The detail of how and when to perform this operations is left to the implementation.

Rationale: FOM implements a *fair* signal consumption policy. The first entity entering the wait queue of a shared resource will be the first getting a signal. However, the multiple wait operation itself is not granted to be fair until the operation exit. Concurrent multiple waits from different contexts are allowed to be non fair up to the termination of the shortest wait operation; then, the

enqueue operation grants that, even if signaled, a third party arbiter will have to fairly resolve the race, as wait will always fail if the shared resource notification queue is not empty. In short, the FOM fairness policy can be specified as fairness for the first waiter able to exit the wait operation, or *first-wait-complete* fairness[14].

Shared resources with critical section semantic

Some shared resources can have a so called *critical section semantic*, or CSS. This is simply a flag that indicates that the context(s) that have performed a successful wait on that resource will be engaged in an important operation for the time being.

Once a wait on a *CSS shared resource* is successful, that resource is set as the *critical section resource* (CSR) for the context, and the context is said to have *entered a critical section*. When a context is in a critical section, the following rules **must** be followed:

1. The *context* **cannot** be implicitly swapped out by its *processor*.
2. When the code in the *context* performs an operation which results in the suspension or termination of the context, the CSR **must** be signaled. Such operations are called *blocking or terminating*, or *BT-ops*.

Notice: the fact that a shared resource has a CSS is not a guarantee that the critical section is entered by just a context at a time, in case more contexts are executing the same code. For that to happen, it is necessary that the CSS resource has also a clear-all-signals wait semantic. Providing some sophisticate lock-unlock schemes, like, for instance, reentrant mutexes, is up to the implementation and not required.

The list of *blocking or terminating operations* is as follows:

- context completion;
- context termination by uncaught error;
- wait on resources (even if non-blocking, that is, even if immediately successful or with a 0 timeout);
- sleep requests (even if unscheduled with 0 sleep time).

Signaling the active CSR of a context, either after a *BT-op* or via an explicit signal operation, causes the context to be not anymore considered critical; this is said, to *exit the critical section*. As the critical section is exited, provided the context is not terminated, the host processor **must** check if some swap-out condition were set but held back:

- time-slice elapsing (with other runnable contexts queued in wait);
- garbage collector explicit swap-out requests

In this case, the context **must** be immediately swapped out as it exits the CS.

Rationale: The *BT-op* causing the exit might contextually lead to the acquisition of another CSS shared resource, or the CSR signaling can be immediately followed by a CSS resource acquisition. In this two cases, a request for GC inspection or time-slice swap out might be ignored, possibly for a long time or indefinitely, causing the implicit swap-out feature to starve.

Message queues

Message queues are synchronization devices that puts a producer and many consumers in a special

[14] The wait operation actually terminates not when the wait function exits, but when the context manager is able to receive the waiting entity and inspect its status.

relationship. They are meant to repeat the same message to all the consumers that declared their availability to read them now or at an undefined time in future. They present an interface that is partially compatible with the operations exposed by the shared resources, so that they can be subject to be part of multiple waits.

Notice: In a object oriented programming pattern, the message queue and the shared resources might be derived from a common *waitable* interface or base class. The operations indicated in this specification having the same names for the message queues and the shared resources are the parts that should be made common.

The behavior difference with a shared resource is in the fact that signals are not globally consumed by waiters, but they are specifically consumed by all and only the waiters having declared a subscription before the posting is performed.

The message queue can be visualized as a post office where waiters can subscribe; as they do, a FIFO postbox is created for them, and as a message is posted to the queue, it is replicated and stored in each postbox that was existing at that moment. The subscribers can then wait for their postbox to contain one or more messages, and read them at any later moment in the same order they were sent.

For example, suppose contexts *x* and *y* subscribe to the same message queue. If then the messages A and B are posted, both x and y will read A and B, independently from the action of the other reader. For instance, *x* might read A, then *y* A and B, and finally *x* will read B.

A *message queue* exposes the following operations:

- **subscribe**: Indicates that the calling context or environment wants to receive messages posted to this queue.
- **unsubscribe:** remove subscriptions previously performed by the caller (no-op if not previously subscribed).
- **post**: Sends a message to the queue. If the queue hasn't any active subscription, this is a no-op. The implementation **can** return a special value to indicate if the posting was actually performed or not.
- **wait**: waits for a message to be readable in the queue by a certain entity (context or environment). If the caller didn't previously subscribe the queue, it is also contextually and atomically subscribed.
- **read**: dequeues a previously posted message, if present, returning it to the caller. Returns nothing if there isn't any message for the reader.

Similarly to the shared resources, the implementation **can** provide a mean for the environment to participate in the message posting and receiving. **If** it does, then it **must** provide *environment handlers* compatible with those provided through the shared resource interface.

Message queues **must** also expose the following operations exclusively to the FOM engine:

- **Enqueue**: stores a *context* or an *environment handler* in the *notification queue*, disregarding the current status of the messages that are already to be delivered to the waiters.
- **Signal queue:** dequeues all the entities stored in the notification queue for which at least one message is pending and notifies them. This operation **can** return a list of notified entities for the implementation to use it.

Notice: As the implementation of a structure fulfilling the given requirement is not easily derived from the general description in this specification, a conforming concrete structure is described in Appendix A – Implementation details.

Message

Messages stored in the queue shall be FOM *items*[15]. Items are the standard data description adopted by FOM and then exposed to the high level languages it implements.

Each subscriber receives the same copy of the

Item in the queue are themselves read-only, but their content is not. The subscriber receiving a message in form of an item, receives it as passed by value; however, it still has modify access to its structure, in cased the passed message-item is an object. This allows subscribers receiving a message before others to alter it. However, notice that:

1. The FOM doesn't grant synchronous access to the objects that can be transferred as messages. Any form of synchronization must be implemented at the final program level by the final agents.
2. There isn't any guarantee, nor any fairness policy, about the order by which different subscribers will receive the posted messages.

Provided the subscribers cope with this limitations, they can safely modify

Context manager

The *context manager* receives contexts that are swapped out by the *processors*, and is in charge to putting communicate the processors when the conditions for them to be ready again are met.

The context manager is composed by the following elements:

- A *context input queue* (CIQ) which contains the context that are to be inspected soon. **If** the implementation exposes shared resources to the environment, **then** the CIQ contains also *environment wait handlers* that require a timed wait on a certain resource.
- A *signaled shared resource input queue* (SRIQ) which contains the shared resources that have been signaled, and on which some context is waiting for notification.
- A *message input queue* (MIQ) where messages relevant for the manager are received. This includes implementation-specific messages as, for example, the request to terminate the execution when the environment wants to terminate the whole FOM engine.
- A *sleeping context set* (SCS) which contains all the contexts that are not eligible for immediate reschedule.
- A *quiescent context set* (QCS) which contains contexts that are now runnable, but whose *context group* maximum runnable context count doesn't allow to be actually put into execution.
- A *runnable context queue* (RCQ) on which contexts scheduled for immediate execution are sent, and that is accessed by the processors through a single-producer multiple-consumer scheme.

Notice: An implementation might find useful to merge the logically different CIQ, SRIQ and MIQ in a single physical input queue. They are kept separate in this specification to better explain the different actions the manager is required to perform as different kind of data is received through this queues.

As contexts are swapped out, they are queued in the CIQ in a multiple producers-single consumer scheme. The manager is the consumer that pulls unscheduled contexts enqueued by the processors.

[15] Todo: insert a cross-reference to item description.

As the manager dequeues a context, it checks if it's part of a context group. If it is, then the running context count of the group is reduced by one. If the context group has a maximum number of runnable context, then the quiescent context set is inspected for quiescent context from that group that can now be run.

In this case, the manager removes quiescent contexts from the quiescent context set and increments the running context count until the maximum count limit is hit, or until no more contexts from that group are in the quiescent set.

This operation is *fair*; the first contexts in a certain group sent to the quiescent context set will be the first ones to be sent to the runnable context queue.

After checking for quiescent context that can now be run, the context manager checks if the *context status register* of the incoming context declares that it's marked for garbage collector inspection. In that case, the context is enqueued for the GC to check it, and will be handled back in the CIQ (with GC inspection request mark removed) when the check is complete.

When a context not requiring GC inspection is dequeued, the manager checks if the reschedule rendezvous time is elapsed. Explicit sleep and resource wait requests are accompanied with a rendezvous absolute system time, with a special value (for instance -1) indicating infinite wait, 0 indicating immediate re-schedulability and any other value indicating an **absolute** point in time in which the context should be made runnable again.

If the rendezvous time is in the past (or zero):

- if the context is not part of a *context group*, or if it's part of a group with current running context count lower than its maximum runnable context count, the context is immediately sent to the *runnable context queue.*
- Otherwise the context is stored in to the *quiescent context set*.

If the rendezvous time is in the future, prior sending the context to the sleeping context set the context manager must check if the context is actually waiting on some resources. If it does, then the manager will perform a try-wait operation on every resource the context is waiting on. If any of the try-wait operations succeeds, the context is immediately sent to runnable context queue, with the resource as the return value of the current wait function the context is engaged in.

Rationale: The resource might have been signaled while the context was traveling in the queue, or during a GC inspection. While the resource shall post the fact of having been signaled to the manager through the SRIQ, this notification might arrive before the manager has a chance to inspect the waiting context. This also means that a waiting context that engages the wait on a shared resource after a given one, but manages to enter the sleeping context set before that one, will be notified first. This doesn't break the general fairness rule as the context doesn't actually exit the wait process until it is inspected by the manager.

If *environment wait handlers* are provided by the implementation, **then** they are simply signaled if the wait time is already in the past as they are received in the CIQ, or put in the sleeping context set if not.

The manager must also check periodically for contexts in the sleeping context set to have reached their wake-up rendezvous time. When the this happens, the awaken context is moved out of the sleeping context set and then:

- if it's are part of a group and the maximum runnable context count equals the running context count, the context is sent to the *quiescent context set;*

- otherwise, it's sent to the runnable context queue.

Notice: The FOM implementers are free to use any strategy that allows to promptly move sleeping contexts with expired rendezvous time *out of the sleeping context set*.

If *environment wait handlers* are provided by the implementation, **then**, as their wait expiration time is reached, they get simply signaled.

When the manager sends removes a context from the *sleeping context set*, it also removes that context from the waiting queue of any shared resource the context might be waiting for; in other words, any wait that the context was trying to fulfill on any shared resource is contextually canceled. Also, the rendezvous time is set to 0, so that it is clear that the context is not sleeping any longer.

When a context is sent to the *runnable context queue*, and it is part of a *context group*, the r*unning context count* of the group is incremented by one.

Context termination

The manager also receives notifications for context termination (from groups being terminated because of errors) in the *message input queue*. When the notification is received:

- If the context is part of a group, the running count of the context is decremented by one, and the completed context count is incremented by one. If the complete context count equals the count of contexts in the group, the parent context is notified.
- Otherwise, the environment is informed that the process is complete.

In case the context for which a termination notification is received while it's already scheduled in the *ready context queue*, the FOM implementation is free to try to remove it from the ready contexts or just have it terminated as it reaches a processor that can see that the termination flag in the *context status register*.

Rationale: at first glance the first option seems more effective and performance wise, but setting that as a requirement may force the implementer to choose solutions that are sub-optimal in the mean non-pathologic case. In a well behaving program already living in a production environment, a context group being terminated by uncaught errors should simply never happen; and when it does, it will probably lead to the whole program being terminated. In that conditions, trying to optimize the terminate-now case at the expenses of the normal ready context workflow seems unwise.

In any case, if the context is part of a *context group* with a *maximum runnable context count*, the quiescent set is searched for other contexts of the same group as when a context is delivered to the manager through the context input queue.

Signals from shared resources

As a shared resource is signaled, the context manager is informed through the *signaled shared resource input queue*, and the contexts that are still waiting and can acquire the resource are notified and moved to the ready queue.

This is performed by invoking the **signal queue** operation of the shared resource. After doing that, the manager checks if any of the waiting contexts in the waiting queue have been actually signaled, and sends them to the *runnable context queue*.

Notice: the action of moving a sleeping context in the *runnable context queue* cancels waits on any resource the context is still waiting for, as the specification above mandates.

Notice: The implementation detail of how to efficiently check which sleeping context, if any, has

been signaled by the **signal queue** operation is outside the scope of this specification. It is **suggested** to have a list of signaled context out of the signal-queue operation, and a mean to find the context by id or direct pointer in the sleeping set.

Garbage collection

Once a context marked for garbage collection reaches the context manager, it is sent to the garbage collector for inspection. Here it is marked through a common mark loop, so that all the resources that can be currently reached by the context are set as non-reclaimable; then, it is immediately sent back to the context manager for further processing.

The garbage collector will separately perform a sweep loop where unused memory is actually reclaimed.

It is **advisable** to organize the garbage collector so that it can immediately inspect incoming contexts through a producer-consumer pattern, where the producer is the context manager. As a new context is ready for inspection, the collector checks it even if it's currently engaged in a swap loop, possibly suspending it or delegating it to a separate thread.

Rationale: in mark-sweep collectors, empirical observations show that 70%-90% of the time is employed in sweeping the memory out (freeing memory takes O/S time). By granting the mark process a higher priority, or at least immediate attention, it is possible to reduce the time in which a context cannot be made runnable again after a suspension.

True parallelism and time slicing

The FOM provides a mechanism to perform true parallelism as well as time slicing of each code involved in parallel execution. Each processor is handled by an underlying O/S low level parallel device (generally known under the name of *machine thread*). The FOM implementer *may* provide means to bind each FOM processor with physical resources (i.e. through CPU affinity masks, or other O/S devices), or to provide other low level parallelism devices to create the processors (i.e. green threads). Anyhow, the minimal interface the FOM implementer must provide are:

- A mean to set the FOM processors to an arbitrary number from 1 up to a reasonable physical-machine and/or O/S reasonable maximum.
- A mean to retrieve the count of physical computational devices.
- A mean to set the number of FOM processors to a reasonable default, where this default is a function of the physical computation devices[16].

The ability to change the number of processors **must** be given also while the FOM is actively executing one or more processes, although the implementation constraints are not specified and the request to change the number of processors *can* be honored also after an arbitrary delay (i.e. lazily as a processor yields a context on which it was working).

Whether this information and interface are available to the underlying code is *unspecified* (and largely language dependent).

[16] For instance, on a machine/OS combinations where I/O is blocking, it might be reasonable to set a default of processors to the number of available cores + 1 or 2, so that all the cores are actively used also during long I/O pauses. On other architectures, for instance on high-performance web servers in farms, it might be more reasonable to set this number to the count of cores -1 or -2, under the assumption of the same physical machine is always in need of some free resource to promptly server foreign requests (and minimize process swap times).

Each group **can** specify the number of contexts that can concurrently run on FOM processors at *creation time* (and this number cannot be changed later on). If given, the group will present to the FOM a number of context ready to be run never greater of this number; otherwise, the group will present all the ready contexts for immediate execution.

As long as there are free processors, all the contexts presented to the FOM by the groups (or by the *processes* in the case of the *main context*), are concurrently run. When there are some ready context that cannot find a free processor where to be run, or that are explicitly held back by the groups that don't want to put more context into active execution, the processors start time-slicing their execution.

After a timeout, they check if there is some ready context still unserved, and if there is, if possible[17], they slice out the context they're serving and take the ready context. Otherwise, they just proceed till the next timeslice.

Time-slicing and true parallelism are hence outside the control of the context and the code related to them; it's transparent at language level. However, it's easy to create pure time-sliced execution environments by creating groups with a concurrency limit set to 1, while it is granted that if there are less or equal context to be run than processors, they all will be concurrently executed[18].

Note on blocking I/O

FOM specification doesn't define any behavior to be taken during I/O operations. This means that blocking during underlying operations that require I/O is a correct behavior for FOM implementations. As the set of operations that a FOM can run is not predefined, it is **required** that the implementers **don't** provide any specific mechanism to swap the contexts out from processors during I/O to avoid block them, unless the language implemented through the given FOM instructions has some sort of constraint that makes it possible to control any form of I/O[19].

On the other hand, FOM implementation **should** provide a shared resource representing the I/O status of a resource (ready to read, ready to write, etc.) to synchronize on I/O non-blocking conditions; in this way, through a little cooperation of the underlying language, synchronization on I/O can follow the same generic rules of synchronization in the virtual machine. Some language implementations may even chose to force the final programmer to synchronize on the shared resource prior performing an I/O operation on the related underlying stream.

Rationale: the FOM is meant to run scripting languages that interface foreign, low-level machine code. Theoretically, it would be possible to provide them with an interface to which they must comply to perform blocking operations[20]. In reality, these approaches are hard to be ported consistently across different concrete O/S architectures, and some third party libraries that a scripting language may want to use cannot be trusted or instructed to behave accordingly to the FOM specifications. For these reasons, adding a high-level mean for the scripts to synchronize on I/O blocking operations grants about the same level of concurrency but higher consistency and robustness. By explicitly requiring the implementers **not to make** any assumption on the characteristics of the I/O operations, the FOM model can be made more consistent across different architectures.

---

17 This is just a generic, high level description of the process. A detailed description is in the following paragraph.

18 Whether this concurrency is real or emulated by the underlying O/S depends on the number of real cores available in the physical machine.

19 In other words, the language specification should prohibit to provide I/O via third party libraries.

20 For instance, we may require the code to use just stream objects provided by the FOM or providing them with some declarative call through which they inform the FOM about being on the brink of performing blocking operations.

Input/Output structure

The FOM interacts with the I/O devices provided by the environment through an entity called *Stream.*

The Stream interface must provide the following functionalities:

- Read and write an arbitrary count of raw data as defined in the *environment* (for instance, sequences of 8-bit bytes).
- Synchronize on read or write availability.
- Provide text oriented input-output through text filtering objects called *Transcoders*.

A *transcoder* is simply a filter that takes data meant to represent text to be stored from the stream or restored from it.

Streams are provided by interfaces to the environment exposed by the final implementation or by third party libraries and modules.

Programs and contexts

A FOM *program* is set of instructions, called *Psteps*, which the FOM *processors* interprets one at a time.

The *program* is organized in a set of one or more instruction trees, called *functions*, each of which having a determined purpose. A minimal *program* is composed of a single *function.*

Notice: the *function* as known to the FOM is not necessarily perfectly matching the concept of a high level language function as exposed by the final implementation. In this context, *function* just means what the above definition declares: a tree of Psteps having a certain, self-consistent purpose.

The *functions* stored in the program produce a result by operating on the input streams provided to it or requested by it, and by manipulating the internal status of the *execution contexts* they operate in.

The status of the *execution contexts* is represented by a set of *stacks*[21]*,* where temporary information needed for the program to operate is stored.

Initially, the *program* is sent to a single *processor* for execution, and is given a single *execution context*; the instructions provided by the Psteps in the *program* are then able to request the FOM to spread the execution creating new *execution contexts*, eventually sending the *program* and its *contexts* to other *processors* for parallel execution.

Notice: The lifecycle of a process is described in the paragraph named Process lifecycle on page 5.

Notice: It's useful to remember here that Psteps are not necessarily countable in number or fully known in advance, and that the tree of steps which constitutes a function needs not to be unchanging through the execution of a *program.*

Context structure

Each context is composed of the following elements:

- **Data stack**: A last-in first-out structure where arbitrary data is stored. Data stored in this stack is represented by items, which are described in the chapter called Data model on page 25.
- **Code stack**: A last-in first-out structure containing the Psteps currently being executed, paired with an integer representing their execution status, called *sequence ID*.
- **Call stack**: A last-in first-out structure keeping track of the *function* currently being processed.
- *Acquired critical section shared variable*, or simply **acquired shared**: it's the shared variable with critical section semantics that the context is currently engaged with.
- **Waiting list**: list of shared resources that the context is currently engaged in waiting.
- **Context status register**: a register keeping the information about the current context status.
- **Parent group**: *Context group* where this context resides.

The implementation **can** define other stacks and elements for the purpose of tracking langauge specific entities.

[21] The term “stack” refers to the standard implementation of a last-input first-output structure.

Code stack and Pstep processing

The code stack holds the Psteps currently being evaluated by the FOM, and an integer status value associated with each of them.

The topmost item in the code stack holds the Pstep that the FOM processor is evaluating in the current context; changing it means to ask the processor to evaluate a different Pstep.

A Pstep keeps being evaluated until it explicitly removes itself from the code stack, or is anyhow removed by any operation.

As the *environment* specific code composing a concrete Pstep is entered for evaluation, it **must** assume that it is stored in the topmost code stack frame; it can freely and asynchronously access the *sequence-id* value in the topmost frame for its own purpose.

The implementation **must not** break this assumption.

Notice: for instance, the implementation must not call directly the code that is part of a Pstep implementation unless it has also consistently modified the code stack of the host *context*.

Pstep implementations can alter the code stack by adding one or more code stack frames with arbitrary Psteps and *sequence-id* values, or by removing the topmost code stack item.

Psteps can also remove an arbitrary number of topmost code stack frames, up to the base level recoreded in the current call stack frame.

If they do so, they must give up the assumption that the topmost code stack frame represents themselves.

Notice: for a Pstep implementation ,it's good practice to give up the execution control by returning to the caller as soon as it alters the code stack.

Notice: It is also possible to call another Pstep that has been just pushed as the topmost code stack, but care must be taken that this is not done indefinitely, or the topmost loop of the processor could be excluded from controlling the status of the context for too long.

Call stack

The call stack holds information about the borders of the self-consistent Pstep tree units called *functions*.

As a new function evaluation is entered, a new call stack frame is pushed on top of the stack to represent the fact.

As the current function is exited, the topmost frame is removed.

Notice: the current function can be exited either because of its Pstep tree has been fully processed or because of an explicit request to exit it, or also because the implementation has generated an error that can be handled by error handlers installed previous element of the call hierarcy.

Whenever a context starts being processed, an initial call stack frame is pushed.

Rationale: As self-consistent Pstep trees must be grouped in functions, this means that the minimal execution unit processable by the FOM and by each processor is a *function*.

When the last frame in the call stack is removed, the *context* execution is considered complete. If the *context* is currently the only one active in a *process,* the *process* is considered complete and terminated.

The call stack holds the following information:

- The currently executed function[22].
- The depth (size) of the data stack when the function was entered
- The depth (size) of the code stack when the function was entered
- Any other implementation-specific information that is to be known about the current discrete boundary context.

Notice: a typical implementation-specific information in object-oriented languages is the “self” or “this” object that represents the object on which a function is applied, hence becoming a “method”.

Data stack

The data stack holds the data currently processed by the Psteps.

Before entering a Pstep processing, the topmost data stack elements are considered the parameters of the operation the Pstep is meant to perform. After exiting the Pstep, the topmost data is considered the result of the performed operation, which can then be considered the paramter for the next Pstep to be performed.

Psteps not generating values must simply maintain the data stack in its current status.

Notice: High level languages providing expressions will implement expression symbols as Psteps; the arity of the symbol will map to the number of parameters the Pstep takes as input. The number of expression results (usually, but not necessarily, one element) will map to the items left by the Psteps on the stack as they complete their processing.

Notice: High level languages using statements as their computational unit will need to remove the extra data left by the expressions. For instance, a typical conditional branch statement should be implemented by a Pstep that takes the value(s) left on top of the stack by the expression representing the branch condition. After considering the truth value of that data, it should take care to remove it from the stack, as generally statements do not generate results used by subsequent statements in the program.

Standard Psteps

Every FOM implementation **must** provide a set of Psteps with a pre-defined behavior.

As the common processing pattern of a Pstep is that of taking operands from the data stack and deposit one or more results, the prototype of Psteps is described with the following notation:

<op1> <op2> … <opN> => <res1> <res2> … <resN>

where opK is a generic nth operand taken from top of the data stack (opN is the topmost element, opN-1 is the second topmost, op1 is the Nth element from stack top), and resK is the generic result left in the stack after removing all the operands (resN being the topmost stack element).

Return

Purpose: Exits the current call frame.

Prototype: <object> => <object>

This Pstep performs the following operations:

[22] remember that the *funciton* is a set of the Pstep tree being currently processed and any implementation-required information regarding that function; for instance, its name, it's parameter prototype etc.

- separately records the topmost stack item as the “return value”;
- reset the code stack and data stack to the depth (size) recorded in the current call frame;
- removes the current call stack frame;
- pushes the recorded return value in the data stack.

Push null

Purpose: Pushes a null item in the data stack.

Prototype: => <null object>

This pstep is meant to simply add a value intended as “nothing” in the target language on the data stack.

After adding the null object to the data stack, this Pstep removes the topmost code stack frame.

Pop data

Purpose: Removes the topmost element of the data stack.

Prototype: <object> =>

After removing the topmost stack, this Pstep removes the topmost code stack frame.

Exchange top stack

Purpose: Exchange the topmost and second topmost items in the data stack.

Prototype: <op1> <op2> => <op2> <op1>

Notice: Some standard operations use two topmost stack items, but the data might be placed in the wrong order by previous operations. This Pstep helps to handle the situation through a standardized operation.

After exchanging the two topmost items this Pstep removes the topmost code stack frame.

Rotate third data

Purpose: Exchange the topmost and second topmost items in the data stack.

Prototype: <op1> <op2> <op3> => <op2> <op3> <op1>

Notice: Some standard operations use three topmost stack items, but the data might be placed in the wrong order by previous operations. This Pstep helps to handle the situation through a standardized operation.

After shifting the third topmost element to the top of the data stack, this Pstep removes the topmost code stack frame.

Data model

Falcon Organic Machine data is represented by entities called *items.* Items are *logically* composed of the following elements:

- A *class.*
- An *object.*
- A *method function*.
- A set of *item flags*.

The attribute *logically* here means that it is not necessary for the items to actually store this elements in their structure.

The *class* in the item is an entity which exposes a set of functions for the virtual machine to operate on the *object* that is stored in the same *item.* In other words, the item *class* is actually an handler for the *object*.

The *object* is an opaque entity which is handled by the FOM exclusively through the functionalities exposed by the *class*.

Notice: the *class* of an item in the FOM has not necessarily a direct relation with any object-oriented programming class entity exposed to the final language implementation.

The *item flags* are a set of binary attributes that the virtual machine and the final language implementation use to keep track of special item significance. The specification requires the following flags to be implemented:

- *Copy flag*: the item was copied by a language-level assignment from or into another item.

Rationale: This allows to implement lazy quasi-deep entities. For instance, it is possible to implement a string type which is just copied by reference as an assignment is done, but then is re-allocated as a new string when a change is performed in any of the existing copies.

Other flags **can** be declared and users by the final language implementation.

The *method function* is a function that shall be applied on the item at a future time, and that will take the item in which is stored as the entity on which it is applied.

The implementation **can** declare other elements as part of the item structure.

Notice: language level type identifiers can be stored as part of language-specific item structure as well.

Item assignment and visibility

Assignment and changes to an *item* are *atomic.* Each change to the any element of the item structure *(object, class, flags, method* and other optional elements used by implementations) must be immediately reflected to any processing happening at the same time in other parts of the the FOM[23].

Moreover, multiple changes to different elements of the same *item* are to be propagated *atomically,* so that they appear to happen at the same time when separately read.

Concurrent access to *objects* is regulated by their *class,* which has the role to arbiter concurrent access to the same object instance.

[23] Avoiding to use the terms “process” and “parallelism” to avoid confusion with the same terms referred to the FOM internal structure.

Notice: The FOM doesn't provide any low level, virtual machine related mean to synchronize access to various elements of an *object,* even when the *object* is exposed to the final language as a set of structured *items.* Language-level means to grant coherent visibility of changes to multiple *items* being presented as a part of a single *object* structure are to be implemented by the final language.

Class functionalities

The *classes* of *objects*, or the handlers that must interface any entity that can be stored in an item, present the following interface:

- **Create**: creates a non-configured but ready for use *object*.
- **Mark**: invoked by the *garbage collector*, this informs the class that the item is still reachable.
- **Check**: invoked by the *garbage collector*, this asks the class if the object is still to be considered alive.
- **Destroy**: informs the *class* that an *object* is completely discarded by the FOM. The *class* can actually destroy it, or inform its *environment* that the object is not needed by the FOM anymore.
- **Mark self**: invoked by the *garbage collector*, informs that the *class* itself is still reachable.
- **Check self**: invoked by the *garbage collector*, asks the class if its own structure is to be considered alive.

Rationale: the FOM doesn't know the internal structure of an object and is not able to mark it for garbage collection. Support from the class is required. Also, this lets the class to actually stay in control of objects that are not to be disposed by the FOM; for instance, if an object is meant to stay alive for the whole duration of the programs run by the FOM or even longer, the class might simply ignore the GC messages and always report the object to be alive.

Notice: The FOM wants to mark and collect *class* entities as well; classes dynamically constructed by loadable/discardable modules (dynamic plug-ins) are to be garbaged as any other data.

Other than this, the *class* **must** expose means to be destroyed by the FOM through appropriate *environment* calls.

Notice: for instance, constructors and destructors in C++ language are a standard mean for the FOM to create and destroy classes. Implementation in other languages, for instance, in C, shall provide standardized functions to create and destroy classes so that the FOM can invoke them.

Class operand support

Other than basic operations on objects, classes **must** expose a set of *operand-like* functionalities that the FOM can invoke through its *PSteps* to perform standardized operations on *objects*.

This *operand-like* functionalities take parameters at fixed positions in the *data stack*, and eventually modify it by removing the consumed data and placing results on top of it.

Notice: FOM stacks are still undefined, but they require the data model to be defined prior their definition can be given. This means we'll be talking about the data stack here and define it later on.

As PSteps, operations on the data stack performed by operand-like functionalities exposed by classes are described through the following notation:

<op1> <op2> … <opN> => <res1> <res2> … <resN>

where opK is a generic nth operand taken from top of the data stack (opN is the topmost element, opN-1 is the second topmost, op1 is the Nth element from stack top), and resK is the generic result left in the stack after removing all the operands.

Operands taking a variable count of operations are indicated using the ellipse (…) in the list of parameters. Interface to operand-like functionalities taking variable parameters **must** include the number of parameters actually passed to the operand.

Implementations are free to add implementation specific requirements to the classes to support higher level language operations.

Notice: This is specular to the fact that the FOM doesn't mandate a rigid set of Psteps, or atomic code units, as they are defined by the implementations to support and reflect directly their language structures.

Initialize

Purpose: configures a newly created *object* through a set of parameters received as *items* from the FOM.

Prototype: <object> … => <object>

This operation takes an uninitialized, newly created object pushed in the data stack of the context and a set of parameters used to configure it. The parameters are then removed, and the initialized object is left on the stack.

The implementation is free to change the initialized object into some other object if this is consistent with its initialization paradigm.

Get property

Purpose: Gets the value of a property, that is, a named element of the given object, leaving the object representing the property value on the stack.

Prototype: <object> <property> => <value>

The adequate object type of the property is defined by the final implementation. It's generally, but not necessarily, a sequence of characters representing a property name.

In case the property is not defined for the given object, the implementation is free to chose to deposit an object indicating the failure of the operation or to set the error status in the *context status register*.

Set property

Purpose: Sets the value of a property, that is, a named element of the given object, leaving the object representing the assigned property value on the stack.

Prototype: <value> <object> <property> => <value>

Usually, the resulting value is the same assigned at the property, but the class of the assignee object **can** change it into something else. If just leaving the assigned value on the stack, the implementation **must** set the *copy flag* of the value *item.*

Rationale: generally, the value given to an assignment expression as seen by high level languages is the assigned value. For instance, an expression like (x=1) is usually interpreted has having itself value 1. Similarly, (x.y=1) should be finally evaluated as 1. However, classes are the masters, and can decide that the appropriate result for an assignment is different than the value that should be

stored in the property.

The adequate object type of the property is defined by the final implementation. It's generally, but not necessarily, a sequence of characters representing a property name.

In case the property is not defined for the given object, the implementation is free to chose to deposit an object indicating the failure of the operation or to set the error status in the *context status register*.

Get index

Purpose: Gets the value of an indexed element of an object.

Prototype: <accessed> <accessor> => <value>

This operation is meant to retrieve an ordinal component of the given object; for instance, a numbered element in a vector, or a value associated with a given key in an associative map structure.

In case the accessor doesn't point to a defined entity in the accessed object, the implementation is free to chose to deposit an object indicating the failure of the operation or to set the error status in the *context status register*.

Set index

Purpose: Sets the value of an indexed element of an object.

Prototype: <value> <accessed> <accessor> => <value>

This operation is meant to change or set an ordinal component of the given object; for instance, a numbered element in a vector, or a value associated with a given key in an associative map structure.

In case the accessor doesn't point to a defined entity in the accessed object, the implementation is free to chose to deposit an object indicating the failure of the operation or to set the error status in the *context status register*.

Compare

Purpose: Determines local ordering, equality or difference between two objects.

Prototype: <comparator> <compared> => <ordering value>

The operation checks for equality or eventually minority of the comparator object with respect to the compared one.

The resulting ordering value should be an entity that is meaningful to the final language implementation as a message that the two objects were equal, or one smaller than another, or simply different.

In case the two objects cannot be compared, the implementation is free to chose to deposit an object indicating the failure of the operation or to set the error status in the *context status register*.

Convert to Boolean (isTrue)

Purpose: Determines if an object or expression result is to be considered having a boolean truth or falsehood value

Prototype: <object> => <boolean>

This operation is meant to convert the object on top of the stack into an entity representing a boolean truth or falsehood value for the final implementation.

Notice: This boolean value might be a boolean type if the language supports it, or the numeric values of 0 and 1, or in general anything that the final language interprets as being “true” or “false”.

Parametric evaluation (call)

Purpose: Evaluates the given object, eventually creating a call frame in the process.

Prototype: <object> … => <result>

This operation applies the concept of parametric evaluation to the given object. If the object represents an independent consistent Pstep tree (code), or *funciton*, then a new call frame **must** be created, and the stored code is placed in the code stack and then processed by the FOM.

If the object represents a non independent Pstep tree, the stored code **must** be placed in the FOM for immediate processing.

In any other case, the the evaluation process is defined by the final implementation, and in particular by the class handling the given object.

Notice: In many high level languages, especially in lambda-calculus based languages, the default result of an evaluation is the evaluated entity itself. For instance, any atomic value as a number, a string, or the null/neuter value, evaluates exactly as that number, string or null/neuter value.

In any case, as the evaluation terminates, the final implementation **must** either remove the parameters and the evaluated entity and then substitute them with a value intended as the evaluation result, or set the error flag in the *context status flags*.

TODO

- Better specification and explicit definition of the context group.

Appendix A – Implementation details

This appendix explains some relevant implementation details which seems to be necessary or highly advisable to correctly implement the Falcon Virtual Machine defined in the previous document.

Context structure

Contexts are objects that are shared between several items:

- the *context group* they are part of;
- the *processors* they run on;
- the context manager keeping a global account of them, or having them accounted in the sleeping and ready to run lists;
- in some cases, the environment that might reference them i.e. for error report or live debug;
- eventually the process they are part of, which might have a global context table that can be used as a reference to lookup contexts by their ID, name or any other mean to identify them.

This suggests that contexts are to be reference counted among their current holders.

A unique name and/or a numeric ID are also advisable features, although none of them is strictly necessary.

The contex status register must be an item that can atomically record for various status-wise events to have happened to the context. It cannot be a simple integer status enumerator, as several events may happen or be cleared concurrently; for instance, the garbage collector may ask to inspect the context while an error is being thrown. Of course, which status prevail is not a matter to be left to a race between the two status setting procedures, as some priorities are involved in the process. For instance, a context even willing to sleep must still be inspected even if the sleep request was issued after a GC inspection request, and it must not sleep nor be inspected if an error was raised right after.

As such, it is necessary to provide a structure with the following characteristics:

1. Must keep record of what status-wise events were set.
2. Can be atomically cleared of a single status-wise event.
3. Can be atomically checked for any status to be set at all.

For instance, an synchronized doubly linked list of events objects could support this requirements, but it would be a relatively complex structure to maintain. The suggestion here is to adopt an atomically accessed bit-field integer, with the following bit-mask meanings:

- Status clear (ready to run or running): 0
- Engaged in critical section: 1
- Garbage inspection request: 2
- Time-slice expired: 4
- Wait on shared resources failed: 8
- Explicit sleep request: 16
- Unwinding stack on error (soft exception raised): 32

- Terminated on error: 64
- Clean termination: 128

This also lets to check for the context to be “done” by checking the high bits of the context status register, while anything in the range 0-63 indicates that the context is still “alive”.

Having the critical section as bit 4 (16), allows to perform the following inferences:

- If the value is > 1, something happened, else proceed as normal.
- If the value is 3 or 5, proceed as normal (ignore time-slice expire and GC requests if in critical section).
- If the value is > 63, the context is dead.
- Else If the value is > 31, manage the exception.
- Else de-schedule the context and send it to the manager.

This allows to go on the fast path when the context is either running normal or in a critical section; processing a context that is in a critical section while gc or time-slice end are requested will be a bit slower, but still bearable.

Message queue structure

The message queue as described in this specification require a particular structure to be implemented.

The specification willfully do not indicate the type or structure of the messages sent to the queue; they can be any information, so they can be *items*.

The structure works as shown in Figure 3.

Initially the queue is empty. When the queue hasn't any subscriber, incoming messages are discarded.

As subscribers register themselves to the queue, a *cursor* is created for each of them. The cursor indicates the next message a reader should read.

The cursor holds a shared resource, which represents the wait status of the waiting entity. To the subscribers, the cursor presents the same interface of a waitable shared resource, but as the subscribers decide to wait for new messages in the queue, the queue actually posts not itself, but the shared resource held by that cursor to the context manager.

When a new message is added in front of a cursor which currently points to the message tail, it is atomically checked for the connected entity to be currently waiting for news. If it is, then the usual resource signal is sent to the context manager; that will wake up the context that is in wait for the queue to receive new messages. Otherwise, nothing is done.

In the example of Figure 3, as Msg1 is added to the queue in step 1 all the cursors for Sub1, Sub2 and Sub3 get their shared resource signaled. When Msg2 is added, there isn't any cursor pointing to the tail, so nothing else is done.

As each subscriber wakes up, it dequeues the message in front of the cursor (as indicated in step 3 of the figure). When all the cursors are moved past a given message, it can be removed from the queue as shown in step 4.

As a queue exposes the same interface of a shared resource to the subscribers, it doesn't provide any simple means for the subscribers to notify it when they get disposed by the collector or the organic

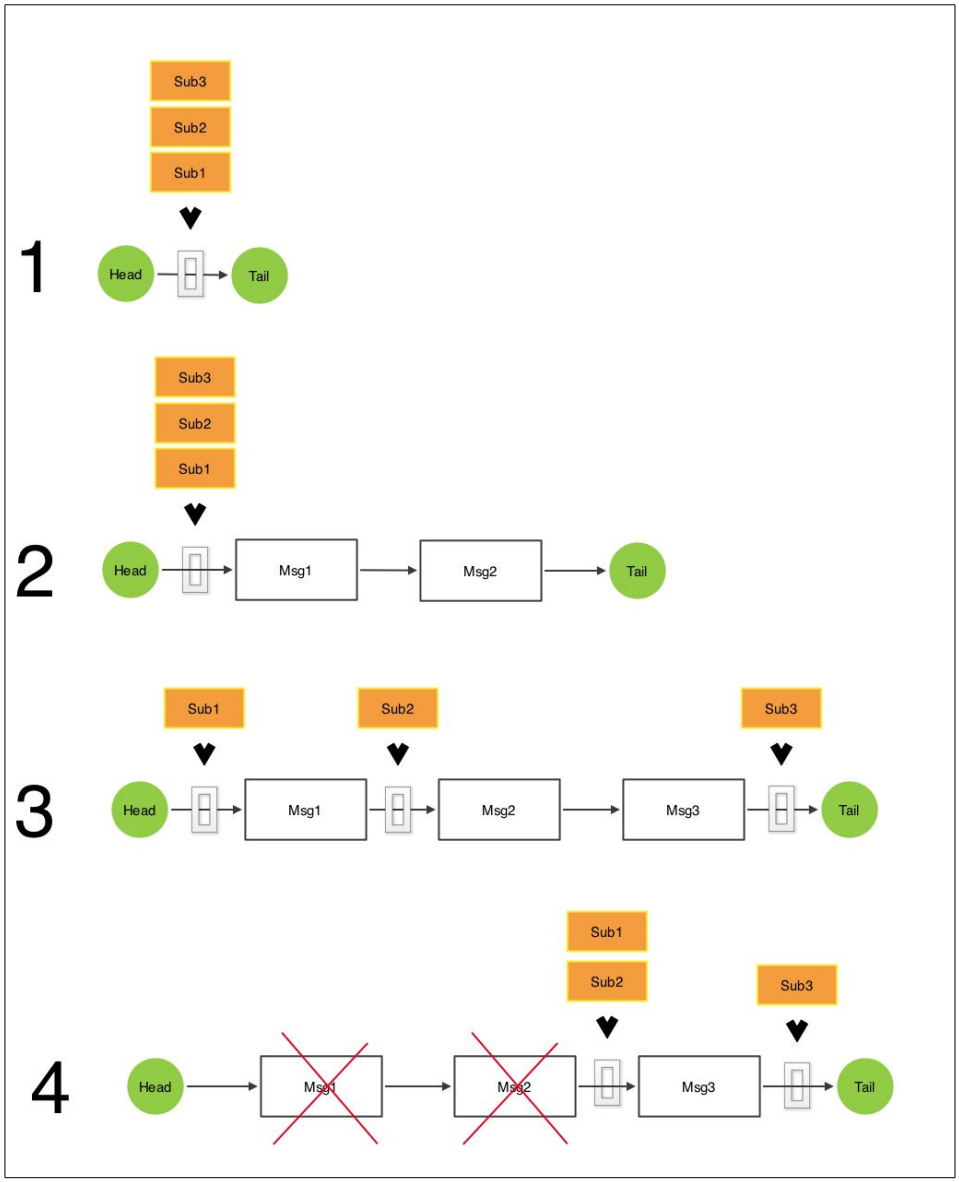

Figure 3: Message queue structure and workflow

machine. For this reason, it is necessary that the cursor holds a *weak reference* to the subscriber entity, so that when a subscriber is destroyed, the queue can be notified and discards the cursor. This means that the queue must periodically check for the cursors to still point to a valid entity, and if not, it must remove them. The most efficient approach is to check all the cursors that still point to earlier messages as a cursor is moved forward after a read; in fact, be pointing to alive entities or not, dangling cursors pointing to newer messages won't cause any problem, while dangling cursors pointing to earlier messages will keep those messages alive even if there isn't any entity that can read them anymore.