P1.Task1.1	Specification of a Schema Language
WP1.D1	Blueprint of the Architecture
Author	Eric Nivel / RU-CADIA
Contributors	Nathaniel Thurston / RU-CADIA
Reviewers	
Status	Version 1.0 Release Candidate 1

Replicode Language Specification

1 Introduction

Replicode is a language designed to encode short parallel programs, models and knowledge in general and is centered on the notions of extensive[1] pattern-matching and code replication. This language is general (domain independent) and has been designed to encode architectures that are *model-based* and *model-driven*, as *production systems* that *understand a domain* in *Freal-time* with *limited resources* and *incomplete knowledge*. Replicode supports the distribution of knowledge and computation across clusters of computing nodes.

In the context of the Humanobs project, Replicode is intended to become the lingua franca used by developers to implement *both* the Domain Architecture (DA) and the Meta Architecture (MA). This document describes Replicode and its associated executive, i.e. the set of systems that execute Replicode constructions (translated into r-code, see below).

The Replicode executive is meant to run on Linux 64 bits platforms and interoperate with custom C++ code. The executive is meant to use mBrane as the underlying middleware providing distribution of computation over a cluster.

An overview presents the main concepts of the language. This is put in the perspective, in section 3, of existing work. Section 4 describes the execution model of Replicode and section 5 the knowledge representation while section 6 describes how the computation is distributed in Replicode. Finally, section 7 describes the executive.

See Annex 1 for a formal definition of Replicode, Annex 2 for a specification of the executive, Annex 3 for the specification of r-code and its C++ API, and Annex 4 for the definition of the Replicode Extension C++ API.

[1] Patterns can be matched by *any code* in a given system, and at *any depth* in the objects' structure.

2 Overview

Replicode is a programming language to specify knowledge and algorithms to implement reasoning about knowledge and algorithms. Replicode is a language that comes in a human-readable form. The latter has to be translated into a machine-readable form – r-code - that will be interpreted at runtime and reciprocally, to translate dynamically generated r-code into the human-readable form. An r-code interpreter, or rCore (i.e. a tiny virtual machine capable of executing one program in r-code), is a generic set of mBrane modules. It is called generic because such an rCore can load and execute any program encoded in the r-code form. rCores are part of the executive (the other essential part is the memory, rMem).

From an architectural perspective, both the DA and the MA will consist of a set of mBrane modules distributed across a cluster of computing nodes. These modules are of the following types:

Type I Modules written in C++, using the mBrane API. Type I modules encapsulate domain-specific code, primarily dealing with I/O. To some extent, type I modules can be called I/O devices. Examples of type I modules are: a pitch tracker, a word spotter, a hum detector, a head position detector, etc. Most of the modules of type I will be developed by RU-CADIA (WP6).

Type II Modules encapsulating learning mechanisms, realizing the function of observers, that is, to learn, recognize and predict occurrences either of known behaviors or of salient phenomena (like shape showing out of noise). These modules are generic in the sense that they can be parameterized to focus on specific behavioral patterns or geometric patterns. Modules of type II will be developed by SUPSI-IDSIA (WP2).

Type III Modules of type III are modules that compose the rCores. The rCore will be developed by RU-CADIA. CM will develop the Replicode/r-code translator (WP4).

Type IV Modules of type IV are specifically designed as part of the Replicode executive (for example, message repositories) and application designers do not typically refer to these modules in their code; in fact designers shall ignore type IV modules.

Type V Modules of type V are application-dependent and coded in C++ using the mBrane API. This is a way for developers to encode programs in a non-explicit way (complied code instead of r-code). Modules of type V are useful to implement programs that do not need to be reasoned about, thus allowing leveraging the speed of complied code (as opposed to the expense of dynamically interpreting r-code).

In general, modules of *all* types shall interact in a standard way, i.e. by sending and receiving messages that carry knowledge *encoded in r-code*. This means in particular that modules of types I, II and V have to embed some device to translate r-code into internal commands and vice-versa – see Annex 3 for the definition of a C++ API to create/interpret r-code. However it is permitted for modules of type V to interact among themselves with raw mBrane messages: these will not be subjected to reasoning, i.e. will not constitute inputs for modules of type III. To some further extent, some modules of type V can chose *not* to interact at all in the standard way, and thus to constitute a “ghost” unreachable at the cognitive level itself (that is the domain of code that can be reasoned about).

Modules of type I and V can be associated with models written in Replicode that express their operational semantics. This allows other modules to reason about them, for example, to predict their behaviors or to know what they can achieve.

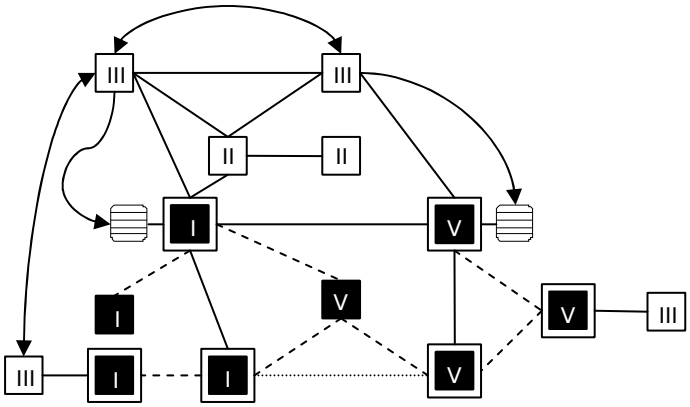

Figure 1 – Architectural Typology

Replicode supports the implementation of hybrid architectures as mixes of reactive and reasoning processes with different depths of semantics and different implementation types. Here are represented possible interactions between modules of all types (except type IV): plain connections denote knowledge flux in r-code, the dashed ones, C++ message flux. All flux have broadcast semantics (controlled at a lower level by mBrane-level subscription schemes). White edged boxes are modules that embed C++ / r-code translators. Arrows represent introspection abilities (e.g. programs reasoning about other parts of the system) - models are represented by the hashed rounded boxes. The distribution of modules over computing nodes is represented.

Architectures coded with Replicode are distributed production systems, where knowledge is scattered across the system, and where program react automatically to knowledge occurrences upon pattern-matching and produce new knowledge. In other words, such architectures are *data-driven*, and are composed of a possibly large amount of programs executed in parallel by as many rCores.

Replicode allows introspection, i.e. the evaluation of knowledge by other programs, including the code of any programs. This meets the fundamental requirement of reflectivity, needed by architectures that need to build models of themselves. In this respect, the executive also notifies programs - at runtime – of essential events that reflect the execution of code (for example, productions, invocation of commands on devices, process spawning and termination, etc).

Said architectures are also *model-based* in that sense that actions (productions) can result from evaluating executable models (forward models to predict, inverse models to reach goals). The evaluation is not performed by brute-force searching, but is instead controlled by the evaluation of other models: in addition to be based on models, the architectures in question are also *model-driven*. The ability to analyze code allows these architectures to reason about actions and contexts, in other words, Replicode and its executive support the dynamic construction of contextualized executable knowledge (for instance, Replicode support searching over the knowledge space and the construction of abstract knowledge, like plans or concepts).

Replicode is a functional language designed to allow both forward chaining and backward chaining, i.e. deductions and inductions. Put in another way, these two ways of exploiting knowledge are akin to, respectively, production and search. Simulation is also supported and constitutes another usage of forward chaining.

Replicode code is structured around the central notion of pattern-matching on knowledge, and enforces a data-driven execution model. Knowledge is represented as instances of the following classes of *objects*[2]:

Program Programs react to incoming messages carrying objects. A program defines a pattern of time-constrained message sequences, specifies patterns

[2] Objects are like C structures (not like C++ objects). Replicode manipulates objects but is not “object-oriented”.

and guards (conditions) on the objects carried by the messages and produces messages with objects built from the incoming ones, the former being called productions. Programs are reactive, i.e. they perform as described whenever incoming messages match the time-pattern and guards.

Programs are state-less and have no side-effects.

Last, programs can run in simulation mode, that is, perform their usual duties while attaching simulation markers to their outputs. To trigger execution in simulation mode, incoming messages would be marked as hypotheses.

Forward Model, Inverse Model Both forward and inverse models are objects. Forward models predict the outcomes of actions given an initial state, whereas inverse models describe algorithms to reach a target state from an initial state. Models behave like programs, i.e. they are reactive objects that perform computation whenever some inputs match the specification of initial states, target states, etc. Models can also run in simulation mode.

Function A function defines a high-level view on more detailed executable code. It is an abstraction. Functions can be executed by programs and models. Functions are not reactive: they must be explicitly called by programs or models to be executed.

Goal A goal object defines a target state in a system. It does so by specifying a pattern. Constraints are encoded as goal objects.

Sub-system A sub-system is a set of references to objects, and a sub-system is also an object itself.

The execution of reactive objects (i.e. programs or models), is controlled by the sub-system they belong to: reactive objects can be activated or not, based on an activation value, to be compared with a threshold defined for the sub.-system.

When objects represent knowledge (models, markers, etc.), their capacity of being inputs for reactive objects is also controlled by the sub-system they belong to: objects can be inputs for reactive objects or not, depending on a saliency value, to be compared to another threshold defined for the sub-system.

Notice that programs and models are reactive object but also knowledge: their code is explicit and can constitute inputs for reactive objects. Programs and models are thus controlled by both activation and saliency.

In some cases, when objects in a sub-system become very active (respectively salient), the whole sub-system can suddenly gain an activation boost. This mechanism allows the activation of objects to be triggered by the activation of semantically related ones.

Objects can be members of only one sub-system, and are hidden from other sub-systems, i.e. they are encapsulated. However, sub-system can pass copies of objects to other sub-systems: such copies live independent lives in their respective hosts.

Sub-systems are hosted by only one computing node and cannot spread across multiple nodes.

Interface An interface is an object allowing passing objects from a source sub-system as inputs to the reactive objects in a destination sub-system. The interface holds an activation value for the destination seen from the source, and a saliency value for the source seen from the destination. Objects from the source will reach the destination if the interface's saliency in the destination is high enough and if the interface's activation in the source is also high enough.

Sub-systems can have several interfaces to other sub-systems, while still being member of (i.e. embedded in) only one sub-system.

Process A process object encodes the fact that a program is being executed. Here "program" is to be understood in a broad way, that is, either, a program as defined in Replicode, or a collection of interacting programs on a broader scale, like a set of programs that achieve a particular goal collectively.
For technical reasons a process is encoded as a marker (see below).

Entity An entity object identifies an entity in the world or in the system. An entity

is a mere symbol used to specify to whom some knowledge applies, be it an event, a prediction, a model, etc.

Marker Markers are tags indicating that a given object is in a specific relation with other objects. A marker identifies the class of the relationship (like in “this_cup has_color blue” where “has_color” would be the class defined by the marker). Thereafter the class of the relationship defined by a marker is simply called “marker class”.

Markers are used to build ontologies. Replicode provides ways to create new markers classes dynamically, thus allowing ontologies to be built at runtime and complement/invalidate/modify user-supplied ontologies.

Device Devices are numerical identifiers for modules of types I, II and V.

Device Function Device functions are identifiers defined by modules of types I and II that represent the commands they expose to Replicode. Device functions are used to encode commands to the devices, whereas these devices use markers to notify the system of their activity.

Objects are embedded in mBrane messages to be transported across the cluster, but this mechanism is transparent to Replicode objects: the carried objects, not the message, are the actual input for reactive objects.

Any object in Replicode holds a *resilience* value representing its time to live in its host sub-system (expressed in time units); when its resilience reaches 0, the object is destroyed. Any executable object in a given sub-system can request the modification of the resilience value of an object, the final value resulting from the averaging of these requests, i.e. resilience values are modified collectively.

The same principle holds for the modification of saliency and activation values held by interfaces.

In addition to object classes, Replicode defines the main following atoms (see Annex 1 for the complete list):

`nil` Nothing.

`|[]` Empty set.

`|nb` Undefined number.

`|ms` Undefined time.

`|sid` Undefined sub-system identifier.

`|did` Undefined device identifier.

`|fid` Undefined device function identifier.

`|bl` Undefined Boolean value.

`this` Self-reference within code (as in C++).

Variables Identify and reference any piece of code.

Wildcards Used to define structures in patterns.

Numbers Floating point numbers.

Timestamps Integers with time semantics.

Identifiers Integers identifying instances of specific classes (devices, sub-systems).

Boolean values.

Replicode also defines operators on atoms. First, operators include the usual arithmetic, logic, algebraic operators that target aforementioned atoms. Second, Replicode operators also include operators to manipulate code. For example, \ (similar to the quote operator in LISP), replace, merge, split, etc.

Notice that Replicode allows developers to extend the set of operators and object classes: this is achieved using an API, the Replicode Extension API.

Replicode also defines a preprocessor (C++ style) for managing code definitions.

Last, Replicode defines a set of functions to control the executive. Examples of such functions are: search (for programs able to satisfy a specified goal), `inj` (inject new objects), `set` and `mod` (to request changes on the resilience, saliency, etc.).

Before getting any further in the specification, here is a summary of the principal characteristics of Replicode with regards to information and information processing:

Programs and Models The prototype of programs/models coded in Replicode has distinctive features, essentially:

- There are no explicit loops in Replicode: the execution is purely reactive (i.e. data-driven) and there is no way to loop around objects in the memory. However, at a finer grain, Replicode allows performing reductions on sets *within a given object*.
- There is no “and” operator in Replicode: conjunctions are achieved by assembling patterns and guards in sets, each of them bearing particular semantics (e.g. input section, timing constraints, etc.).
- There is no “or” operator in Replicode: disjunctions are achieved by running different programs/models targeting the different cases of the desired disjunction.
- Programs and models react to salient knowledge. The saliency of knowledge results from a collective agreement reached by programs.
- Programs have no application-dependent internal states. Knowledge is shared by all programs and models.
- Programs and models run in simulation mode when presented with inputs marked as hypothetical.
- Programs and models have facilities to search the knowledge space for models able to reach some states given an initial state of the system.
- Programs and models all run in parallel. Sequentiality and synchronization have to be explicitly coded for.
- Programs and models are also knowledge, and can be reasoned about: their code is explicit, i.e. readable by other Replicode programs.
- Code can be produced dynamically and executed on the fly.
- Programs and models are small, ephemeral and numerous. Small, because understanding small code is easier than understanding large complicated code. Numerous, because small programs cannot achieve what an omniscient omnipotent Behemoth could (if such a thing ever existed), and we’ll need to compensate the individual specialization and limited scope by the number. Ephemeral, because programs/models exist for a reason that may no longer exist later.
- Modularity is supported in Replicode by means of sub-systems: these encapsulate objects and hide them from other sub-systems. The entire system is a structured as a sub-system, except that it does spread across the entire cluster.

Data The prototype of data coded in Replicode has the following characteristics:

- Replicode represents knowledge as non-axiomatic logic expressions. “Truth” is always to be subjected to interpretation by programs and models and shall be revised. In other words, knowledge can be incomplete, context-dependent, and does not reflect any absolute truth but reflects instead what the system has learned so far from its experience.
- Data is ephemeral. Facts only hold as long as they are (a) of some use for programs/models and (b) as long as they are meaningful, that is, recognized as signifying facts in the domain by other programs/models.
- Once produced, data cannot be modified[3]. The memory holds the succession of occurrences of facts, and it is up to the application to unify these into more permanent constructions, i.e. more general facts that hold for longer periods of time.
- Saliency is contextual. In contrast with anecdotal information or noise, salient

[3] The control values can be modified. However, they are not considered data as they do not convey information about the application domain.

data acquire their relevance to programs/models by collective agreement. Data are said salient if they contribute noticeably to the achievement (or failure) of some function possibly implemented by a collective of programs/models.

- Data is potentially global, i.e. used by all the programs in a given sub-system. Scope restriction is however necessary to avoid the flooding of the system by possibly irrelevant data. Scope is actually controlled (a) by pattern matching, i.e. the scope of information is determined by its usage, (b) membership in sub-systems, i.e. inputs are chosen among salient objects in a given sub-system.
- Data is instance-based. Replicode is meant to support bottom-up construction as well as top-down (i.e. axiomatic) designs and therefore Replicode is instance-based, not class-based: ideally, classes are concepts to be constructed by a system in an attempt to generalize instances – roughly speaking, instances come first. In addition to expanding ontologies (building new classes), the system shall be able to revise them.
- Data can be patterns, i.e. abstractions with free variables and guards.
- Application-dependent data types - and relevant operators and equations - can be wedged in Replicode using an API. For performance reasons – and also to keep complexity low - this way is preferred over the classical way of expressing complex data types as complex constructions involving simpler ones (for example a vector in the Euclidian space would be encoded as a specific type, not a list of numbers).

3 Related Work

Replicode stems from Ikon Flux [Nivel and Thorisson 2008, Nivel 2007], although the former is a strongly limited variant of the latter. In particular, the control of programs and knowledge is much simpler in Replicode than its counterpart in Ikon Flux, and the two approaches toward the distribution of objects and of computation are radically different.

Ikon Flux relies on a flat shared memory model: there is no a-priori axiomatic segmentation of data and computation in the distribution model. Ikon Flux has been designed to allow the formation (and the control) of *emergent* structures. In contrast, Replicode assumes a more conventional way of segmenting computation by the means of encapsulation - sub-systems - that need explicit programs to replicate data and code to other sub-systems. This is due to two factors, mainly, (a) the underlying distribution mechanism used by Replicode (mBrane) has message passing semantics whereas Ikon Flux relies on shared memory semantics and, (b) the programming paradigm endorsed by Ikon Flux is programming by side-effects, which poses difficulties for engineering large systems and also for collaborating with separate, loosely coupled development teams: that is why Replicode provides support for more a top-down design approach, allowing teams to develop independent components more efficiently than with Ikon Flux. Nevertheless, Replicode allows the construction, destruction and reorganizing of components and thus, still allows the dynamic computation of structures – even if it does so at much smaller scales.

The execution model of Ikon Flux is synchronous, i.e. step-locked (as in cellular automata execution models), whereas the execution model of Replicode is asynchronous: this is also due to the underlying distribution mechanism. As a consequence, the ability to leverage massive parallelism for search is reduced in Replicode as the fine synchronization of threads[4] is an issue, and thus, parallel searches are bound to be less efficient.

Probably the most important difference between the two approaches is the following: Ikon Flux implements a global mechanism for controlling the execution of vast numbers of programs in a goal-oriented fashion, that is, programs are selected for execution – and data for processing – based on past experience with respect to goal achievement. In other words, Ikon Flux defines the topology of a system as a space of processes where the distance function is learned, and corresponds to the effective cooperation/competition between programs with regards to achieving goals. This feature has been dropped down in Replicode since it requires essentially (a) comprehensive runtime reflectivity which calls for very low latencies and very high bandwidth – both being not always available - and (b) the ability for a program to react to anything, anywhere and anytime in a system - a requirement that collides with encapsulation, which, by

[4] Replicode uses native threads, i.e. has limited control over scheduling (an issue of granularity of the execution of code) and preemption (both services provided by the underlying OS).

definition restricts the scope of computation and knowledge. As a consequence, in Replicode, the control of the multitude of data and programs is to be explicitly programmed within the boundaries of sub-systems.

In the broad landscape of programming languages, Replicode stands close to P-systems [Paun 2002], in that sub-systems act like membranes, both execution models following the paradigm of production systems. The main difference is that Replicode does not support multi-sets and therefore does not rely on concentrations and quantitative distributions of code. Replicode is also related to Fraglets [Yamamoto et al. 2007, Tschudin 2003], a rewriting system which allows higher expressivity than code found in P-systems, and based on self-modifying and replicating code. To a minor extent, as a functional programming language, Replicode presents some technical similarities with Erlang, and therefore, LISP (although these are not data-driven). Finally, Replicode is message-based and allows the execution of code in simulation mode: these features are shared with AKIRA [Pezzulo and Calvi 2007].

4 Execution Model

This section describes the behavior of individual programs, as executed by rCores.

4.1 Program Structure

The general structure of a program can be sketched as follows (for full details, see Annex 1):

```
pgm; program
[]; a set
   [patterns]; template arguments section
   []; input section
      [patterns]; input patterns sub section
      [expressions]; timing sub section
      [expressions]; guard sub section
   [production-sub-sections]; production section
```

where a *production-sub-section* is defined as:

```
[]
   [expressions]; guard sub section
   [function-calls]; commands to devices
```

N.B.: `[elements]` denote sets.

In addition to this definition, a program has read-only access to a set of runtime data including (see Annex 1 for the complete list):

- saliency (`sln`);
- activation (`act`);
- resilience (`res`);
- the time of its construction: injection time (`ijt`);
- the time of its reception by a sub-system (if passed from one to another): ejection time (`ejt`);
- the host sub-system (`hst`);
- the sub-system that created it: origin (`org`);
- a pointer to itself: `this`.
- a set of all the markers that reference it (`mks`).

This set of runtime data is maintained by the executive, not only for programs but for *any object* in the system.

Programs in Replicode are actually program *templates*, that is, any code fragment in the program can be parameterized by some arguments. For example, the specification of the input arguments, the guards or the productions can be function of the template arguments. Template arguments are specified by patterns: they express constraints on the objects to be used to parameterize the program. Guards in the guard section express conditions taking as arguments any part of any of the input objects, i.e. they can express constraints on correlations *between*

objects, whereas guards in the input section cannot.

Guards in the production sub-sections apply further constraints on any input objects and control the actual production specified in the sub-section. This is meant to reuse the filtering performed before (according to the patterns and guards defined in the previous sections): pattern matching can be expensive and allowing further refinements on data already filtered is less so than defining and running other programs with almost the same initial filtering (input, timing, and guard sections). Common filters are factorized in their relevant sections, and particularities are dealt with at the latest stage, i.e. the production section.

Program objects are automatically executed by the executive when their saliencies rise above a threshold (see section 6).

4.2 Program Execution

General Model The execution of a program can be sketched as follows:

> I. whenever instances of objects are received (carried by messages) and (a) their order of reception matches the timing pattern (i.e. satisfy the guards in the timing sub-sections) and (2) each object individually matches its respective input pattern, then,
>
> II. check the guards in the guards sub section. If none of them returns `nil`, or any undefined value (e.g. `|[]`, `|nb`, etc.) then,
>
> III. for each production sub-section in the production section, (a) check the guards associated with the sub-section and if none returns `nil` or any undefined value, then (b) execute the code specified in the sub-section's production list. Outgoing messages are time-stamped by the hosting rCore. Notification is also handled by the rCore, i.e. injection events are output by the rCore, describing which program successfully matched which input objects and produced which objects at which time.

The rMem handles the case when two programs output identical productions: the latest production replaces any older identical one – identity being defined as the identity of code regardless of resilience, saliency and activation values, and injection time.

Pattern Matching Pattern matching is the only way in Replicode to valuate formal arguments. A pattern is specified as any piece of code (program, model, marker, etc.), except that it may contain variables and wildcards. For example, this is a pattern designed to filter input objects in a program:

```
(ptn (mk.position r: s: ::) |[])
```

This pattern will be matched by any object of class `mk.position` (marker of class position). The variables `r` and `s` would store the resilience and saliency of the matching object. The injection time and other members of the object are not filtered (wildcard ::). Notice the pattern's empty guard set (`|[]`).

Replicode allows matching patterns on sets. The operator `red` (reduction) admits two arguments: (a) the set to be filtered, (b) a set containing a pair {set of pattern to apply on each of its elements, set of productions}. The result of such a reduction is a set containing the productions computed from the elements that match the pattern – in the fashion of the reaction model of a program.

Any object carries a set of marker objects that reference it. Replicode provides the `mks` (marker set) member identifier which returns the markers the object has been associated with. This allows for example, filtering inputs not only on the basis of their own structure but also on the basis of the structure of the markers they have been tagged with.

Finally, notice that patterns can also specify the structure of objects that are *not* expected. This allows programs to react also on the *absence* of some specific inputs during a specific time window (see Annex 1 for details).

For performance reasons, it is not allowed to define input patterns without specifying the class of the expected object. For example, it not allowed to write input patterns to catch "any kind of object such as …".

Execution Overlays Regarding the enforcement of timing constraints, the rCore performs using four main rules exemplified in the following situations:

I. In the example of a program P with only one input pattern IP, if an object o1 is received, then the rCore spawns a copy of P to process o1, and is then ready to spawn other copies of P upon reception of subsequent objects oi matching IP while o1 is being processed: the execution is *overlaid*. This rule applies in a more general way to programs with any number of input patterns and timing constraints (see other rules below).

II. In the example of a program P defined like:

```
P:(pgm
[]
   |[]; template arguments, empty set
   []
      [o1: pattern_1 o2: pattern_2]; inputs
      [; timings
      (> o2.ijt (+ o1.ijt 10)); ijt=injection time member
      (< o2.ijt (+ o1.ijt 20))
      ]
...
)
```

receiving messages like:

```
a1 (matching pattern_1) at time t,
a2 (matching pattern_1) at time t+5,
b (matching pattern_2) at time t+22
```

triggers, upon reception of `a1`, the rCore spawning a copy of P, P1, like:

```
(pgm
[]
   |[]
   []
      [o2: pattern_2]
      [
      (> o2.ijt (+ o1.ijt 10))
      (< o2.ijt (+ o1.ijt 20))
      ]
...
)
```

where `o1` is valuated by `a1`. According to the rule from the example 1 above, upon reception of `a2`, the rCore spawns another copy of P, P2, where `o1` is valuated by `a2`. P, P1 and P2 now run in parallel.

Upon reception of `b` by P1, the timing constraint is violated and the rCore kills P1.

Upon reception of `b` by P2, the timing constraint is respected and further processing by P2 is allowed.

III. In the example 2 above, let's change the order of arrival of `b` and set it to `t+18`. Then, upon reception of `b` by P1, the timing constraint would be respected and P1 would possibly (if other guards are satisfied) output a production, a function of `a1` and `b`. In a similar way, P2 would output another production, a function of `a2` and `b`.

IV. What if, for some reason, the developer would like to avoid P2 producing anything if P1 does? Replicode allows to specify for a reactive object whether the input object matching the pattern (and satisfy the associated guards) will be *consumed* or left available for other copies of the same program (as in the example 3).

V. A program defines a time scope (`tsc`), i.e. a limit in time beyond which an offspring is automatically killed by the executive. An offspring's time to live is computed as $t_{initial}$+`tsc` where $t_{initial}$ is the time of the spawning of the offspring. N.B.: an offspring can be killed before expiration of its time to live in case the program itself dies (when its resilience drops down to 0).

Overlays are the execution of copies (called offsprings) of a given program. Offsprings

therefore share with the original its control values (activation, etc.). From the vantage point of other programs, offsprings and overlays simply *do not exist*, i.e. offsprings are hidden by the original program and overlays by process objects referring to the original program.

Code Modification Programs are state-less, that is, programs do not encapsulate private persistent application-dependent data. Programs however, do hold control values (saliency, activation, resilience, injection/ejection time) but only have limited access to these: the executive maintains (writes) these values. They will thus not be considered internal states.

This means that the only way for programs to memorize application-dependent states is for them to refer to common knowledge, i.e. knowledge accessible (read/write) by any other program. In other words, programs refer to states by pointing to objects in their host sub-system's memory. Such states are explicit and global in a sub-system.

There is also the possibility to *alter* a program's code dynamically and produce a variant that would embody some state: the variant would be built as a function of some previous input object (the so-called state) caught by the original program. For example, let's consider the following case: a program P0 continuously outputs the position of an entity – a ball, under the control of yet another program -, and we need a program P1 that outputs a message M telling the last known position of the ball. Obviously, there is no need to produce M when the ball is stationary. Here is a possible encoding for P1:

```
(pgm
[]
   |[]
   []
      [; inputs
       (ptn o1:(mk.position :b1 :pos1 : : :t1 ::) |[]) ; o: is a label
; o will be valuated by any incoming object matching the pattern. b:, pos: ..
and t: are variables within the pattern and will be valuated by data ..
occurring at the same location in o. : is a wildcard. :: is a wildcard ..
used to discard the tail of the expression.
      ]
      |[]; timings
      |[]; guards
   [; productions
      []
         |[]
         [
         [(inj [
             P2:(pgm; names can be declared anywhere in the code
             []
                []; template args
                   (ptn :ref_b |[])
                   (ptn :ref_pos |[])
                   (ptn :ref_t |[])
                []
                   [; inputs
                   (ptn o2:(mk.position b2: pos2: : : t2: ::)
                    [
                    (= b2 ref_b)
                    (> t2 ref_t)
                    (<> pos2 ref_pos)
                    ]
                   )
                   ]
                   |[]; timing
                   |[]; guards
             [; productions
                []
                    |[]
                    [
```

```
                    (inj [(mk.last_known o ...)]); that's M
                    (inj [(ins P2 |[])]); P2 instantiated by |[] ..
(no template args).
                    (set [this.res 0]); kills the original
                    ]
                ])
            ...]
        )]
        ]
    ] ...
)
```

Expressions are enclosed in parentheses, sets are delimited by brackets.

Carriage returns + tab stand for opening a parenthesis/bracket, carriage returns + backtab stands for closing one.

`;` is a comment.

`..` is the line continuation symbol.

`...` means that additional code is omitted for clarity.

`inj` is a function call to the executive to produce new code (ex: M, P2) and make it available in the system (store in memory).

`set` is a function call to the executive to request to set the resilience of P1 to zero. The executive accumulates potential similar requests on P1 issued in a (tunable) time window and decides a final value accordingly.

All three calls in the production sub section (`inj` and `set`) are published as messages that the rMem will build appropriately.

If there was a need to refer to P2 in P2 itself, then, in P1, this would be written `(\ this)`, preventing the evaluation of `this` in the expression (which would be a reference to P1). This also means that when P2 is injected, the expression `(\ this)` is replaced by `this`, i.e. in P2, `this` is not prevented from being evaluated anymore.

The pattern in the input section describes a marker object of class position (`mk.position`) that points to an object referred to by a variable b and a position, p. The injection time of the marker is known as a variable t, its saliency and resilience values are unused (wildcards :). The first guard on o makes sure that the two objects the position is reported for are actually the same object. The second guard checks that the position report caught by P2 is actually younger that the one caught by P1. The third guard checks if the two reported positions are different. In any of these guards, `this` results in P1, and `this.pos` results in the actual value of the variable t in P1 after an input object o matches the pattern.

P1 catches an occurrence of the ball position, reproduces itself as P2 with three alterations, the guards on o, inject M and kill P1. The same result can be achieved using the replace function (`rep`) which replaces the program currently executed by an rCore by a new one:

```
(rep (ins P2 |[]))
```

instead of:

```
(inj [((ins P2 |[]) ...))])
```

and

```
(set this.res 0)
```

This yields better performance since there is no need to spawn a new rCore with the code of P2, and no need to notify neither the injection of the new code nor the destruction of the old code.

Notice that P2 could be augmented so that it would also kill previous occurrences of M; in our example such occurrences would die “naturally”, i.e. after the delay specified by their resilience value.

Execution Mode Programs react to incoming messages carrying objects. Programs react only to *salient objects* produced so far. When an object is produced, its producer assigns it an *initial* saliency value which can be changed over time. Programs react only to objects with sufficiently high a saliency value (see section 6.1).

Saliency is to be computed by other programs and determined as a combination of features of the inputs. These are for example, the injection time (the latest the better), the initial saliency (the higher the better). In addition to these common features, an application can define its own classes of objects (marker classes) that can introduce additional features to take into account when computing saliency. For example, this could be a measure of reliability on sensors outputs. In this case, higher reliability could outweigh youth, and the most salient sensor output could be the latest of the most reliable outputs.

Each time an object having a low saliency value is modified by a program so that its saliency gets high enough, said object is passed as a potential input to all active programs by the executive.

When a program's becomes deactivated (see section 6), it goes “asleep” and will not receive any inputs anymore. When it wakes up, it receives inputs again as previously, that is, it receives as inputs the objects that *become* salient after it wakes up.

Notice that the code given in the previous section could be modified advantageously so that, instead of injecting `mk.last_known` objects, it would simply update the saliency of the last reliable position of the hand.

Simulation Programs have the ability to run in simulation mode. There exist a built-in marker in Replicode - `mk.hyp` – that represent hypotheses and simulation results. When a program receives at least one input marked as an hypothesis, it then turns into simulation mode, and expects the other inputs to be also marked as hypotheses. Upon successful pattern matching, the program will produce its results as usual, but the rCore will attach a `mk.hyp` marker on them. Hypotheses also refer to a simulation run, that is, a process representing the execution of the program in simulation mode; such a process is marked with a simulation marker (`mk.sim`).

Modules of type I and V are expected to interpret such markers correctly. For example, a device I/O module that receives a simulated command to move a robot's arm shall *not* actually perform the move. Other programs shall use the device's forward models to predict the result of the movement simulation.

Programs switch automatically in simulation mode depending on markers tagging the inputs. This means that both the actual and the simulated performances are controlled by the same activation values.

5 Knowledge Representation

Replicode supports the execution and dynamic construction of hybrid - reactive and reasoning - processes. This section describes the representation of knowledge in Replicode, as needed by processes of the reasoning kind.

5.1 States and Goals

Replicode proposes to disambiguate the notion of state by offering a specific way to encode each of its different meanings or usages.

If by “state”, one means an absolute truth (i.e. atemporal and non-contextual) – in other words, that a logical predicate is true -, like “this cup is blue”, then this shall be encoded as an instance of a marker class, like `(mk.blue cup1)` or `(mk.color cup1 blue)` where `cup1` is the cup referred to by “this cup”. If the predicate has free variables, like “there is an x such as x is blue”, then this shall be encoded as a pattern like

> `(ptn x:(ent ::) [(red x.mks [[(ptn (mk.blue x) |[]) [x]]])])`[5] – meaning “an entity x such as there exists a marker referring to x as being of class blue”.

If “state” denotes a phenomenon under development, like “I am running”, then this shall be encoded as an instance of a process, like `(mk.pro (app (ins run |[]) [self]))` - the ongoing application of the function `run` to the list of arguments `[self]`. The state of a phenomenon having occurred, like “after running for 2 hours”, shall be encoded as a termination marker (`mk.xet`) that refers to the terminated process object.

If “state” refers to the result of the evaluation of an observable of a system, like “I am thirsty”,

[5] N.B.: this is expression is equivalent to: `(ptn x:(ent marker_set: ::) [(red marker_set [[(ptn (mk.blue x) |[]) [x]]])])`

then this could be encoded – in a first analysis - as a marker, like `(mk.thirst_level self a_value)`, with a high saliency, while assuming the existence of active programs that would evaluate this fact (thirst_level=a_value) and take appropriate action, for example decide that the thirst level is too high and shall be decreased: in Replicode, evaluating facts is never done in an absolute fashion, but in the context of constraints or goals (thirst is never too intense in itself, but can be considered so for example, when compared to a threshold that shall not be crossed lest the system suffers unwanted damage). In fact what we really mean by “I am thirsty” is “it occurs to me that I am feeling uncomfortable because I am thirsty and I'd rather not feel uncomfortable”: more than the saliency of thirst, we acknowledge the *impact* of such an saliency, i.e. a goal intense enough to have us think about it; this shall be encoded by a goal object, like:

```
(gol
|[]
[
(ptn (mk.thirst_level self v: ::) [(v < discomfort_threshold)])
]
)
```

with a strong saliency value. In short, the state “thirsty” shall be encoded as the goal “lower the level of thirst”.

To summarize, there is no specific representation for states in Replicode. Instead, states are to be encoded with different constructs – markers (including processes and contexts) or goals – depending on their semantics.

Regardless of the encoding of states, a goal is an instance of a specific class (`gol`) that specifies a target state.

5.2 Functions

Functions are abstractions that represent algorithms, as in most of the programming languages. The structure of a function is:

```
fun; function
[]
   [patterns]; template arguments section
   []; input section
      [patterns]; input patterns sub section
      [expressions]; timing sub section (empty)
      [expressions]; guard sub section
   [production-sub-sections]; production section, as defined for programs
```

This is analogous to the structure of a program, except that there are no timing constraints. Functions do not react to incoming messages. Functions are in contrast to programs, mere building blocks that are not reactive, i.e. they are not allocated a thread.

The algorithm of a function is specified in the production section. Notice that in Replicode, a function can produce *more than one result*.

Functions in Replicode are actually function *templates*, that is, the guards and algorithm of the function can be parameterized by some arguments. That is why the argument section features two sub-sections, the template arguments and the input arguments. The first one expresses constraints on the objects to be used to parameterize guards and the algorithm, whereas the second describes constraints on the actual input parameters to be fed to the function. Notice that the specification of the input arguments can be function of the template arguments.

Functions are instantiated using the `ins` operator:

`(ins` *function-name template-argument-list*`)`

Functions are invoked using the `app` (application) operator:

(app (ins *function-name template-argument-list*) *argument-list*)

Functions can invoke the execution of functions, programs or models; recursive calls are supported.

N.B.: functions of the executive API - like `inj` – are *not* function objects: they are mere addresses of entry points in the executive.

5.3 Models

Models are abstractions that describe executable operational knowledge. Replicode provides two sorts of models: forward models and inverse models. They are *both behaving like programs*, i.e. they are ruled by the same execution model. As knowledge, models are potential inputs for programs (or models) when salient enough; as executable objects, models are automatically executed when their activation value is high enough.

Forward Models They describe the consequences of actions given an initial state, i.e. they have predictive abilities. The structure of a forward model is:

```
fmd; forward model
[]
   [patterns]; template arguments section
   []; input section
      []
         []; spec of the initial state
            [patterns]; input patterns sub section
            [expressions]; timing sub section
            [expressions]; guard sub section
         []; spec of the actions
            [patterns]; input patterns sub section
            [expressions]; timing sub section
            [expressions]; guard sub section
      [expressions]; timing sub section
      [expressions]; guard sub section
   [production-sub-sections]; production section, as defined for programs
```

Like functions, forward models are templates and admit template arguments.

Forward models define timing constraints for both the initial state and the actions, as these can be complex time series, i.e. correlated events scattered over time.

The productions of a forward model are either (a) predictions – states tagged with markers (`mk.pre`) –, (b) programs that produce predictions or forward models or, (c) forward models.

Inverse Models They describe algorithms to achieve a specified target state given an initial state. The structure of an inverse model is:

```
imd; inverse model
[]
   [patterns]; template arguments section
   []; input section
      []
         []; spec of the initial state
            [patterns]; input patterns sub section
            [expressions]; timing sub section
            [expressions]; guard sub section
         []; spec of the target state
            [patterns]; input patterns sub section
            [expressions]; timing sub section
            [expressions]; guard sub section
      [expressions]; timing sub section
      [expressions]; guard sub section
   [production-sub-sections]; production section, as defined for programs
```

Like forward models, inverse models are model templates and admit template arguments.

Inverse models define timing constraints for both the initial and target states as these can be complex time series.

Inverse models can output long algorithms: for example, they can produce programs in a timely fashion (i.e. inject different code at different points in the future), thus fostering collaboration to reach the target state over an arbitrary time horizon.

5.4 Grounded Knowledge

Knowledge in Replicode is expected to be *grounded*, that is, the ontology of an object or concept is to be expressed in terms of the operational semantics of entities it relates to. In other words, ontologies are to be encoded as *models of operation*. For example, for a robot, the concept “cup” would be encoded as a collection of models expressing the fact that a cup can be grabbed, moved, etc., i.e. models expressing what the robot can *do* with a cup. This perspective is extended to perception: the concept “cup” would also contain models defining *how to* sense an instance of a cup, or what sensing a cup *does* to the robot (sensing can, for example, produce objects salient for some processes). Concepts and ontologies are thus to be defined with respect to the entity that interacts with it (e.g., the robot).

Many entities can be modeled in a system, and for example, the robot could maintain a model of human beings: in this context, drinking from a cup makes sense, whereas for the robot (which cannot drink) it does not. In this example, the concept of “cup” would be extended by the addition of the models of “drinking” that involve a cup, these models being relevant only for entities that are instances of “human being”: this relevance would be encoded as constraints (guards) on the action specification expressed by said models.

As stated in the introduction, architectures coded in Replicode are deemed to operate with incomplete knowledge. This means in practice that *not all* the possible models describing a given concept have to be given to a system. Exhaustivity is unattainable in real-world conditions and systems shall strive to acquire more knowledge while being able to display useful behaviors computed from the knowledge they have accumulated so far. Notice also that such architectures shall not rely on a hypothetical consistency of models, nor on any absolute reliability: models are to be constructed, maintained, augmented and *revised* by a system according to its own experience. Of course, developers have to give initial knowledge to bootstrap further knowledge acquisition and revision.

Replicode does not enforce knowledge grounding per se. In fact Replicode allows encoding ungrounded knowledge, that is, axioms that are not described by any models of operation. This can be the case for (a) concepts that a system does not need to understand or (b) concepts that a system does not understand *yet*, i.e. concepts under formation, i.e. the construction of operational models of which is underway.

Here is an example of an encoding of the concept of “cup” for a one-handed robot, using three models `M1`, `M2` and `M3`:

```
M1:(fmd
[]
   |[]; no template arguments
   []
      []
         []; initial state
            [
            (ptn (mk.xet (cmd grab : [x:] ::) : : t_grasp: ::)
            ; :: is a wildcard on the tail of an expression.
            ; grab is a device function, the device is unspecified (:).
            [(red x.mks [[(ptn (mk.isa x cup) |[]) [x]]])]
            )
            ]
            [(> now t_grasp)]; timing
            |[]

         []; actions
            [(ptn (mk.is_at_pos ent.hand p: ::) |[])]
            |[]
            |[]
      |[]; no timing
      |[]; no guards
```

```
      [; productions
         [|[] [(inj [(mk.pre (mk.is_at_pos x p ... now ...) 0.98 ...)])]]
      ]
   )
```

The model above states that whenever an instance of cup is grabbed, whenever the hand moves, the cup will be predictably in the same position as the hand. “cup”, “is_at_pos” are members of the ontology of the world – in this case, they are given by the programmer. “is_a” belongs to a general ontology, that is, a domain-independent ontology.

`grab` is a device function, i.e. a user-defined identifier (added to Replicode using the Replicode Extension API) meant to be interpreted by a type I module, i.e. an actuator module (the device). Markers of class `is_at_pos` are assumed to be produced by another type I module, a sensing module.

The prediction contains a confidence value (98%) - in our example, this hard-coded value accounts for the delay between the reception of the position of the hand and the production of the prediction (with respect to the motion speed); this confidence value could be estimated more finely, e.g. on the basis of actual measurements of the computation time and using an accurate model for the motion of the hand.

```
   M2:(fmd
   []
      |[]; no template arguments
      []
         []
            []
               [; initial state
               (ptn (mk.is_at_pos x: ::)
               [(red x.mks [[(ptn (mk.isa x cup) |[]) [x]]])]
               ]
               |[]
               |[]
         |[]; no timing constraints
         |[]; no guards
      [; productions
         [|[] [(inj [(mk.pre (cmd _act grasper ... now ...) ... 0.7)])]]
      ]
   )
```

It is assumed here that there exists a program `grasper` that tries to grab things whenever it can, and another program that controls the activation of the former (so that the robot does not attempt to grab things all the time for no reason). In this setup, `M2` expresses that whenever a cup is located, the grasping program will get activated, with a certainty of 70%. We shall also refine this model to make sure that the cup has not been released between the occurrences of grasp and move.

```
   M3:(imd
   []
      [(ptn delta_p: |[]) (ptn delta_t: |[])]; template arguments
      []
         []
            []
               [; initial state
               (ptn (mk.is_at_pos x: p: : : :t_initial)
               [(red x.mks [[(ptn (mk.isa x cup) |[]) [x]]])]
               ]
               |[]
               |[]

            []
               [; target state
```

```
            (ptn (mk.is_at_pos x final_p:(+ p delta_p)) : :
            deadline:(+ t_initial delta_t) ::) |[])
            ]
            |[]
            |[]
      |[]; no timing constraints
      |[]; no guards
   [; productions
      [|[] [(inj [(cmd move_object appropr_device [x final_p deadline]
...)])]]
   ]
```

`M3` is an inverse model that states that, in order to be displaced (by `delta_p`) before a deadline, then a way to do so is to send a command `move_object` with such and such parameters. In this example, `move_object` is also a user-defined function belonging to some actuator's API. It could also be replaced by the activation of a program that tries to move objects (in collaboration with the grasping program discussed in the case of `M2`).

Notice that the models `M1`, `M2` and `M3` describe not only the class “cup”, but also the semantics of the operations applicable to cups, in our case, move and grab. The point is here that models are reused: the models that describe cups also contribute to the description of operations on cups, and therefore belong also to the description of these operations, considered instances of concepts such as “move” and “grab”. Here is an example of an encoding of the concept “move” for the robot:

```
M4:(fmd
[]
   |[]; no template arguments
   []
      []
         |[]; no particular initial state
         []
            [; actions
            (ptn (cmd move_object : [x: final_p: final_t:] ::)
            [(red x.mks [[(ptn (mk.isa x cup) |[]) [x]]])]
            ]
            |[]
            |[]
      |[]; no timing constraints
      |[]; no guards
   [; productions
      [|[] [(inj [(mk.pre (mk.is_at_pos x final_p ... final_t ...) 0.9
...)])]]
   ]
)
```

`M4` predicts the final position of a cup upon reception of a `move_object` message. Notice that `M3` also describes the concept “move”, and thus belongs to both the ontology of move and the ontology of cup.

```
M5:(fmd
[]
   |[]; no template arguments
   []
      []
         []
            [; initial state
            (ptn (mk.xet (cmd : grab [x:] ::) : : t_grasp: ::)
            [(red x.mks [[(ptn (mk.isa x cup) |[]) [x]]])]
            ]
            |[]
            |[]
```

```
            []
               [; actions
               (ptn (cmd move_object : [x final_p: final_t:] : : t_move: ::)
               |[])
               ]
               |[]
               |[]
         [(> t_move t_grasp)]; timing
         |[]; no guards
      [; productions
         [|[] [(inj [(mk.pre (mk.is_at_pos x p ... final_t ...) 0.98 ...)])]]
      ]
   )
```

M5 predicts the effects of move_object on cups, i.e. a change of their location if they have been grasped. We shall also refine this model to make sure that the cup has not been released between the occurrences of grasp and move.

All these models actually target cup objects, and therefore the semantics of “move” and “grab” are limited to this sort of objects: a process that generalizes models is needed. More over, we have given these models as developers would have, i.e. in a top-down fashion. Obviously, a system would also need to generate - bottom-up - new grounded knowledge, i.e. to build semantics equivalent to the ones discussed in the examples above: generalization and concept formation.

6 Sub-systems

To organize the collaboration of a multitude of programs calls for distributing and controlling their execution at both the individual scale - single Replicode constructs - and the collective scale – from aggregates of individuals, up to the scale of sub-systems. This section presents how Replicode distributes and controls computation and knowledge.

6.1 Control

Objects in Replicode are controlled individually by a set of control values, namely, resilience, saliency and – for programs, models, and sub-systems[6] – activation.

The resilience is a delay (in milliseconds) after which, if left unchanged, the object will be permanently removed from the sub-system's memory by the executive. N.B: some objects, at the developer's convenience, can be set to live forever (see Annex 1).

The saliency is a value (in [0,1]) that controls whether an object can be an input for programs/models or not, whereas the activation is a value (also in [0,1]) that controls whether a program/model will receive inputs or not (i.e. as discussed earlier, whether a program/model is “asleep” or “awake”).

Activation and saliency are evaluated by the executive against thresholds defined in the relevant sub-system: below its respective threshold a control value is considered “off”, above “on”.

Any program or model in a given sub-system can request modifications on any control value of a particular object in said sub-system, and also on the thresholds defined by the sub-system. The executive will gather these requests and, at a certain frequency, will use the average as the final value: each sub-system specifies a particular update period (upr) for the control values of the objects it contains (exception: resilience values are updated at a period defined for *all* sub-systems). Requests for modification are expressed by the functions mod and set followed by a selector on the value and, depending on the function, either a signed increment or an absolute value.

The case of activation and saliency values is particular: in addition to being collectively

[6] Devices are also controlled by activation, although in a different way (see the sub section about the main system).

modified by programs, models, etc. - that is, on an application-dependent basis - they are also modified by the executive itself as a result of control on the collective scale (see below).

6.2 Distribution

Basically, a sub-system is a set of objects and a set of interfaces. The sub-system gathers programs, models and possibly other sub-systems that collectively perform some operation. A sub-system isolates its members from the rest of the system, i.e. it encapsulates them. It follows that the content of a sub-system is entirely hidden from the content of other sub-systems (not even accessible in read-only mode). Sub-systems can be embedded in other sub-systems.

Sub-systems communicate with each other using interfaces. An interface is a unidirectional data sink plunged into a source sub-system to collect data and pass them as inputs for the reactive objects living in a destination sub-system (this latter end of the interface is called a drain). In a given sub-system, data is sent by programs and models to the sinks (all of them) using a function called eject (`eje`) – in contrast, the inject function (`inj`) injects data in the sub-system but not in the sinks.

The visibility of a sub-system is defined as follows: the reactive objects in the destination sub-system can access the source sub-system object as an input object, but not the other way around.

To create an interface between two sub-systems, a reactive object has to be able to access said sub-systems as input objects, and therefore be situated in a third sub-system having drains from the first two. If one sub-system embeds a second one as a source, then any reactive object in the embedding sub-system can create interfaces in the second (see below). The executive provides the function `new` to create interfaces.

Interfaces are objects that are never embedded in messages, and have no resilience. Interfaces have to be destroyed explicitly (using the function `del`) by reactive objects that have visibility on the sub-systems involved in the interface.

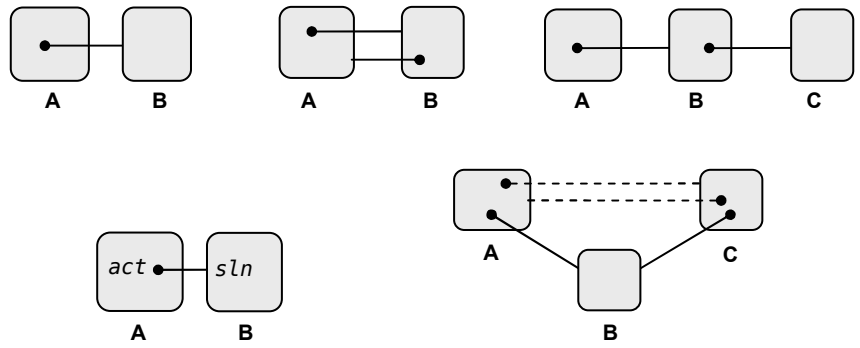

Figure 2 – Interfaces

Top left *- Two sub-systems A and B and one interface (line). Data flows from A to B (the sink is represented by the dot ending the line).*

Top middle *- Two sub-systems with two interfaces ensuring bi-directional data flow.*

Top right *- Three sub-systems. B can hold programs that filter data from A to C.*

Bottom left *– An interface is controlled at its two ends: in the source sub-system (A), by its activation value, and in the destination sub-system (B), by its saliency value. For data to flow from A to B, the activation of the interface must be higher than the activation threshold in A and the saliency of the interface must be higher than the saliency threshold in B. The sink is controlled by activation since it acts as a reactive object being passed input objects from A, while the drain is controlled by saliency, being an influx of potential inputs for the reactive objects in B. A is visible from B, but not B from A.*

Bottom right *– The sub-system C can "see" sub-systems A and B and thus, can*

create interfaces between A and B (dashed lines). A call to `eje` *issued in A would broadcast data to B and C, if the sinks are active in A, and if the drains are salient enough in B and C.*

A sub-system is said to be embedded in another if the activation of the former is controlled by the latter. An embedded sub-system cannot have interfaces to/from sub-systems that are not embedded in the same sub-system. Embedded sub-systems run on the same computing node as their embedding sub-sub-system.

A sub-system can be created in another by ejection from a destination sub-system, or by injection from the embedding sub-system itself.

Sub-systems have a resilience value that limit their life time. When a sub-system dies, so do all the interfaces that connect it to other sub-systems, and so does its content (the objects members of the sub-system and embedded sub-systems).

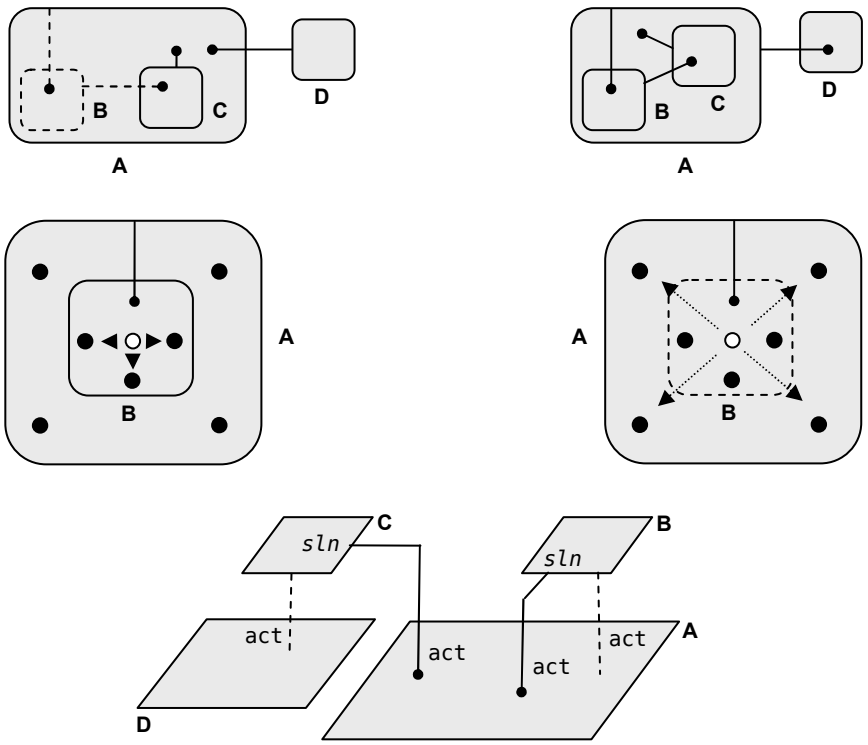

Figure 3 – Embedded Sub-systems

Top left *– Creation of an embedded sub-system. Reactive objects in D have visibility over A and can thus eject a new sub-system (B) in A, and also create interfaces between B, A and other sub-systems in A (like C). Reactive objects in C can also create sub-systems in A; this is also true for any reactive object in A.*

Top right *- Embedded sub-systems. B and C are embedded in A: their activation value is controlled by A, which is not the case for D. B and D constitute source sub-systems for A; C is a destination for A.*

Middle left *– When either the sink is not active or the drain is not salient, data (white dot) living in B can only reach reactive objects (black dots) in B. B is in a closed state.*

Middle right *– When the sink is active and the drain is salient, data living in B can reach reactive objects in A. B is in an open state. Notice the unidirectionality of the data flow.*

Bottom *– The dashed lines represent the embedding of sub-systems B and C respectively in A and D. This means that the activation of B is controlled by A, and*

the activation of C by D. The sinks are controlled by their activation in A. The drains are controlled by their saliencies, respectively in B and C.

The whole system has a similar function as a sub-system, although its structure presents noticeable differences:

- the main system *does* spread across multiple nodes in the cluster;
- the main system does not hold anything but sub-systems and modules (types I, II and V). Knowledge is held exclusively by sub-systems.
- the main system is managed by mBrane, and as such contains spaces (in mBrane terminology) that hold projections of modules or of spaces (sub-systems are implemented as mBrane modules) and a local activation threshold. Projections hold an activation value for a module projected on a space, or for a space projected on other spaces. Such module (or space) is locally active when its threshold is above the local threshold defined by the space it is projected on. There is an ultimate space called root on which other spaces are projected. A module is activated (i.e. can receive inputs) if its final activation value it above a threshold defined on root. The final activation value is computed as the disjunction (logical or) of its local activations.

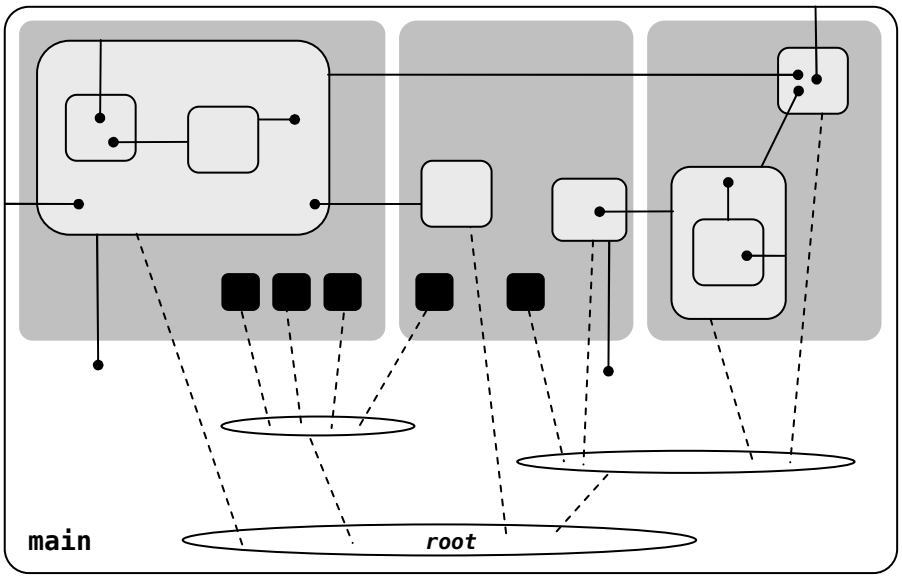

Figure 4 – An Entire System

The grey zones represent computing nodes[7], white rounded rectangles sub-systems, and black squares modules of type I, II and V. Sub-systems and modules are all projected (dashed lines) on groups (ellipses) in the main system. The main system determines the final activation of the sub-systems.

mBrane acts like the main system message bus, allowing bidirectional communication (broadcast) between modules (type I, II and V) and sub-systems.

Sub-systems communicate among themselves via interfaces. Notice that not all sub-systems have a sink in the main system. In contrast, all modules have at least one interface to the main system (not represented).

Sub-systems have semantics in the application domain, i.e. they realize a collective operation that has a specific meaning in the domain. Shall this operation need to be generalized – for example “get an object x such as y” - then using a function is in order (since a function allows template parameters) – to parameterize the sub-system’s content. This content consists in the objects created and injected in the sub-system depending on the parameters passed to the

[7] It may seem strange that the spaces in the main system do not appear to be distributed on any node. This is a mere technical detail: the spaces are managed by mBrane, and are actually replicated on each node.

function. Such a function would then constitute the sub-system's class, as it defines the construction of sub-system instances, modulo the template arguments: it is actually a member of sub-system objects – similar to a constructor in C++. Here is an example of a sub-system class - "take an object":

```
take_object:(fun
[]
   [object_kind:]; variable on a pattern
   []
       |[]; no input arguments
       |[]; timings always empty
       |[]; no guards
   [; productions
      [|[] [(inj []
         (app [(ins reach_object [object_kind]) |[])]) ...
      ]
      [|[] [(inj []
         (app (ins look_for_object [object_kind]) |[])]) ...
      ]
      [|[] [(inj []
         (app (ins reach [object_kind]) |[])]) ...
      ]
      [|[] [(inj []
         (ins (
            ctrl: pgm
            []
            |[]
            []
               [(ptn (mk.grabbed x: ::) |[])]
               |[]
               |[]
            [(|[] [(inj [(mk.taken_object x ...)])])]
         |[])
         )
      ]
   ]
)
```

`reach_object`, `look_for_objects` and `reach` are sub-system classes (not shown in the example). `object_kind` is a template argument, here a pattern specifying the object to be taken.

`grab` is a program, assumed to exist already in the system. It would usually be coded in a reactive way, that is, for example, that `grab` would attempt to grab any object within the reach of a robot's fingers.

The program `ctrl` depicted here simply notifies that the object has actually been taken when the grasper succeeds and sends an instance of `mk.grabbed`. We could need more control over the components of the sub-system, for example, to activate `grab` only when the hand is close enough to the target to avoid the grasp of any object on the path of the hand motion. In this case, we would need to inject another program P like `ctrl` to perform this additional control (activating `grab`). Here is a code for P, which would be added to the program list of the sub-system S:

```
P:(pgm
[]
   |[]
   []
      [
      (ptn (mk.position x: pos_object: : : t_object:)
      [(red [x] [[object_kind [x]]])])
      (ptn (mk.position hand pos_hand: : : t_hand:) |[])
```

```
      ]
      |[]; no timing constraints here: the positions of the object and of ..
   the hand are assumed to be monitored by other programs that will make ..
   them salient at close intervals.
      [(< (dis pos_object pos_hand) 20)]; dis is the "distance" operator
   [
      |[] [(set [grab.act  1])]; set the activation value of the ..
   grab program.
   ]
)
```

Notice the check of `x` matching the pattern `object_kind` using the `red` operator.

As discussed in section 6.1, the call to the function `set` is a request to the executive for assigning a new activation value to the program grab in the sub-system. This will take effect after a delay bounded by the update period defined for the sub-system containing `grab` (omitted in the example).

Finally, sub-systems offer some additional facilities for handling activation and saliency values (see Annex1 for details). Here is a descriptions of these features, discussed for activation values, knowing that the features are also proposed for saliency values in the same way:

- A sub-system features a parameter (auto-eject) that, when set to "on", automatically ejects objects having (re)gained activation (i.e. activation above the threshold).
- A sub-system can be parameterized so that, when inactive, it gets better chances to be reactivated when a sufficient amount of its members have gained enough activation. A sub-system specifies a control value called out-of-range, a decrease/increase for the activation and the activation threshold of the sub-system, and a relaxation policy. When a member of the sub-system gains activation such as its value is above the threshold of the sub-system and also above the average activation value plus the out-of-range value, then the activation threshold of sub-system is decreased/increased and the sub-system's own activation is also decreased/increased. The relaxation policy defines the way the affected value (sub-system's threshold or activation) returns to its original value. For example, a relaxation policy can express: "bring the activation value of the sub-system to its original value over 2000ms, using intervals of 100ms", i.e. if the activation value was boosted by 80%, then, each 100ms, it would be decreased by 4% of its boosted value.
- A sub-system can reduce the activation values of its members over time: one does so by defining a decay for the activation values as a percentage of their initial values, to be reached incrementally over a given period. For example, a sub-system can specify decays in a way such as: "bring the activation values down to 80% of their respective initial values over a period of 2000ms, using intervals of 100ms", i.e. every 100ms, the values will be decreased by 4% of their respective initial values.
- It is possible to parameterize a sub-system to have it delete members whose activation value falls below a deletion threshold (not to be confused with the activation threshold).

Increasing the saliency or activation of an object in a sub-system in a significant way (above the average plus out-of-range) can trigger chain-reactions of increased saliencies or activations across different objects semantically related to the object from which the reaction originated. This capability can be leveraged to build associative memories across sub-systems. Propagation across sub-systems does however require either explicit ejection of salient/active data or enabling the auto-eject functionality.

7 Executive

This section describes the sub-systems that constitute the executive: these are the rCore (already presented in the overview) and the rMem – the memory. The latter is a repository of the knowledge of the system and is responsible in the main for making available salient knowledge for programs and models in a given sub-system.

A detailed specification of the executive is given in Annex 2.

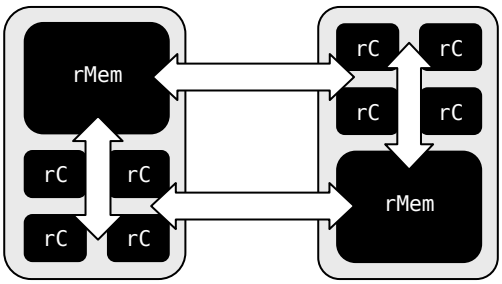

Figure 5 – Overview of the Executive

Two sub-systems are represented, each hosting a population of rCores and one rMem. rMems act as message repositories: they hold the history of messages sent to the sub-system (or created within), deallocate objects when their resilience drops down to zero, maintain activations and saliencies, pass salient objects as inputs to programs and models (vertical arrows). rCores are composed by modules of type III, rMems are modules of type IV.

Inter-sub-system communication is achieved by means of interfaces (horizontal arrows).

N.B.: a sub-system cannot spread across multiple computing nodes, i.e. it is hosted by only one node in the system (except the main system). This also holds for all its embedded sub-systems (hosted by the same node as the embedding sub-system).

7.1 rCore

rCores are essentially responsible for the details of program/model execution. As described in section 4.2, they manage offsprings of programs (and models) when overlapping message patterns come as inputs.

On demand, rCores can also perform the periodic triggering of reactive objects. To enable this feature, the member `sig` in a reactive object shall be set to 1: if the object is in position to execute the code defined in its production section, this execution will be synchronized with an internal signal, i.e. will be delayed until the next signaling period has elapsed. The signaling period (`spr`) is defined for each sub-system.

The execution of reactive objects has the purpose of changing the state of sub-systems. rCores provide functions to do so, performing the following main operations:

- injection: the function `inj` creates new objects and broadcasts them to any rCore in the sub-system. When the injected object is a program or a model, a new rCore is dynamically created to run said object (but suspended until its activation gets high enough).
- ejection: the function eje copies salient objects to any sub-system having a sink in the sub-system hosting the rCore. Technically, if the sub-systems are on the same node, objects are shared – only pointers are copied.
- creation: the function `new` allows creating a new object in a sub-system. This function cannot be called directly by user code: it has to use inject or eject instead – exception: interfaces and marker classes.
- deletion: the function `del` deletes permanently an object from the sub-system. This function has the same effect than setting a resilience value to 0. This function cannot be called directly by user code to kill objects: it has to modify the resilience instead – exception: interfaces and marker classes.
- backward chaining: the function `bwd` takes a target state as an argument and executes inverse models able to produce said state from the current state (i.e. the state of the system at the time `bwd` is invoked). The function is also passed a timeout argument to limit its search time, and a depth argument to limit the depth of the search. `bwd` searches the *salient* knowledge backwards, i.e.

performs *breadth-first goal regression*. Salient inverse models are analyzed at the level of the input section specifying their target state. When the target state defined by a model matches the goal, then a new goal is instantiated that corresponds to the input section specifying the model's initial state; if the initial state matches the current state of the system, `bwd` stops analyzing the model furthermore and outputs the productions of the model, i.e. its actions. The depth parameter specifies the maximum number of jumps backwards `bwd` shall perform for each of the models analyzed. Last, `bwd` is also passed a sub-system where to store the found solutions - `bwd` operates within the boundaries of a sub-system. Notice that the products of `bwd` (goals and actions) are marked as hypotheses referencing the process representing the execution of one invocation of `bwd`; this process is attached a simulation marker (`mk.sim`). The simulation marker refers to a simulation run (`smr`) created by the rCore upon invocation of `bwd`.

- forward chaining: the function `fwd` takes salient inverse models as an argument and computes their outcome, marked as hypotheses referencing the process representing the execution of one invocation of `fwd`, as in the case of `bwd`. `fwd` performs breadth-first exploration, starting with the models whose specification of the initial state match the system's current state. Like `bwd`, `fwd` is passed a depth and timeout arguments and also operates within the boundaries of a given sub-system. Simulation markers and simulation runs are created for each invocation of `fwd` (as for `bwd`).
- notification: rMems and rCores can notify programs of important events related to the production of objects, their death or the modification of their control values. The function `ntf` allow programs to turn on or off some of the notifications (see below).

Besides serving calls to the functions mentioned above, the rCores provide notifications (as regular objects sent across the system) of the key operations they perform. As a general rule, any invocation of functions of the executive triggers the automatic injection of process objects (`mk.pro`) referencing command objects (`cmd`). In particular, each time a program or a model produces objects from input objects, process objects are injected by the rCore, and these are in turn marked by `xet` markers by the rMem. These notifications include the starting time, the program that spawned the process, the program/model that is executed, the input objects that triggered the production, the termination time, duration and the resulting productions.

Sending such fine-grained information at high frequencies can bring a system down to its knees. To prevent this problem, Replicode allows some notifications to be turned on or off per program/model, depending on the performance constraints and on the importance of the objects they refer to, or of the programs responsible for the events. This decision is application-dependent.

7.2 rMem

An rMem is the memory of a given sub-system as seen by the programs/models hosted by said sub-system. rMems perform the following continuous tasks:

- modification of control values: as described in section 6.1, rMems collect requests for modifying control values and assigns them a final value to be used sub-system-wide. These final values are computed as averages of the requested values. Updates of saliency and activation values are performed at periods (`upr`) defined individually for each sub-system; updates of resilience values are performed at a period defined globally for *all* sub-systems. Values subjected to this treatment by the rMems are called *mediated values*.
- monitoring of the activation and saliency values in the sub-system:
  - when appropriate, apply the boost, relaxation and decay on control values and on thresholds.
  - whenever objects gain saliency, present them as inputs to any active program/model in the sub-system; eject these objects in case the auto-eject feature is activated for the sub-system.

Like rCores, rMems also inject runtime notifications (possibly enabled or disabled by programs using the `ntf` function discussed above), essentially:

- control value modifications: whenever a program requests a modification on another object's control value, a message is sent to programs carrying a process marker referring to a command object: `(set/mod` *arguments* `...)`.
- activation state transition: when reactive objects are getting activated or deactivated, programs are notified by a message carrying a process marker referring to a command object: `(_act [`*target-object* `0/1] ...)`. Similar notifications are triggered by saliency changes on any object (function `_sln` instead of `_act`).
- end of execution: when a program terminates another one[8], or when a program completes its computation, a message is sent carrying an end-of-execution marker referring to the process `(mk.xet` *terminated-process* `...)`. The marker indicates the duration of the process and its productions. This applies also to the execution of device functions and models.
- garbage collection: just before the destruction of an object (i.e. when its resilience drops down to 0), programs/models are notified by a message of this imminent death (with a grace period equal to the resilience update period) to give them a chance to react. If the resilience remains at 0, the object is destroyed, unless some other objects hold a reference to it. In the latter case, the object will remain in memory but will never constitute inputs for programs/models anymore. It will however still be visible in patterns targeting objects holding a reference to it.

Notice that Replicode does not define any queries on rMems - the execution model of Replicode is strictly data-driven.

References

Nivel E. (2007). Ikon Flux v2.0. Reykjavik University Department of Computer Science, Technical Report RUTR-CS07006.

Nivel E. and Thorisson K. (2009). Self-Programming: Operationalizing Autonomy. *Proc. of the Second Conference on Artificial General Intelligence*, 150-155.

Paun G. (2002). *Membrane Computing. An Introduction*. Springer-Verlag, Berlin.

Pezzulo G. and Calvi G. (2007). Designing Modular Architectures in the Framework AKIRA. *Multi-agent and Grid Systems, 3,* 65-86.

Tschudin C. (2003). Fraglets - a Metabolistic Execution Model for Communication Protocols, *Proc. Of the 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*.

Yamamoto L., Schreckling D., Meyer T. (2007) Self-Replicating and Self-Modifying Programs in Fraglets. *Proc. of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*.

[8] by bringing its resilience to 0; suspending a process is achieved by lowering the activation below the sub-system's threshold, and notification occurs as described above.

Annex 1 – Replicode Specification

1 Overview

This annex defines the *syntax* of Replicode. See Annex 2 for a specification of the *behavior* of objects encoded in Replicode.

Code in Replicode is written using seven categories of syntactic elements:

- **Atoms**: constants, object class members, labels, variables, references, numbers, and numerical identifiers.
- **Separators**: comments, separators in sets and expressions, line continuation.
- **Sets**: sets of atoms and/or sets and/or expressions.
- **Expressions**: an operator followed by operands, or an object class followed by constructs that implement its structure. Examples of expressions are (respectively), (+ a b), an expression that computes the sum of two variables a and b, and (mk.isa a b), an expression denoting an object of class mk.isa followed by two members a and b (meaning b is an instance of a).
- **Directives**: instructions to specify how the code translator shall handle code. Directives are not subject to rewriting, they are commands defined by the pre-processor, interpreted at translation time, not at runtime. The pre-processor is similar to the C++ pre-processor.
- **Operators**: symbols defining operations to be performed on one or more symbols; an operator followed by such symbols (operands) constitutes an expression. Operators are implemented in the executive and their semantics cannot be modified at runtime. The list of available operators can be extended, i.e. operators can be user-defined. Operators fail gracefully when applied to ill-formed or unsuitable operands.
- **Object Classes**: symbols defining *static* types. A class defines a specific structure and its (axiomatic) semantics. For example, programs, models, functions, groups, markers are object classes. Objects classes can be user-defined.

Operators and objects types are represented internally (in r-code) by numerical *opcodes*. However, for the sake of readability, when these elements are built-in the executive, they are identified by tri- /tetragrams and/or by alternate non numerical symbols (aliases).

2 Syntax

2.1 Atoms

Symbol	*Description*
`nil`	constant meaning “nothing”, polysemic, i.e. its meaning has to be interpreted in the context of an embedding expression. For example, `nil` can be interpreted as false, not a number, empty set, etc.
`[]`	opcode of sets.
`|[]`	constant identifying an empty set. `[ ]` is illegal. `[nil]` is a set containing one element.
`|nb`	constant for NaN.
`|ms`	constant meaning “undefined time”.
`|sid`	constant meaning “undefined sub-system identifier”.
`|did`	constant meaning “undefined device identifier”.
`|fid`	constant meaning “undefined device function identifier”.

`|bl`	constant meaning “undefined Boolean value”.
|st	constant meaning “undefined character string”.
`true`	constant.
`false`	constant.
a-name`:`*an-expression*	declaration of a label identifying the expression. *a-name* is an alphanumerical identifier. Notice the absence of a separator before and after `:`.
a-name`:`	declaration of a named variable inside a pattern. *a-name* is an alphanumerical identifier.
a-name	reference to the value identified by a label or by a named variable. *a-name* is an alphanumerical identifier.
`:`	declaration of an unnamed variable (wildcard) inside a pattern.
`::`	declaration of an unnamed variable inside a pattern. It stands for the tail of an expression starting at the position of `::` in the pattern.
a-floating-point-number	a floating point number on 32 bits; also used to represent integers (lesser than or equal to 2^{24}).
a-timestamp	an unsigned integer on 64 bits (milliseconds since 01/01/1970).
“*a-character-string*”	character string (8 bits per character).
`this`	reference to a reactive object in the code of the object itself (like in C++).
`this.`*a-name*	reference to an object identified by the variable *a-name* in the code of the object itself.
`this.`*a-member*	reference to a value identified by the reactive object’s member *a-member* in the code of the object itself.
a-name`.`*a-member*	reference to a value identified by the member *a-member* of an object or a set referred to by the variable *a-name* – in other word, direct member access. Recursion of member access is permitted (ex: *a-name*`.`*a-member*`.`*a-member*).
a-member	a numerical identifier indexing a member in an object’s structure.
opcode	an operator or an object/marker class.
identifier	a integral value identifying a device, a device function or a sub-system. Such values are typically defined as macros mapping symbols into r-code constructs.

Replicode is case-sensitive.

N.B.: `|` is not an operator: it is a mere character in some symbols.

Atoms of types named variable and label are literals encoded as null-terminated character strings of unlimited length, obeying the following rules:

- each character is in the US-ASCII subset: A, …, Z, a, …, z, 0, …, 9, _

- the first character cannot be a digit or an underscore: the latter is reserved for label and variable names decoded from dynamically generated code (see section 2.2 in Annex 3).

- the string cannot be equal to any of the symbols denoting atoms, built-in operators, built-in object types, built-in device functions, and built-in macros.

Atoms of type character string are limited in length (1020 characters) and can be composed of any character.

Atoms of types floating point number and timestamp are literals which follow the syntactic rules defined by the C++03 standard.

2.2 Separators

Syntax	*Description*
blank-space	separator between atoms, expressions, sets.
carriage-return	separator between atoms, expressions, sets. The actual character is OS dependent (e.g. LF on Linux).
;	comment, extending to rest of the line.
blank-space..*carriage-return*	line continuation. Can only appear in the commented part of a line, not within code.

Tab characters are illegal (see section 2.5 below).

In the syntax described thereafter, blank spaces and carriage returns denote separators between atoms, expressions, sets and are used interchangeably.

2.3 Sets

Syntax	*Description*
[*list-of-elements*]	a set.
list-of-elements	list of atoms and/or sets and/or of expressions, separated by exactly one separator from each other.

2.4 Expressions

Syntax	*Description*
(*opcode list-of-elements*)	an expression. N.B.: parentheses are useless since the arity of the opcodes is known, but are nevertheless part of the language for improved readability – they are not optional.
list-of-elements	list of atoms and/or sets and/or of expressions separated by exactly one separator from each other.

2.5 Code Formatting

To avoid the infestation of code by parentheses, Replicode allows using a carriage return followed by an indent (a fixed number of spaces) instead of an opening parenthesis, and a carriage return followed by a back-indent instead of a closing one. Thus,

carriage_return indent *list-of-elements* carriage_return back-indent

is equivalent to

(*list-of-elements*).

This formatting style can be mixed with parentheses at will, provided that, when made explicit, parentheses are matched explicitly.

In a similar way, [] (set opcode) followed by a carriage return and an indent opens a set, and a carriage return and back-indent closes it (see examples below).

Tab characters ('\t') are illegal - when using a text editor, one shall set it to convert tab characters into a number of spaces.

In the following examples we assume indents to be 3 blank spaces:

```
(opcode arg1 arg2 arg3)
```

can be written:

```
(opcode arg1 arg2
arg3)
```

or:

```
(opcode arg1
arg2
arg3)
```

or:

```
(opcode
arg1
```

```
arg2
arg3)
```
or:
```
   opcode arg1 arg2
; notice the mandatory carriage return + back indent just before..
this comment.

(opcode1 arg1 (opcode2 arg4 arg5) arg3)
```
can be written:
```
(opcode1 arg1
(opcode2 arg4 arg5)
arg3)
```
or:
```
(opcode1 arg1
   opcode2 arg4 arg5
arg3)
```
or:
```
(opcode1 arg1
   opcode2
   arg4
   arg5
arg3)
```
or:
```
(opcode1
arg1
   opcode2
   arg4
   arg5
arg3)

(opcode1 arg1 [obj1 obj2 obj3] arg2)
```
can be written:
```
   opcode1 arg1
   [obj1 obj2 obj3]
   arg2
; mandatory carriage return + back indent just before this comment.
```
or:
```
   opcode1 arg1 [
   obj1
   obj2
   obj3
   ]
   arg2
; mandatory carriage return + back indent just before this comment.
```
or:
```
   opcode1 arg1
   []
      obj1
      obj2
      obj3
   arg2
; mandatory carriage return + back indent just before this comment.
```
or:

```
   opcode1 arg1
   [
   obj1
   obj2
   obj3
   ]
   arg2
; mandatory carriage return + back indent just before this comment.
```
or:
```
   opcode1 arg1 []
      obj1
      obj2
      obj3
   arg2
; mandatory carriage return + back indent just before this comment.
```
or:
```
   opcode1 arg1 [
   obj1
   obj2
   obj3
   ]
   arg3
; mandatory carriage return + back indent just before this comment.

(opcode1 arg1 [obj1 obj2 obj3])
```
can be written:
```
   opcode1 arg1
   [obj1 obj2 obj3]
; mandatory carriage return + back indent just before this comment.
```
or:
```
   opcode1 arg1
   []
      obj1
      obj2
      obj3
   ; mandatory carriage return + back indent just before this comment.
; mandatory carriage return + back indent just before this comment.
```
or:
```
   opcode1 arg1
   []
      obj1
      obj2
      obj3
; mandatory carriage return + 2 back indents just before this comment.
```
or:
```
   opcode1 arg1 []
      obj1
      obj2
      obj3
   ; mandatory carriage return + back indents just before this comment.
; mandatory carriage return + back indents just before this comment.
```
or:

```
   opcode1 arg1 []
      obj1
      obj2
      obj3
; mandatory carriage return + 2 back indents just before this comment.

(opcode1 arg1 (opcode2 arg2 [obj1 obj2]) arg3)
```

can be written:

```
   opcode1 arg1
      opcode2 arg2 []
         obj1
         obj2
   arg3; notice the 2 back indents before arg3.
; mandatory carriage return + back indent just before this comment.
```

2.6 Pre-processor Directives

Element	*Syntax*	*Description*
Literal	A null-terminated character string of unlimited length, each character is in the US-ASCII subset: A, …, Z, a, …, z, 0, …, 9, _, |.	The | character is allowed only by explicit mention. In such cases, it must appear once and be the first character of the literal. Additional rules apply to specific literals, depending on their semantics.
Counter directive	`!counter` *counter-name initial-value*	Defines a counter that will be post-incremented each time it referenced, i.e. each time a line containing its name is evaluated by the pre-processor (except when used as an initial value). Counters are encoded on 16 bits.
counter-name	A literal starting with two underscores.	A counter name is replaced by its value in floating point format (even if it represents an integer).
initial-value	An integer (>=0) or a counter name.	If the initial value is given as the name of an existing counter, said counter is not incremented.
Macro directive	`!def` *macro-name macro-replacement*	Defines a macro identified by the literal *macro-name* (like `#define` in C++).
macro-name	A literal, possibly starting with |.	
Macro-expression directive	`!def` *macro-expression macro-replacement*	Defines a macro expression (like `#define` in C++ when a macro has arguments).
macro-expression	(*macro-name list-of-variables*)	
variable	:*variable-name*	

variable-name	A literal.	
macro-replacement	An expression. A set. An atom. *a-development*	
development	{*list-of-elements*}	
element	An expression. A set. An atom. A macro name. An instantiated macro expression. A counter name.	
Instantiated macro expression	(*macro-name list-of-elements*)	When *macro-name* is the name of a macro expression, the code (`macro`-*name list-of-elements*) is replaced by the replacement defined in the macro expression, where the variables are replaced by the corresponding actual arguments.
Undefine directive	`!undef` *a-name*	Un-defines a macro, a counter or a macro expression (like `#undef` in C++).
a-name	A macro name. A counter name.	

Positive precondition directive	`!ifdef` *a-name*
	list-of-directives
	`!else`
	list-of-directives
	`!endif`
		Condition on the existence of a macro, a macro expression, or a counter (like `#ifdef #else #endif` in C++). The `else` part is optional.

Negative precondition directive	`!ifundef` *a-name*
	list-of-directives
	`!else`
	list-of-directives
	`!endif`
		Condition on the absence of a macro, a macro expression, or a counter (like `#ifndef #else #endif` in C++). The `else` part is optional.

Class directive	`!class` (*class-name list-of-member-declaration*)	Exposes the structure of an object class. This directive define the identifiers of the members of a given class, to be used as direct references to parts of an object, instead of declaring variables on said object in a pattern. This refers to the ability to encode atoms like *an-object*.*a-member-name*. See section

		4.2.1 of this Annex for details.
class-name	A literal.	
	a-literal`[]`	Defines a class having the structure of a set.
member-declaration	*member-name*`:`*type*	
	`:`*variable-name*	A class featuring members declared as variable is a class template. See section 4.2.1 of this Annex for details.
	`nil`	Only allowed in class template instantiations. See section 4.2.1 of this Annex for details.
member-name	A literal.	
type	`nb`	Number.
	`~nb`	Mediated number.
	`ms`	Timestamp.
	`~ms`	Mediated timestamp.
	`bl`	Boolean value.
	`sid`	Sub-system identifier.
	`fid`	Device function identifier.
	`did`	Device identifier.
	`[]`	Set.
	`st`	Character string.
	A class name.	
	`[`*list-of-member-declarations*`]`	This type corresponds to an anonymous class having the structure of a set, and for which members are declared.
	`[`*iterative-member-declaration*`]`	This type corresponds to an anonymous class having the structure of a set, for which all elements are of the specified type.
	no-character	A member declared this way is of any possible type.
iterative-member-declaration	*member-name*`::`*type*	
Forward class directive	`!class` *class-name*	Declares the existence of a class without defining its structure. This allows defining classes with circular dependencies.
Class template instantiation	`(`*class-template list-of-member-declarations*`)`	Expands into a structure where the variables are replaced piece-wise by the member declarations passed as actual arguments. See section 4.2.1 of this Annex for details.
Operator directive	`!op (`*operator-name list-of-anonymous-member-*	Exposes the signature of an operator, i.e. the type of its

	declarations`):`*type*	arguments and return value. Overloads are not allowed.
anonymous-member-declaration	`:`*type*	
Device function directive	`!dfn (`*function-name list-of-anonymous-member-declarations*`)`	Exposes the signature of a device function.
Load directive	`!load` *file-locator*	Loads code from a file (like `#include` in C++), in Replicode for r-code form, depending on *file-locator*. Code loaded this way will be loaded only once even if several `load` directives targeting the same file are interpreted (behavior equivalent to `#pragma once` in C++).
file-locator	A literal.	Path to a file relative to the directory where the file containing the directive is located. The syntax follows the UNIX style. When the file locator ends with the string ".replicode", source code is loaded, when it ends with ".rcode", binary code is loaded.

Replicode provides a set of predefined macro definitions gathered in a file named `std.replicode`, given in section 5 of this Annex. In particular, this file contains the definitions of device identifiers and device function identifiers.

2.7 Code Structure

Code is written in text streams and contains, in any order:

- comments,
- pre-processor directives,
- definitions of objects,
- calls to functions of the executive.

Calls to functions of the executive (e.g. `_ldc`) have the purpose of initializing an application by instantiating objects in the main system. These function calls are called thereafter initialization commands.

The usage of object definitions / macros must occur after the definitions – except in the case of forward class declarations.

The stream is parsed from top to bottom, left to right.

3 Summary of Built-in constructs

3.1 Operators

Opcode	*Alias*	*Description*
`now`		current time
`equ`	`=`	equality check.

neq	<>	non equality check.
gtr	>	greater than.
lsr	<	lesser than.
gte	>=	greater or equal.
lse	<=	lesser or equal.
add	+	addition.
sub	-	subtraction.
mul	*	multiplication.
div	/	division.
dis		distance (norm of a difference) between two numbers.
ln		Neperian logarithm.
exp		elevation of e to a power.
log		logarithm in base 10.
e10		elevation of 10 to a power.
syn	\	keeps an expression or a set as raw syntax, i.e. prevents the evaluation of the expression or of the elements of the set.
ins		instantiation of a function, program, model or goal, i.e. valuating the formal template arguments.
app		application of a function to some arguments. The function must be instantiated.
red		scans an expression or a list for objects matching a pattern, returns a list of matching objects.
|red		same as red, but returns a list of objects that do *not* match the pattern.
com		returns the intersection of two sets.
spl		split a set in two according to a set of patterns.
mrg		merges two sets into one.
ptc		patch expressions, i.e. searches and replaces sub-expressions in an embedding one, using a set of pattern and an substitutions.
rpl		replaces elements in a set by substitutes when they match a set of patterns.

Notice that operators are polymorphic, i.e. they are defined for any relevant argument types and behave accordingly.

When users add new object types they also have to add new variants of the existing operators bearing the same semantics (e.g. add for vectors, a variant of add for numbers, both having the semantics of an addition).

3.2 Object Classes

Opcode	*Description*
ptn	pattern. Patterns are expressions containing variables, followed by a set of expressions encoding conditions on the variables. An expression will match if it has the same structure as defined in the pattern and if all of the conditions are met. The opcode ptn is omitted in the definition of patterns except when held by a system object.
|ptn	anti-pattern. Same structure as a pattern, but an expression will be considering matching the anti-pattern if it would not match the pattern (i.e. without the | alteration) – not having the same structure or not satisfying at least one condition. The sub-symbol ptn is omitted in the definition of patterns except when held by a system object.

`pgm`	program.
`|pgm`	anti-program. Behaves as a program, but reacts when no objects match the pattern within a period of time.
`fun`	function.
`fmd`	forward model.
`imd`	inverse model.
`sys`	sub-system.
`int`	interface.
`gol`	goal, i.e. a goal to reach a target state.
`|gol`	anti-goal. A goal not to reach a target state.
`smr`	simulation run.
`cmd`	command to a device, i.e. invocation of a device function.
`dev`	device.
`mk.`*class-name*	marker of class *class-name*.

There also exist so-called internal classes, not mentioned in the list above (see section 4.2 of this Annex for details).

3.3 Devices and Device Functions

Device	*Function (alias)*	*Description*
`exe`		the executive.
	`inj`	injects a new object inside a sub-system.
	`eje`	ejects a new object in all interfaces referencing as a sink the sub-system where `eje` is invoked.
	`rep`	replaces the code currently executed by an rCore by another one.
	`mod`	requests the modification of one of the various control values and thresholds.
	`set`	requests the assignment of one of the various control values and thresholds.
	`new_class`	creates a new object or marker class.
	`new_int`	creates a new interface.
	`del_class`	deletes an object or marker class.
	`del_int`	deletes an interface.
	`bwd`	performs goal-regression to find inverse models able to reach a goal given an initial state.
	`fwd`	performs forward exploration to predict the outcome of some actions, given an initial state.
	`ldc`	loads code from a text input stream (Replicode form) or from a binary input stream (r-code form).
	`swp`	swaps code to/from disk (in r-code form).
	`ntf`	sets on or off one of the notification types.
	`new_sys`	instantiates a sub-system in a computing node.
	`del_sys`	deletes a sub-system from the main system.
	`new_obj`	instantiates an object in a sub-system.
	`new_dev`	instantiates a device in a computing node.
	`del_dev`	deletes a device from the main system.
	`new_spc`	creates a space in the main system.
	`del_spc`	deletes a space from the main system.
	`new_prj`	projects a sub-system or a device onto a space in the main

	system.
del_prj	un-projects a sub-system or a device from a space in the main system.
mod_act	modifies the activation of a sub-system or of a device in a space in the main system.
mod_act_thr	modifies the activation threshold of a space in the main system.
_act	activates or deactivates an object. For notification purposes. Cannot be called by reactive objects.
_sln	makes an object salient or not. For notification purposes. Cannot be called by reactive objects.
_del	deletes an object from memory. For notification purposes. Cannot be called by reactive objects.
_start	starts the application.
_suspend	suspends the application.
_resume	resumes the application.
_stop	terminates the application.

N.B.: device and function identifiers are actually macro definitions.

3.4 Marker Classes

Opcode	*Description*
mk.pro	instantiation of a process.
mk.xet	termination (or halting) of a process.
mk.pre	prediction.
mk.hyp	hypothesis.
mk.sim	simulation.
mk.isa	instance relationship (“is a”).

3.5 Entities

The following entities are entities already injected in the system before user code is loaded.

Entity	*Description*
self	the application, i.e. the system as an actor in the world.

3.6 Sub-systems

Sub-system	*Description*
main	the main sub-system object.

4 Semantics

This section describes the semantics of the built-in constructs, i.e. the meaning of the elements they are composed of.

The built-in functions are exposed by the executive and therefore their semantics is specified in Annex 2.

4.1 Operators

In this section, operator aliases (when defined) are used in the syntactic definitions instead of the opcodes. Operator signatures technically do not require member names. However for the sake of clarity these have been added (in italics) in some of the descriptions that follow – they

constitute a simple convenience, and are not part of the syntax. Some operators admit variations on their argument types, i.e. are overloaded: even if overloads are not formally allowed, the signatures of such overloads are given when appropriate for readability.

When passed illegal arguments, operators return `nil`, `|nb`, `|ms`, `|[]`, `|bl`, depending on their return type.

`_now`

Signature

```
!op (_now):ms
```

Description

Returns the current time (in milliseconds) since 01/01/1970.

Return value

A timestamp representing the current time (in milliseconds) since 01/01/1970.

`equ`

Signature

```
!op (= : :):bl
```

Description

Checks for the equality of the two arguments.

Return value

In the case where none of the arguments are `nil` or undefined values, `equ` returns `true` if the two arguments are equal, `false` otherwise.

`equ` returns `true` when both arguments are undefined values, regardless of their type.

`equ` returns `true` when one of the arguments is an undefined value, regardless of its type, and the other is `nil`.

`equ` returns `true` when both arguments are `nil`.

`neq`

Signature

```
!op (<> : :):bl
```

Description

Checks for the inequality of the two arguments.

Return value

In the case where none of the arguments are `nil` or undefined values, `neq` returns `true` if the two arguments are different, `false` otherwise.

`neq` behaves as the negation of `equ` in the case of arguments being `nil` or undefined values.

`gtr`

Signature

```
!op (> : :):bl
```

Description

Checks for the first argument being greater than the second.

Return value

`true` if the first argument is greater than the second, `false` otherwise.

`lsr`

Signature

```
!op (< : :):bl
```

Description
Checks for the first argument being lesser than the second.

Return value
`true` if the first argument is lesser than the second, `false` otherwise.

`gte`

Signature
```
!op (>= : :):bl
```

Description
Checks for the first argument being greater than or equal to the second.

Return value
`true` if the first argument is greater than or equal to the second, `false` otherwise.

`lse`

Signature
```
!op (<= : :):bl
```

Description
Checks for the first argument being lesser than or equal to the second.

Return value
`true` if the first argument is lesser than or equal to the second, `false` otherwise.

`add`

Signature
```
!op (+ : :):
```

Overloads
```
!op (+ :nb :nb):nb
!op (+ :nb :ms):ms
!op (+ :ms :nb):ms
```

Description
Adds the two arguments.

Return value
A number or a timestamp representing the sum of the two arguments.

`sub`

Signature
```
!op (- : :):
```

Overloads
```
!op (- :nb :nb):nb
!op (- :ms :nb):ms
!op (- :ms :ms):nb
```

Description
Subtract the second argument from the first.

Return value
A number or a timestamp representing the difference between the first argument and the

second.

mul

Signature

```
!op (* :nb :nb):nb
```

Description

Multiplies the two arguments.

Return value

A number representing the product of the two arguments.

div

Signature

```
!op (/ :nb :nb):nb
```

Description

Divides the first argument by the second.

Return value

A number representing the division of the first argument by the second. If the latter is 0, `|nb` is returned.

dis

Signature

```
!op (dis : :):nb
```

Description

Computes the norm of the difference of two arguments.

Return value

A number representing the norm of the difference of two arguments.

ln

Signature

```
!op (ln :nb):nb
```

Description

Computes the Neperian logarithm of the argument.

Return value

A number representing the Neperian logarithm of the argument. If the argument is 0, `|nil` is returned.

exp

Signature

```
!op (exp :nb):nb
```

Description

Computes the exponent of the argument.

Return value

A number representing the exponent of the argument.

log

Signature

`!op (log :nb):nb`

Description

Computes the logarithm of the argument in base 10. If the argument is 0, `|nil` is returned.

Return value

A number representing the logarithm of the argument in base 10.

`e10`

Signature

`!op (e10 :nb):nb`

Description

Computes 10 at the power of the argument.

Return value

A number representing 10 at the power of the argument.

`syn`

Signature

`!op (\ :):`

Description

Prevents the evaluation of the argument. If the argument is a set, the evaluation of its elements is prevented. When a program/model injects an object defined like `(\ (opcode operands))` then what is instantiated in memory is `(opcode operands)`. This rule applies to any expressions and atoms.

Return value

The argument in syntactic form.

`ins`

Signature

`!op (ins` *object*`:` *arguments*`:[]):`

Description

Instantiates an object featuring a template argument list with a list of actual arguments. The latter comes as a set (with list semantics, i.e. the order of its elements matters).

Return value

An instantiated object. The set of template arguments in the object is filled with the actual arguments.

`app`

Signature

`!op (app` *function*`:` *arguments*`:[]):`

Description

Calls a function that has been instantiated, with the arguments specified in the set (with list semantics, i.e. but the order of its elements matters).

Return value

A set of productions..

`red`

Signature

`!op (red` *input-set*`:[]` *production-section*`:[]):[]`

production-section is a set containing a set of production sub-sections. A production sub-section is (a) one pattern and, (b) a set of productions. A production is an expression, function of the element in the input set.

Description

Fills a set with productions computed as functions of the elements in the input set that match the patterns in the production section.

For each element in the first argument, if this element matches all the patterns defined in the second argument, adds the corresponding production to the result set.

Return value

A set containing the elements of the first argument matching the pattern(s) in the second argument.

The resulting set may contain duplicates depending on the patterns and associated productions. Such duplicates are eliminated.

`|red`

Signature

`!op (|red` *input-set*`:[]` *production-section*`:[]):[]`

production-section is defined as for `red`.

Description

Fills a set with productions computed as functions of the elements in the input set that do not match the patterns in the production section.

For each element in the first argument, if this element does not match all the patterns defined in the second argument, adds the corresponding production to the result set.

Return value

A set containing the elements of the first argument not matching at least one pattern in the second argument.

The resulting set may contain duplicates depending on the patterns and associated productions. Such duplicates are eliminated.

`com`

Signature

`!op (com :[] :[]):[]`

Description

Fills a set with the common elements of two sets.

Return value

A set resulting from the intersection of two sets.

The resulting set may contain duplicates depending on the content of the two input sets. Such duplicates are eliminated.

`spl`

Signature

`!op (spl` *input-set*`:[]` *set-of-patterns*`:[]):[]`

Description

Splits a set in two according to the patterns.

Return value

A set containing two sets: the first contains the elements of the input set that match the patterns,

the second, the elements that don't.

mrg

Signature

`!op (mrg :[] :[]):[]`

Description

Merges two sets.

Return value

A set containing the elements of both sets.

The resulting set may contain duplicates depending on the content of the two input sets. Such duplicates are eliminated.

rpl

Signature

`!op (rpl` *input-set*`:[]` *production-section*`:[]):[]`

production-section is defined as for `red`.

Description

Replaces the elements in the input set that match the patterns with substitutes associated to the patterns.

Return value

A set containing the elements from the input set that do not match the pattern and the productions, function of the elements of the input set that match the pattern.

The resulting set may contain duplicates depending on the patterns and associated productions. Such duplicates are eliminated.

ptc

Signature

`!op (ptc` *expression*`:` *production-section*`:[]):`

production-section is defined as for `red`.

Description

Patches code in an expression, that is, searches elements in the expression that match the patterns and replaces them by the productions associated with the patterns.

Return value

An expression where the elements that match the patterns are replaced by the corresponding productions.

4.2 Object Classes

4.2.1 Structure

Object classes are structured as a concatenation of opcodes, members and classes.

Class structures – pre-processor directives - allow referencing members in objects directly, using the syntax *an-object.a-member*, instead of referring to variables declared in patterns matching the objects. For example:

```
(sim_run: (mk.sim (mk.pro :executed_code ::) :resilience ::) |[])
...
(... executed_code resilience ...)
```

can be written:

```
(sim_run: (mk.sim ::) |[])
...
(... sim_run.proc.code sim_run.res ...)
```

where `proc` is the name of a member of `mk.sim` and `code` a member of `mk.pro`.

When the symbol *an-object*`.`*a-member* evaluates to an object, then it can be post-fixed with another reference to members of said object in the form: *an-object*`.`*a-member*`.`*a-member*.

This alternate syntax can reduce the depth of the patterns involved – in the example above, the standard syntax needs to specify a pattern for the process, while in the alternate syntax the process is accessed directly as a member of the simulation run, and there is no need to specify a pattern matching a process in `sim_run`.

Member identifiers are integers, used as indexes in the internal structure of the code (encoded in r-code, see Annex 3). For example:

with:

```
!class (c1: m1: m2:); m1=1 m2=2
```

the structure of an object of class `c1` is:

(*opcode-of-c1* `m1 m2`).

Different classes can expose members with identical names when they bear the same semantics. Member identifiers are actually registered per class: in atoms like *an-object*.*a-member*, the correct member identifier is selected according to the class of *an-object*. For example:

assuming:

```
!class (c1 m1: m2:); m1=1 m2=2
!class (c2 m3: m1:); m3=1 m1=2
```

if `o1` is an instance of class `c1` and `o2` an instance of class `c2`, then `o1.m1` is interpreted as `o1.1` whereas `o2.m1` is interpreted as `o2.2`.

There exist a notation for defining members on sets: (*set -name*`[]` *member-list*). This does not declare a particular class: it declares instead a set for which members are defined. This set name can be referred to for instantiating class templates (see below). It is also possible to define members on sets using an implicit set definition. For example:

```
!class (c1 m1: m6:[m2: m3: m4:] m5:)
```

exposes a class c1 with three members: `m1`, a set known as `m6` and `m5`. This is equivalent to:

```
!class (a_set[] m2: m3: m4:)
!class (c1 m1: m6:a_set m5:)
```

the sole difference being that `a_set` can be reused by other class structures.

Notice also that:

```
!class (c1 m1: m2:[m1: m2: m3:] m3:)
```

is a valid expression – no confusion is possible.

Classes can embed other classes, for example:

assuming:

```
!class (c1 m1: m2:)
!class (c2 m3: m4:c1 m5:)
```

where `m4` is an object of class `c1`, then the structure of an object of class `c2` is (*opcode-of-c2* `m3 m4 m5`) and the structure of `m4` is (*opcode-of-c1* `m1 m2`).

Class structures allow variables in the member list. Such a variable must be valuated either by an existing class structure, by a member or by a member list. Class structures containing variables are actually class templates that need to be instantiated with actual arguments for the variables they contain. When instantiated, the variables of template classes will be replaced by

the actual arguments, that is, either a member, a member list or the member list of a class. For example:

assuming:

```
!class (c1 m1: m2:); m1=1 m2=2
!class (c2 m3: :c m4:); m3=1 m4=2
!class (c3 (c2 c1) m5:)
```

then `c3` embeds an instantiation of `c2` with `c1`. `(c2 c1)` is translated into:

`m3 (`*`opcode-of-c1`* `m1 m2) m4`

and the structure of an object of class `c3` is:

`(`*`opcode-of-c3`* `m3 (`*`opcode-of-c1`* `m1 m2) m4 m5); m3=1 m4=3 m5=4`

`c2` is a class template and there is no opcode for it - also, no objects can be instances of `c2`.

`!class (c4 (c2 {m6: m7:}))` has the following structure:

`(`*`opcode-of-c4`* `m3 m6 m7 m4).`

When valuated by `nil`, variables in class structures are skipped, for example:

assuming:

```
!class (c1 m3: :c m4:)
!class (c2 (c1 nil) m5:)
```

then the structure of an object of class `c2` is:

`(`*`opcode-of-c2`* `m3 m4 m5)`

Variables declared in class templates have the scope of the `!class` directive.

Class names starting with an underscore denote class templates.

When a member is declared as being of a type, valuating it with `nil` actually valuates the structure denoted by the type. For example passing `nil` where `nb` is expected actually passes the value `|nb`. In a similar way, `nil` can translate to `|[]`, `|ms`, `|did` and `|sid` depending on the context. This also applies for members of a class type: in this case, the member is valuated by a structure as defined by the class, where the members are valuated with `nil`. For example:

assuming:

```
!class (c1 m1:nb m2:[m3: m4:[]])
!class (c2 c1 m5:ms)
```

an object like: `(c2 nil nil)` is translated into:

```
c2
   c1 |nb []
      nil
      |[]
   |ms
```

Replicode predefines the class structures for built-in object/marker classes (in `std.replicode`) – see section 5 in this Annex.

4.2.2 Built-in Classes

For each class is provided:

- the class structure, i.e. its structure (if any) and an expression of the form: *opcode list-of-member-names* with a description of its members.;
- a development of the structure. Comments indicate member names. The symbol `...` used in the developments denotes place-holders defined in template classes, i.e. corresponds to a variable. N.B.: this symbol is not part of Replicode;

- a description of the class.

_obj

Structure

`!class (_obj :x res:~nb sln:~nb ijt:ms ejt:ms org:sid hst:sys :y mks:[])`

`res` – resilience: a number representing the remaining time to live of the object, in milliseconds.

`sln` – saliency: a number in [0,1] representing the final saliency value of the object.

`ijt` - injection-time: a timestamp representing the creation time of the object.

`ejt` - ejection-time: a timestamp representing the reception time of the object if it comes from another sub-system.

`org` – origin: the identifier of the sub-system (if any) that ejected the object into the host sub-system.

`hst` – host: a reference to the sub-system the object lives in.

`mks` - marker-set: the set of all markers referencing the object.

Development

`... res sln ijt ejt org hst ... mks`

Description

A template class defining common data for all system objects, i.e. objects that can constitute input data for reactive objects.

ptn

Structure

`!class (ptn skel: guards:[])`

`skel` – skeleton: an expression or an atom possibly containing variables.

`guards` - a set of guards: a guard is an expression referencing variables in the skeleton and expressing conditions on these.

Development

`ptn skel guards`

Description

Pattern. Patterns are not system objects.

|ptn

Structure

`!class (|ptn skel: guards:[])`

Same constituents as in a pattern.

Development

Same development as `ptn`.

Description

Anti-pattern. Anti-patterns are not system objects.

in_sec[]

Structure

`!class (in_sec[] inputs:[] timings:[] guards:[])`

`inputs` – anything.

`timings` – a set of guards expressing timing constraints on the inputs.

`guards` – a set of guards expressing constraints on the inputs and timings.

Development

```
[]
   anything; inputs
   [expressions]; timings
   [expressions]; guards
```

Description

A construction used for defining inputs in reactive objects and functions. Input sections are not system objects. Notice that Replicode classes do not enforce typing. Here, any objects of any class can be used as inputs (e.g. sets of patterns or forward/inverse model inputs, see below).

`_model_inputs`

Structure

```
!class (_model_inputs initial_state:in_sec :x)
```

`initial_state` – an `in_sec` specifying the initial state in a model; `initial_state.inputs` is a set of patterns.

Development

```
[]; initial_state
   [patterns]; inputs
   [expressions]; timings
   [expressions]; guards
...
```

Description

A template class used to define inputs in models. Model inputs are not system objects.

`fmd_inputs[]`

Structure

```
!class (fmd_inputs[] (_model_inputs actions:in_sec))
```

`actions` – an input section specifying the actions in a forward model; `actions.inputs` is a set of patterns.

Development

```
[]
   []; initial_state
      [patterns]; inputs
      [expressions]; timings
      [expressions]; guards

   []; actions
      [patterns]; inputs
      [expressions]; timings
      [expressions] guards
```

Description

A construction used to define inputs in forward models. Forward model inputs are not system

objects.

imd_inputs[]

Structure

```
!class (imd_inputs[] (_model_inputs target_state:in_sec))
```

`target_state` – an input section specifying the target state in a forward model; `actions.inputs` is a set of patterns.

Development

```
[]
   []; initial_state
      [patterns]; inputs
      [expressions]; timings
      [expressions]; guards

   []; target_state
      [patterns]; inputs
      [expressions]; timings
      [expressions]; guards
```

Description

A construction used to define inputs in inverse models. Forward model inputs are not system objects.

prod_sub_sec[]

Structure

```
!class (prod_sub_sec[] guards:[] prods:[])
```

`guards` – a set of guards expressing additional constraints on the inputs and timings in reactive objects and functions.

`prods` - a set of calls to functions or device functions.

Development

```
[]
   [guards]; guards
   [productions]; prods
```

Description

A construction to define production sub-sections in reactive objects and functions. Production sub-sections are not system objects.

head[]

Structure

```
!class (head[] tpl:[::ptn] inputs:in_sec prods:[])
```

`tpl` – template arguments: a set of patterns.

`inputs` – `in_sec`: `inputs.inputs` is a set of patterns.

`prods` – set of `prod_sub_sec` defining the productions of a function or a reactive object.

Development

```
[]
```

```
   [patterns]; tpl
   []; inputs
      [patterns]; inputs
      [expressions]; timings
      [expressions]; guards
   [production-sub-sections]; prods
```

Description

A construction to define functions and reactive objects.

fun

Structure

```
!class (fun (_obj head:head nil))
```

Development

```
fun
[]; head
   [patterns]; tpl
   []; inputs
      [patterns]; inputs
      [expressions]; timings, always empty
      [expressions]; guards
   [production-sub-sections]; prods
res sln ijt ejt org hst mks
```

Description

Function. A function is instantiated using the instantiation operator (`ins`). A function is called using the application operator (`app`). N.B.: the timing set is always empty (`|[]`) for functions.

_react_obj

Structure

```
!class (_react_obj (_obj {head:head act:~nb tsc:~ms csm:nb sig:nb nfc:nb
nfr:nb} :x))
```

`act` – activation value: a number in [0,1] representing the activation value of programs, models and sub-systems.

`tsc` – time scope: a number (in milliseconds) representing the time to live of an offspring.

`csm` – consumption flag: a number in {0,1} representing whether or not a program or model consumes its input objects, i.e. whether input objects when matched successfully are also inputs for offsprings or not. Predefined macros: CONSUME and PASS.

`sig` – signaling flag: a number in {0,1} representing whether or not a program or model is to be periodically signaled by its host sub-system. Predefined macros: SIGNAL_ON and SIGNAL_OFF.

`nfc` – notify flag: a number in {0,1} representing whether or not the rCores executing a program or model notifies the function calls to the executive. Predefined macros: `NOTIFY` and `SILENT`.

`nfr` – notify flag: a number in {0,1} representing whether or not the rCores executing a program or model notifies the reduction of input objects. Predefined macros: `NOTIFY` and `SILENT`.

Development

```
[]; head
   [patterns]; tpl
   []; inputs
      [patterns]; inputs
```

```
      [expressions]; timings
      [expressions]; guards
   [production-sub-sections]; prods
act tsc csm sig nfc nfr res sln ijt ejt org hst ... mks
```

Description

A template class used to define common data for reactive objects.

pgm

Structure

```
!class (pgm (_react_obj nil))
```

`inputs.inputs` is a set of patterns.

Development

```
pgm
[]; head
   [patterns]; tpl
   []; inputs
      [patterns]; inputs
      [expressions]; timings
      [expressions]; guards
   [production-sub-sections]; prods
act tsc csm sig nfc nfr res sln ijt ejt org hst mks
```

Description

Program. Productions are output when objects match the patterns and guards defined in the input section. To be executable, a program needs to be instantiated, i.e. passed actual template parameters (using `ins`).

|pgm

Structure

```
!class (|pgm (_react_obj nil))
```

Same constituents as in a program.

Development

Same development as for a program.

Description

Anti-program, i.e. a program that outputs its productions when no objects match its input section. To be executable, an anti-program needs to be instantiated, i.e. passed actual template parameters (in an application object, see below).

fmd

Structure

```
!class (fmd (_react_obj cer:nb))
```

`cer` - certainty is a number in [0,1] representing the likelihood of the model to depict to reality.

Development

```
fmd
[]; head
   [patterns]; tpl
   []; inputs
```

```
      [];inputs
         [];initial_state
            [patterns];inputs
            [expressions];timings
            [expressions];guards

         [];actions
            [patterns];inputs
            [expressions];timings
            [expressions];guards
      [expressions];timings
      [expressions];guards
   [production-sub-sections];prods
act tsc csm sig nfc nfr res sln ijt ejt org hst cer mks
```

Description

Forward Model. Forward models are structures like reactive objects where the inputs of the input section are defined by an instance of `_fmd_inputs`. To be executable, a forward model needs to be instantiated, i.e. passed actual template parameters (using `ins`).

imd

Structure

```
!class (imd (_react_obj cer:nb))
```

`cer` – defined as for forward models.

Development

```
imd
[];head
   [patterns];tpl
   [];inputs
      [];inputs
         [];initial_state
            [patterns],inputs
            [expressions];timings
            [expressions];guards

         [];target_state
            [patterns];inputs
            [expressions];timings
            [expressions];guards
      [expressions];timings
      [expressions];guards
   [production-sub-sections];prods
act tsc csm sig nfc nfr res sln ijt ejt org hst cer mks
```

Description

Inverse Model. Inverse models are structures like reactive objects where the inputs of the input section are defined by an instance of `_imd_inputs`. To be executable, an inverse model needs to be instantiated, i.e. passed actual template parameters (using `ins`).

sys

Structure

```
!class (sys (_obj head:[ctr:head upr:nb spr:nb sinks:[] drains:[] auto_eject:nb
sln_thr:nb act_thr:nb sln_oor:nb sln_boost:nb sln_thr_boost:nb sln_rlx_time:ms
act_oor:nb act_boost:nb act_thr_boost:nb act_rlx_time:ms sln_dec:nb
sln_dec_prd:nb act_dec:nb act_dec_prd:nb sln_del_thr:nb act_del_thr:nb]
act:~nb))
```

`ctr` - constructor: a `head` executed in the sub-system just after the instantiation of the sub-system.

`upr` - update-period: a number representing the period with which the sub-system (actually, the rMem managing the sub-system) updates the saliency and activation values of the objects in the sub-system. The period is to be chosen among a list of available periods defined by the executive (see Annex 2).

`spr` – signaling period: a number representing the period with which the sub-system triggers the execution of programs and models. The period is to be chosen among a list of available periods defined by the executive (see Annex 2).

`sinks` – the set of interfaces for which the sub-system is a source.

`drains` – the set of interfaces for which the sub-system is a source.

`auto-eject` - a number in {0, 1} turning on or off the auto-eject functionality. Predefined macros: `AUTO_EJECT_ON` and `AUTO_EJECT_OFF`.

`sln_thr` – saliency threshold: a number in {0,1} specifying the saliency threshold of the sub-system.

`act_thr` – activation threshold: a number in {0,1} specifying the activation threshold of the sub-system.

`sln_oor` – saliency out of range: a number in [0,1] specifying a distance from the average of the saliencies held by the projections in the sub-system. When an object in the sub-system gets a saliency greater than or equal to the average plus `sln_oor`, the sub-system's saliency is increased by a value specified by `sln_boost`, and the sub-system's threshold is increased by the value `sln_thr_boost`.

`sln_boost` – saliency boost: a number in [0, 1]. Cf. `sln_oor`.

`sln_thr_boost` – saliency threshold boost: a number in [0, 1]. Cf. `sln_oor`.

`sln_rlx_time` – saliency relaxation time: a number representing the time (in milliseconds) of relaxation for the saliency value and saliency threshold of the sub-system after a boost.

`act_oor` – activation out of range: like `sln_oor`, for activation values.

`act_boost` – activation boost: like `sln_boost`, for activation values.

`act_thr_boost` – activation threshold boost: like `sln_thr_boost`, for activation values.

`act_rlx_time` – activation relaxation time: like `sln_rlx_time`, for activation value and threshold.

`rlx_prd` – relaxation period: an integer specifying the intervals when the adjustments of the control values (saliency, activation and respective thresholds) are to be performed. This number is to be chosen in accordance with the executive period index (see Annex 2),

`sln_dec` – saliency decay: a number in [0,1] representing the value by which the saliency values of the objects in the sub-system.

`sln_dec_prd` – saliency decay period: a number representing the period at which the saliency and activation values of objects in the sub-system are decreased. This number is to be chosen in accordance with the executive period index (see Annex 2),

`act_dec` – activation decay: like `sln_dec` for activation values.

`act_dec_prd` – activation decay period: like `sln_dec_prd` for activation values.

`sln_del_thr` – saliency deletion threshold: a number in [0,1] specifying a threshold. Any object in the sub-system holding a saliency value under this threshold will be destroyed.

`act_del_thr` – activation deletion threshold: like `sln_del_thr` for activation values.

`act` – activation: defined as for `_react_obj`.

Development

```
sys
[]; head
   []; ctr
      [patterns]; tpl
      []; inputs
         [patterns]; inputs
         [expressions]; timings
         [expressions]; guards
      [production-sub-sections]; prods
   upr spr
   [interfaces]; sinks
   [interfaces]; drains
   auto_eject sln_thr act_thr sln_oor sln_boost sln_thr_boost sln_rlx_time
   act_oor act_boost act_thr_boost act_rlx_time sln_dec sln_dec_prd act_dec
   act_dec_prd sln_del_thr act_del_thr
res sln ijt ejt org hst act mks
```

Description

Sub-system. The various periods are to be chosen among a list of available periods defined by the executive (see Annex 2)..

int

Structure

```
!class (int dest:sid src:sid sln:nb act:nb)
```

`dest` - destination: the identifier of the destination sub-system. The interface will appear in the destination's `drains`.

`src` - source: the identifier of the source sub-system. The interface will appear in the source's `sinks`.

`sln` - saliency: a number in [0,1] representing the saliency of the interface in the destination sub-system. This value is not mediated.

`act` - activation: a number in [0,1] representing the activation of the interface in the source sub-system. This value is not mediated.

Development

```
int dest src sln act
```

Description

Interface between two sub-systems, the first one being the source and the second, the destination. None of the references can be `nil`.

Interfaces are not system objects.

gol

Structure

```
!class (gol tpl (_obj target_state:head nil))
```

`target_state` - defined as the input section of a program.

Development

```
gol
[]; target_state
```

```
   [patterns]; tpl
   []; inputs
      [patterns]; inputs
      [expressions]; timings
      [expressions]; guards
res sln ijt ejt org hst mks
```

Description

Goal. Goals must be instantiated, i.e. passed actual template arguments (using `ins`).

|gol

Structure

```
!class (|gol tpl (_obj target_state:head nil))
```

Same constituents as in a goal.

Development

Same development as for a goal.

Description

Anti-goal, i.e. the goal not to reach the target state. Anti-goals must be instantiated, i.e. passed actual template arguments.

cmd

Structure

```
!class (cmd (_obj head:[function:fid device:did args:[]] nil))
```

`function` - device function: the function to be executed by the device.

`device` - the device commanded to execute the function.

`args` - arguments: a list of arguments in a set.

Development

```
cmd
[];head
   function
   device
   args
res sln ijt ejt org hst mks
```

Description

A call to a device function, i.e. a command, sent in asynchronous mode - there is no return value. The result of the command, if any, shall be produced by the device, using a marker `xet` referencing the process triggered by the creation of the command object and holding the value produced by the process. In case of failure, the device shall produce a specific (i.e. application dependent) marker referencing the process indicating the reasons of the failure.

When a command is defined as a production in a reactive object, a macro is defined (CALL, see section 5 in this Annex) to assign default values to the members `res`, `sln`, `ijt`, `ejt org`, `hst` and `mks`.

ent

Structure

```
!class (ent (_obj nil nil))
```

Development

`ent res sln ijt ejt org hst mks`

Description

Entity.

smr

Structure

`!class (smr (_obj nil nil))`

Development

`smr res sln ijt ejt org hst mks`

Description

Simulation run. Simulation runs are referred to by hypothesis and simulation markers: they allow relating these markers to a common simulation object to differentiate them from other simulations.

dev

Structure

`!class (dev sln:~nb act:~nb mks:[])`

`sln` – saliency*:* defined as in `_obj`.

`act` – activation*:* defined as for `_react_obj`.

`mks` – marker set*:* defined as in `_obj`.

Development

`dev sln act mks`

Description

Device.

4.3 Marker Classes

References or sets of references held by markers shall always be declared last before the marker set (`mks`) – see section 1.2 in Annex 3. This rules shall govern the writing of `!class` directives exposing the structure of marker classes, but also the writing of the code intended to create dynamically new marker classes.

`mk.pro`

Structure

`!class (mk.pro (_obj {code: args:[]} nil))`

`code` - the executed code: a reference to a reactive object or to a command (`cmd`)

`args` – the inputs objects that triggered the execution of the code: a set of references to the input objects.

Development

`mk.pro code args res sln ijt ejt org hst mks`

Description

Process instantiation.

`mk.xet`

Structure

```
!class (mk.xet (_obj {proc: prod:[] dur:ms} nil))
```

`proc` - process: a reference to a process (`mk.pro`) or to a command (`cmd`).

`prod` - productions: the set of objects injected as the result of the process, or of the execution of a command by a device.

`dur` - duration: a number in milliseconds representing the duration of the process.

Development

```
mk.xet proc prod dur res sln ijt ejt org hst mks
```

Description

Process execution time. The reference to the process cannot be `nil`.

`mk.hyp`

Structure

```
!class (mk.hyp (_obj {data: sim_run:smr} nil))
```

`data` – the object marked as a hypothesis.

`sim_run` – simulation run: a reference to a simulation run (smr).

Development

```
mk.hyp data sim_run res sln ijt ejt org hst mks
```

Description

Marks an object as being a hypothesis, i.e. a premise or the result of a simulation. None of the references can be `nil`.

`mk.sim`

Structure

```
!class (mk.sim (_obj proc:mk.pro nil))
```

`proc` – process: a process executing code in simulation mode.

Development

```
mk.sim proc res sln ijt ejt org hst mks
```

Description

Simulation marker, i.e. a tag attached to a process representing a program or model being executed in simulation mode, or to a process representing the execution of one invocation of either `bwd` or `fwd`. The marker refers to an existing simulation run (`smr`).

`mk.pre`

Structure

```
!class (mk.pre (_obj {data: cer:nb} nil))
```

`data` – the predicted object.

`cer` – certainty: a number in [0,1] expressing the likelihood of the prediction to turn to be true.

Development

```
mk.pre data cer res sln ijt ejt org hst mks
```

Description

Prediction. The predicted time of the occurrence of `data` is the injection time in data. When injecting a prediction on `data`, it is allowed to set some of the latter members to `nil` (ex: activation, saliency, etc.). The reference to `data` cannot be `nil`.

mk.isa

Structure

`!class (mk.isa (_obj {inst: class: cer:nb} nil))`

`inst` – instance: the object described as being an instance of `class`.

class - the object representing a class of objects in an application domain.

`cer` - certainty is a number in [0,1] representing the likelihood of the marker to depict to reality.

Development

`mk.isa inst class cer res sln ijt ejt org hst mks`

Description

Instance relationship: the first object being an instance of the second. None of the references can be `nil`.

4.4 Entities

self

Description

The application, i.e. the system as an actor in the application domain. The executive injects `self` is any newly created sub-system.

4.5 Sub-systems

main

Description

The main system, i.e. a technical constituent of the `self`.

4.6 Devices

exe

Description

The executive.

5 Standard Definitions

This section gives the standard definitions used in Replicode, specified in the file `std.replicode`.

```
;std.replicode

;mapping objects -> r-code
!class sys
!class (_obj :x res:~ms sln:~nb ijt:ms ejt:ms org:sid hst:sys :y
mks:[])
!class (head[] tpl:[::ptn] inputs:in_sec prods)
!class (fun (_obj head:head nil))
!class (_react_obj (_obj {head:head act:~nb tsc:ms csm:nb sig:nb
nfc:nb nfr:nb} :x))
!class (pgm (_react_obj nil))
!class (|pgm (_react_obj nil))
!class (fmd (_react_obj cer:nb))
```

```
!class (imd (_react_obj cer:nb))
!class (in_sec[] inputs:[] timings:[] guards:[])
!class (_model_inputs initial_state:in_sec :x)
!class (fmd_inputs[] (_model_inputs actions:in_sec))
!class (imd_inputs[] (_model_inputs target_state:in_sec))
!class (ptn skel: guards:[])
!class (|ptn skel: guards:[])
!class (prod_sub_sec[] guards:[] prods:[])
!class (gol tpl:[] (_obj target_state:head nil))
!class (|gol tpl:[] (_obj target_state:head nil))
!class (sys (_obj head:[ctr:head upr:nb spr:nb sinks:[] drains:[]
auto_eject:nb sln_thr:nb act_thr:nb sln_oor:nb sln_boost:nb
sln_thr_boost:nb sln_rlx_time:ms act_oor:nb act_boost:nb
act_thr_boost:nb act_rlx_time:nb sln_dec:nb sln_dec_prd:nb
act_dec:nb act_dec_prd:nb sln_del_thr:nb act_del_thr:nb] act:~nb))
!class (int dest:sid src:sid sln:nb act:nb)
!class (cmd (_obj head:[function:fid device:did args:[]] nil))
!class (ent (_obj nil nil))
!class (smr (_obj nil nil))
!class (dev sln:~nb act:~nb mks:[])

;mapping operator opcodes -> r-atoms
!op (_now):ms
!op (= : :):
!op (<> : :):
!op (> : :):
!op (< : :):
!op (>= : :):
!op (<= : :):
!op (+ : :):
!op (- : :):
!op (* :nb :nb):nb
!op (/ :nb :nb):nb
!op (dis :nb :nb):nb
!op (ln :nb):nb
!op (exp :nb):nb
!op (log :nb):nb
!op (e10 :nb):nb
!op (\ :):
!op (ins : :[]):
!op (app: :[]):
!op (red :[] :[]):[]
!op (|red :[] :[]):[]
!op (com :[] :[]):[]
!op (spl :[] :[]):[]
!op (mrg :[] :[]):[]
!op (rpl :[] :[]):[]
!op (ptc : :[]):

!operator aliases
!def now (_now)
!def = equ
!def <> neq
!def > gtr
```

```
!def < lsr
!def <= gte
!def => lse
!def + add
!def - sub
!def * mul
!def \ syn

;mapping devices -> r-atoms
!def exe 0x8B000000; the executive.

;mapping sub-systems -> r-atoms
!def main 0x8A000000; the main sub-system.

;mapping device functions -> r-atoms
!def (_inj :)
!dfn (_eje :)
!dfn (_rep :)
!dfn (_mod : :nb :nb)
!dfn (_set : :nb :nb)
!dfn (_new_class :)
!dfn (_new_int :int)
!dfn (_del_class :)
!dfn (_del_int :int)
!dfn (_bwd :[] :nb :ms)
!dfn (_fwd :nb :ms)
!dfn (_ldc :st)
!dfn (_swp :sys :nb)
!dfn (_ntf :nb :nb)
!dfn (_new_sys :sid :nb)
!dfn (_del_sys :sid)
!dfn (_new_obj :sid :)
!dfn (_new_dev :st :nb)
!dfn (_del_dev :sid)
!dfn (_new_spc)
!dfn (_del_spc :nb)
!dfn (_new_prj :nb :nb :nb)
!dfn (_del_prj :nb :nb)
!dfn (_mod_act :nb :nb :nb)
!dfn (_mod_act_thr :nb :nb)
!dfn (_act_ : :nb)
!dfn (_sln_ : :nb)
!dfn (_del_ :)
!dfn (_start :nb :ms)
!dfn (_suspend)
!dfn (_resume)
!dfn (_stop)

;device functions aliases
!def (CALL function device args) (cmd function device args 100 1
|ms |ms |sid |sid |[])

!def (inj args) (CALL _inj exe args)
```

```
!def (eje args) (CALL cmd _eje exe args)
!def (rep args) (CALL cmd _rep exe args)
!def (mod args) (CALL cmd _mod exe args)
!def (set args) (CALL cmd _set exe args)
!def (new_class args) (CALL cmd _new_class exe args)
!def (new_int args) (CALL cmd _new_int exe args)
!def (del_class args) (CALL cmd _new_class exe args)
!def (del_int args) (CALL cmd _del_int exe args)
!def (bwd args) (CALL cmd _bwd exe args)
!def (fwd args) (CALL cmd _fwd exe args)
!def (ldc args) (CALL cmd _ldc exe args)
!def (swp args) (CALL cmd _swp exe args)
!def (ntf args) (CALL cmd _ntf exe args)
!def (new_sys args) (CALL cmd _new_sys exe args)
!def (del_sys args) (CALL cmd _del_sys exe args)
!def (new_obj args) (CALL cmd _new_obj exe args)
!def (new_dev args) (CALL cmd _new_dev exe args)
!def (del_dev args) (CALL cmd _del_dev exe args)
!def (new_spc args) (CALL cmd _new_spc exe args)
!def (del_spc args) (CALL cmd _del_spc exe args)
!def (new_prj args) (CALL cmd _new_prj exe args)
!def (del_prj args) (CALL cmd _del_prj exe args)
!def (mod_act args) (CALL cmd _mod_act exe args)
!def (mod_act_thr args) (CALL cmd _mod_act_thr exe args)

;various constants
!counter __constant 0

!def OFF __constant
!def ON __constant

;parameters for tuning the behavior of reactive objects
;member sig
!def SIGNAL_OFF __constant; disables signaling by the sub-system
!def SIGNAL_ON __constant; enables signaling by the sub-system
!def SILENT __constant; no notification upon production
!def NOTIFY __constant; notification upon productions
;member csm
!def CONSUME __constant; inputs objects are consumed
!def PASS __constant; input objects are passed to offsprings

;parameter for sub-systems
!def AUTO_EJECT_OFF __constant
!def AUTO_EJECT_ON __constant

;constants for _ntf
!def INJ __constant
!def EJE __constant
!def MOD __constant
!def SET __constant
!def BWD __constant
!def FWD __constant
!def NEW_CLASS __constant
```

```
!def DEL_CLASS __constant
!def NEW_INT __constant
!def DEL_INT __constant
!def NEW_DEV __constant
!def DEL_DEV __constant
!def NEW_SPC __constant
!def DEL_SPC __constant
!def NEW_PRJ __constant
!def DEL_PRJ __constant
!def MOD_ACT __constant
!def MOD_ACT_THR __constant
!def _ACT __constant
!def _SLN __constant
!def _DEL __constant
!def STOP __constant

;mapping markers -> r-code
!class (mk.pro (_obj {code: args:[]} nil))
!class (mk.xet (_obj {proc:mk.pro prod:[] dur:nb} nil))
!class (mk.hyp (_obj {data sim_run:smr} nil))
!class (mk.sim (_obj proc:mk.pro nil))
!class (mk.pre (_obj {data: cer:nb} nil))
!class (mk.isa (_obj {inst: class: cer:nb} nil))

;user-defined classes
!load ./usr.classes.replicode

;user-defined operator opcodes
!load ./usr.operators.replicode

;user-defined device opcodes
!load ./usr.devices.replicode

;user-defined device functions opcodes
!load ./usr.device.functions.replicode
```

Notice the last five directives for loading user-defined object classes, operators, device, device functions and class structures. All these are to be defined in files having the predefined paths, given by their corresponding loading directives.

Annex 2 – Specification of the Executive

1 Overview

The executive is composed of several mBrane modules distributed over the cluster:

- the rMems are managing sub-systems (see section 7.2 of the main document);
- the rCores are responsible for the execution of instantiated programs and models (see section 7.1 of the main document);
- each node contains a module – called loader – responsible for loading code in the node from images, i.e. archives of r-code (see section 2 of the Annex 3);
- the frontal node contains also a loader module – called main loader - that distributes the code amongst the loaders in the nodes;
- the frontal node contains optionally a debugging node that receives debugging messages issues by reactive objects and sends debugging commands to other reactive objects.

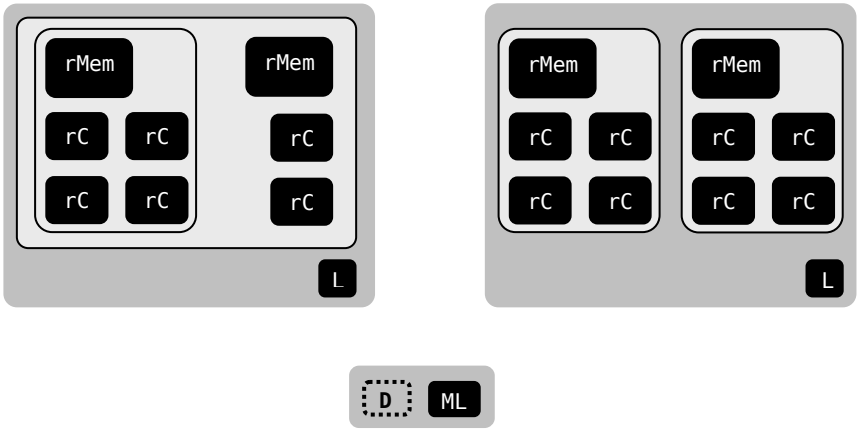

Figure 1 – The Executive

A system with two nodes (top) and a frontal (bottom) is represented. The two nodes contain a code loader (L) and the frontal host a main loader (ML) and an optional debugging module (D). Each of the computing nodes hosts a number of sub-systems (black-outlined boxes). Each sub-system is managed by one rMem, and hosts several rCores (rC). One rCore executes one reactive object. The top left node hosts a sub-system that embeds another one.

All the modules that constitute the executive communicate using mBrane as a message bus; they are all projected on the same mBrane space (not represented).

The components forming the executive are defined in one shared object, `r_exec.so`.

Besides the various modules, the executive defines two values to be used for defining the time periods that parameterize cyclic behaviors (e.g. `upr`, `spr`, the various periods used by the sub-systems and the updating period of mediated timestamps). These values are (in milliseconds):

- a base period: periodic values are expressed as multiple of this base value;
- the update period of mediated timestamps.

These values are initialized by the function `_start`.

2 rCore

2.1 Role

The role of an rCore is:

- to execute reactive objects – this activity is thereafter referred to as *reduction*;
- to signal reactive objects;
- to notify its sub-system when appropriate.

2.1.1 Signaling

When the member `sig` of a reactive object is set and when the object is active, the rCore that executes it periodically signals the object. This means that if the object is in position to execute the code defined in its production section – i.e., it has already matched all the input objects specified in its patterns -, the execution will be synchronized with an internal signal, i.e. delayed until the next signaling period has elapsed. N.B.: this behavior applies de facto to offsprings.

The signaling period (`spr`) is an instance member of rMems.

2.1.2 Reduction

Reduction consists in performing pattern matching on incoming objects and executing the code in the production section. Reduction operates on temporal sequences of objects, and these sequences may overlap. rCores create execution overlays to handle these overlaps, as defined in section 4.2 of the main document.

The output of the reduction is a set of views on command objects (`cmd`). These objects are passed to the rMem for execution. The latter will result on the actual injection of the products of the reduction in the sub-system.

An rCore is expected to define an array (`mBrane::sdk::_Array`, see Annex 3) called the value array, to store template arguments and intermediate results when evaluating expressions in code.

Evaluating expression requires executing operators: these are defined in a structure called the Operator Register (defined in Annex 4).

The rMem is responsible for spawning rCores and loading a reactive object when the latter gains activation; symmetrically, rMem terminates the rCore reactive object when the reactive object loses activation – see section 3 of this Annex for details.

2.2 Integration

An rCore is implemented as a set of components, some of them being mBrane modules, each performing pattern matching on one single pattern. These computations are shared by many rCores. Pattern matchers shall subscribe to one message class upon loading a pattern: a subscription specifies a message class and an integer value (an instance member of said class retrieved by the `getSubType` function of the class). The message class is `RView` (see Annex 3) and the integer value is the opcode of the r-atom leading the definition of the object encoded in the skeleton of the pattern.

Although mBrane supports the migration of modules across the cluster, this ability is not required for rCores: if migration was needed, the reactive objects would be relocated instead of the rCores.

2.3 Functions

Functions of the executive are exposed by rCores as instance member functions. An rCore is able to call the right function given its r-opcode[9]. Consult Annex 3 for the details of the r-code constructs (e.g. r-atoms, r-opcodes, etc.).

[9] e.g. using a switch on the r-opcode, or an array indexed by the r-opcode, containing pointers to the instance member functions.

The prototype of these functions is:

```
typedef void (*Operator)(ExecutionContext& context);
```

See Annex 4 for the definition of ExecutionContext. In the descriptions given below the type gives the arity of the functions.

Although rCores are meant to serve function calls to the executive, they actually implement only a few of said functions and delegate most of them to their respective rMems. The two sections below describe the functions of the executive (implemented and delegated): the arguments in the description of the functions given below are identified by the ordinal of an iterator in the function's signature.

Unless stated otherwise, when functions calls feature arguments that are actually r-atoms pointers (r-atoms of types internal, view, value, this or chain pointer), the indirections must be performed before invoking the C++ code. In particular, timestamps are to be passed using the leading r-atom, not the first 32 bits word.

The constants used to valuate arguments of some of the functions are defined in `std.replicode` (see section 5 of Annex 1).

As a general rule, all functions are required to fail gracefully when passed ill-formed or unsuitable arguments.

2.3.1 Implemented Functions

The following function is the only one actually implemented by the rCore.

_rep

Type

1

Arguments

`0:` the r-atom leading the definition of an instantiated reactive object.

Description

Replaces the code currently executed by another one. The code currently executed stops immediately and is replaced by the code provided as the argument. If said code is active enough, its execution starts immediately. The input objects already matched by the rCore before the call to `_rep` are discarded.

2.3.1 Delegated Functions

The following functions are delegated by the rCores to their respective rMems.

_inj

Type

1

Arguments

`0:` the r-atom leading the definition of an object.

Description

Injects a new object inside the sub-system hosting the rCore. The `ijt` member of the injected object holds a value defining a lower boundary on the injection time. For example, the value `(+ now 100)` means that the object shall be injected *after* 100 milliseconds, counted from the time of the invocation of `inj`. The `ejt` member shall be set to `|ms`. `ijt` shall be set in fine by the rMem to the actual time of injection.

_eje

Type

1

Arguments

`0:` the r-atom leading the definition of an object.

Description

Ejects a new object in all interfaces referencing the sub-system hosting the rCore as a sink. The member `ijt` shall be set to `|ms`, and `ejt` to a value defining a lower boundary on the ejection time. `ejt` shall be set in fine by the rMem to the actual time of ejection.

`_mod`

Type

`2`

Arguments

`0:` an r-atom representing a pointer chain to a numerical value.

`1:` a number specifying the amount to be added to the current value. For activation, saliency and thresholds, the number is in [-1, 1], for resilience values the number is in {0, 2^{24}}.

Description

Requests the rMem to modify of one of the members of an object. Such members must map to numerical values.

`_mod` does not do anything when the first argument is not a pointer chain.

When appropriate (i.e. when the value holds a mediation flag – see section 1.1 of the Annex 3), the rMem mediates these requests, i.e. it computes their average and uses the result as the final value of the target member. This mediation is performed at a fixed period (`upr`) defined for each sub-system. Members mapped to mediated timestamps (like `res`) are a special case: they are updated at a fixed period defined for the entire system.

When an object enjoys an infinite resilience, this cannot be affected by any invocations of `_mod` or `_set`.

`_set`

Type

`2`

Arguments

`0:` an r-atom representing a pointer chain to a numerical value.

`1:` a number specifying the new value. For activation and saliency values and respective threshold, it is a number in [0, 1]; for resilience values, a number in {-1, 2^{24}}, -1 specifying an infinite resilience; for periods, a number representing a multiple of the base period value.

Description

Requests the rMem to assign a value to one of the members of an object. As for `mod`, rMems mediate the final assigned value when appropriate.

`_set` does not do anything when the first argument is not a pointer chain.

For an object to gain infinite resilience, the average of set values must be -1. When an object enjoys an infinite resilience, this cannot be affected by any invocations of `_mod` or `_set`.

`_new_class`

Type

`1`

Arguments

`0:` an r-atom leading the definition of an object. This argument is used as a template to define a new r-atom (its type - object or marker – and arity).

Description

Creates a new object or marker class in the sub-system where the call is issued.

_new_int

Type
1

Arguments
0: an r-atom leading the definition of an interface.

Description
Creates a new interface. This interface is added to the sets of sinks and drains of the sub-systems involved in the interface.

_del_class

Type
1

Arguments
0: an r-atom leading the definition of the object to be deleted.

Description
Deletes an object or marker class from the sub-system where the call is issued. In addition, *all* instances of the class in the sub-system will be deleted, regardless of their resilience.

_del_int

Type
1

Arguments
0: an r-atom leading the definition of the interface to be deleted.

Description
Deletes an interface. This interface is removed from the sets of sinks and drains of the sub-systems involved in the interface.

_bwd

Type
3

Arguments
0: an r-atom representing a set of goals.
1: a numerical value representing the maximum depth of the search.
2: a timestamp value, representing a timeout.

Description
Performs goal-regression to find inverse models able to reach a goal given an initial state.

_fwd

Type
2

Arguments
0: a numerical value representing the maximum depth of the search.
1: a timestamp value, representing a timeout.

Description
Performs forward exploration to predict the outcome of some actions, given an initial state.

_ntf

Type

2

Arguments

0: a numerical constant indicating the notification type to change; the predefined constants correspond to functions of the executive:

```
INJ
EJE
MOD
SET
BWD
FWD
NEW_CLASS
DEL_CLASS
NEW_INT
DEL_INT
NEW_DEV
DEL_DEV
NEW_SPC
DEL_SPC
NEW_PRJ
DEL_PRJ
MOD_ACT
MOD_ACT_THR
_ACT
_SLN
_DEL
STOP
```

1: a number in {0,1} to enable or to disable (respectively) the notification.

Description

Enables or disables the notification of one function call to the executive. Regardless of the status of the notification, notifications in {INJECT, …, MOD_ACT_THR} are disabled if the member nfc of the issuing reactive object is set to 0. In other words, nfc is interpreted as a first level filter on notifications of function calls.

_ldc

Type

1

Arguments

0: a character string representing the path of the file containing the code to load (i.e. an image, see section 2 of the Annex 3).

Description

Loads code from a text input stream (Replicode form) or from a binary input stream (r-code form). _ldc actually creates an image, loads it from disk and maps it contents into memory.

_swp

Type

2

Arguments

0: an r-atom identifying a sub-system.

1: a numerical constant in {0, 1} indicating the operation to be performed (swap to disk: 0, swap

to memory. 1).

Description

Swaps a sub-system and its contents to/from disk. When swapped to disk. The r-code is stored as an image (see section 2 of the Annex 3).

_new_sys

Type

2

Arguments

0: a sub-system identifier.

1: a number identifying a node.

Description

Instantiates a sub-system in a computing node.

_del_sys

Type

1

Arguments

0: a sub-system identifier.

Description

Deletes a sub-system from the main system.

_new_obj

Type

2

Arguments

0: a sub-system identifier.

1: an r-atom leading the definition of an object.

Description

Instantiates an object in a sub-system. Equivalent to an (initial) injection.

_new_dev

Type

2

Arguments

0: a character string identifying the class of the device to create.

1: a number identifying a node.

Description

Instantiates a device in a computing node.

_del_dev

Type

1

Arguments

0: a number identifying a device.

Description

Deletes a device from the main system.

_new_spc

Type
`0`

Arguments
`None.`

Description
Creates a space in the main system.

_del_spc

Type
`1`

Arguments
`0:` a number identifying a space.

Description
Deletes a space from the main system.

_new_prj

Type
`3`

Arguments
`0:` a number identifying a space.
`1:` a number identifying a sub-system or a device.
`2:` an initial activation value in [0, 1].

Description
Projects a sub-system or a device onto a space in the main system.

_del_prj

Type
`2`

Arguments
`0:` a number identifying a space.
`1:` a number identifying a sub-system or a device.

Description
Un-projects a sub-system or a device from a space in the main system.

_mod_act

Type
`3`

Arguments
`0:` a number identifying a space.
`1:` a number identifying a sub-system or a device.
`2:` an activation value in [0, 1].

Description
Sets the activation value of a sub-system or of a device in a space in the main system.

_mod_act_thr

Type
2

Arguments
0: a number identifying a space.
2: an activation threshold value in [0, 1].

Description
Modifies the activation threshold of a space in the main system.

act

Type
2

Arguments
0: an r-atom leading the definition of an object.
1: a number in {0, 1} indicating the status (off or on) of the activation.

Description
Activates or deactivates an object. Used for notification only. Cannot be called by reactive objects.

sln

Type
2

Arguments
0: an r-atom leading the definition of an object.
1: a number in {0, 1} indicating the status (off or on) of the saliency.

Description
Makes an object salient or not. Used for notification only. Cannot be called by reactive objects.

del

Type
1

Arguments
0: an r-atom leading the definition of an object.

Description
Deletes an object from memory. Used for notification only. Cannot be called by reactive objects.

_start

Type
2

Arguments
0: the update period for mediated timestamps.
1: a number (in milliseconds) specifying the base of periodic values.

Description
Initializes the period value and base, then authorizes the execution of rCores and rMems.

_suspend

Type
0

Arguments
None.

Description
Suspends the execution of the application.

_resume

Type
0

Arguments
None.

Description
Resumes the execution of the application.

_stop

Type
0

Arguments
None.

Description
Terminates the application. The memory is flushed and all modules terminated.

2.4 Notifications

Each time an rCore calls a function of the executive, it also injects a notification of the call. This notification is enabled by setting the member nfc of the executed reactive object to 1.

Notifications from rCores are issued in the form of process objects (mk.pro) that reference command objects (cmd). These process objects will be tagged later on by execution markers by the rMem – since it is the rMem which actually executes the functions. There are two exceptions:

- there is no notification for _rep and _ntf;
- although the injection of the notification is technically a call to a function of the executive (_inj), this call is *not* notified.

The prototype of the notification of a call is:

(mk.pro (cmd *function* exe [*function-arguments*] ...) [*input-objects*] ...)

In addition to the notification of individual function calls, an rCore also notifies the reduction of the input objects, if the reactive object's member nfr is set to 1. The prototype of the notification of the reduction is:

(mk.pro *reactive-object* [*input-objects*] ...)

The rMem will tag this process object with an execution marker referencing the set of all the function calls issued by the reactive object.

Notifications from rCores are injected in their respective sub-systems.

By default, no notifications are enabled.

3 rMem

3.1 Role

The role of an rMem is:

- to mediate periodically the valuation of object members requested by calls to `_mod` and `_set`. The final value is the average of the requests, and mediated members are updated at periods defined by the `upr` member of the sub-system – except timestamp members, updated at a fixed period defined for the entire system.
- to detect reactive objects gaining activation, and spawning rCores to execute them; symmetrically, the rMem unloads a reactive object from an rCore when it loses activation;
- to detect object gaining saliency, and passing them to rCores as input objects;
- to update periodically the resilience of objects (decay) and detect their losing resilience to delete them from memory; if some other objects hold a reference to an object with no resilience, the object remains in memory but will never constitute an input for reactive objects anymore. It will however still be visible in patterns targeting objects that hold a reference to it;
- to detect identical objects and to discard the older ones - identity is the identity of r-code regardless of the mediated values;
- to notify reactive objects of key events occurring in the sub-system;
- to implement the behavior of sub-systems as defined in section 4.2.2/sys of the Annex 1.

3.2 Integration

rMems are tightly integrated with mBrane. rMems subscribe to all mBrane control messages to gain awareness of the changes involving modules, groups, spaces and projections. These changes are subjected to notification by rMems to their respective sub-systems.

3.3 Functions

rMems implement the functions delegated by the rCores. This implementation is not part of the current specification.

rMems do not expose any functions that can be called directly by reactive objects – that is, without the mediation of an rCore.

3.4 Notifications

Notifications by an rMem come as execution markers (`mk.xet`) referencing new or existing process objects:

- existing process objects: these are created by rCores when reactive objects call functions of the executive. The rMem tags these with an execution marker when the call completes. N.B.: the rMem executes the function call, not the rCores, except for `_rep`, which is not notified anyway. The prototype of such a notification is:

  `(mk.xet` *process productions duration* `...)`

  where *process* is the process produced as a notification by an rCore of a call to a function of the executive (see section 2.4 of this Annex), *productions* is a set containing the results of the function call[10] (if any), and *duration* is the duration of the execution of the function. In a similar way, the rMem tags the notifications of the reduction of input objects with a similar marker:

  `(mk.xet` *process productions duration* `...)`

  where *process* is the process object notifying the reduction of input objects, and *productions* is the set of all the function calls issued by the reactive object.
- new process objects: these are created by rMems in the following cases:

[10] For example, the result of `_inj` is the injected object.

- state transition: when reactive objects are getting activated or deactivated. The process object refers to a command (`cmd _act_ exe [`*target-object* `0/1] ...`). Similar notifications are triggered by saliency changes on any object in the sub-system (function `_sln` instead of `_act`).

- just before the destruction of an object, i.e. when its resilience drops down to 0 and the next timestamp update period has elapsed. The process object refers to a command (`cmd _del_ exe [`*target-object*`] ...`). Actual deletion occurs after a grace period equal to the update period for timestamps to give reactive objects a chance to react. If at this point in time the resilience remains at 0, the object is permanently destroyed. The actual deletion occurs one update period after the time when the rMem has computed a final value of 0 for the resilience.

Notifications from rMems are injected in their respective sub-systems.

By default, no notifications are enabled.

4 Loader Module

4.1 Role

The loader modules receive the images (see section 2 of the Annex 3) distributed by the main loader and map these images into the memory of their respective nodes; the loader then executes the initialization commands contained in the images.

Loader modules also have a symmetrical role, that is, to gather images generated by rMems as a result of `_swp`, and send them to the main loader.

4.2 Integration

The image file is made available to a loader module by the main loader, using NFS.

A loader module subscribes to the following message class (with a sub-type equal to the identifier of the main loader):

```
// file r_code.h

using namespace mBrane;
using namespace mBrane::sdk;
using namespace mBrane::sdk::payloads;

namespace r_code{
namespace messages{
class Image:
public DynamicData<Message<Image> >{
public:
   char path[256];
   uint16 getSubType();
   size_t dynamicSize() const;
};
}
}
```

`path` is a null-terminated string; it represents either the path of one image or the path of one directory containing several images.

The function `getSubType` returns the identifier of the sending module.

The function `dynamicSize` returns the length of `path`.

When produced locally on one node, image files are also transmitted using NFS to the frontal node. Notification to the main loader is performed by sending instances of `r_code::messages::Image`.

5 Main Loader

5.1 Role

The main loader sends the images produced by the translator to the loaders located on each node of the cluster.

The main loader has the symmetrical role of aggregating images sent by loader modules into one single image, suitable for storage.

5.2 Integration

The main loader distributes copies of the image to the computing nodes using NFS. The main loader notifies the loader modules by sending instances of `r_code::messages::Image`, and subscribes to the same class (with any sub-type).

6 Debugging Module

6.1 Role

Debugging a Replicode application makes use of hand-crafted reactive objects (called probes) whose purpose is to catch events and report to the developer. These probes are to be injected in the relevant sub-systems, either manually or as the result of the execution of code. The probes can be coded to display proactive behaviors, e.g. injecting corrective code dynamically on behalf of the developer, upon reception of specific control messages.

The debugging module is a module able to communicate with reactive objects in the application. The purpose of the debugging module is:

- to instantiate, load and delete probes in sub-systems;
- to receive debugging messages from the probes and send them to the developer;
- to receive debugging commands from the developer and send them to probes in the sub-systems.

6.2 Integration

The developer shall define custom object/marker classes to encode debugging information: no specific additional message class needs to be defined.

The debugging module is a module written in C++ that encodes/decodes r-code to interact with the developer. In order to interact with an application, the debugging module does not require any particular constructs or functions in addition to these already provided by the r-code specification. Consequently, the specification of the debugging module is not part of the current specification.

Annex 3 – r-code Specification

1 Overview

r-code is the name of the internal representation scheme of code specified in Replicode.

Objects are transported across the cluster as mBrane messages: basically, objects in r-code are instances of message classes – C++ classes deriving from mBrane base classes specified in the mBrane API (called the Core Library). There exists one message class per Replicode object class - either built in Replicode or user-defined (as extensions of the language).

To reduce overhead in memory and to reduce the overhead due to copy operations, the content of objects (for example, the code of a program) is shared amongst the sub-systems hosted by the same node. However, the values describing the object (`res`, `sln`, `hst`, `org`, etc.) must be defined locally per sub-system and cannot be shared. This calls for differentiating the content of the object from its descriptive data (exception: markers are encoded only in views). The latter are instances of specific classes, called views.

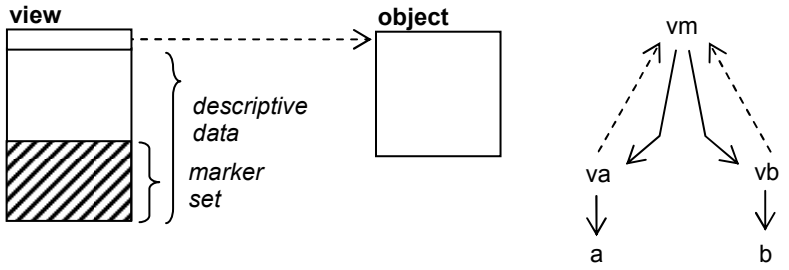

Figure 1 – Overview of Views and Objects

Left – *Views reference the content of their object (dashed arrow). Marker sets are part of the descriptive data. As they can change dynamically, they constitute the last instance member of the views. Descriptive data are stored in contiguous arrays.*

Right – *A marker (view* vm*) holding references to two views (*va *and* vb*), referencing respectively two objects (a and b). Arrows are smart pointers, dashed arrows are smart pointers contained in the respective marker sets held by va and vb. Smart pointers (class* `mBrane::sdk::P`*) count references on their targets and deallocate them when the reference count drops down to 0 – they also handle the circularity of the referencing scheme represented here.*

Some objects in Replicode hold references to others – markers, for instance. Both the transmission of pointed objects and the resolution of the pointers are handled by mBrane. However, markers pointing to views are themselves pointed by said views (in their marker sets): it is the duty of rMems to ensure consistency between markers and marker sets.

For transmission purposes, mBrane requires classes to provide two instance member functions, `ptrCount` and `ptr`, the former giving the number of pointers embedded in the instance and the latter iterating trough the pointers. For this reason, views and objects contain a separate array holding said pointers, and r-atoms corresponding to references on views (view pointers, see below) are actually indexes in the pointer array.

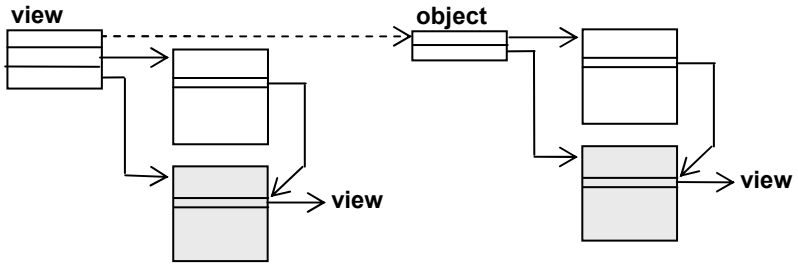

Figure 2 – Details of Views and Objects with Pointer Arrays

Detailed data structure for views and arrays, including pointer arrays. An r-view has 3 pointers: one to the object (dashed arrow), one to the descriptive data (white square) and one to a pointer array (in grey). Objects have a similar structure. Pointers in r-code are 16 bits indexes to the pointer arrays that contain the actual 64 bits addresses in memory of the pointed views. As in Figure 1, descriptive data and marker sets in views are stored contiguously, the latter after the former. Notice the absence of back-pointers form objects to views.

N.B.: all the pointers are smart pointers; the rMem in a sub-system also holds a smart pointer to each view (see Annex 2).

The distribution of data between views and objects is performed as follows:

- if a Replicode class defines a structure as its first member (a class or a set), said first member is stored in an instance of `object`, and the other members are stored in an instance of `view`. The reference to the object from the view is actually coded as a view pointer, even if the object is not technically a view;
- if a Replicode class does not defines its first member as a structure, then no object is needed and the reference to the object from the view is NULL. For example, markers do not generally require an object and, in that case, are encoded using only views.

1 r-code

Data in both the views and objects are Replicode constructs encoded in r-code. A Replicode atom is encoded by its counter part in r-code, called an r-atom. An r-atom is four bytes long, and the r-code expressions contained in views and objects are stored in a compact form, that is, in arrays of 64 bits words.

r-code is specified as a C++ shared object, (header `r_code.h`, binary `r_code.so`), containing the definitions of the constructions defined in the sections below.

1.1 r-atoms

An r-atom is composed of one byte - the descriptor - followed by three bytes - the data. The descriptor identifies the semantics of the data, i.e. how to interpret it.

There is one noticeable exception, r-atoms that represent floating point numbers: such r-atoms are numbers coded in the IEEE754-32 format on 32 bits, *right shifted by two bits*. The significand part of such a number is thus reduced to 21 bits (not counting the hidden bit). To differentiate r-atoms from floating point numbers, the convention is that r-atoms *not* representing floating point numbers have their most significant bit set to 1. The second leftmost bit of r-atoms representing floating point numbers indicate whether (1) or not (0) the value is a mediated value (as defined in section 7.2 of the main document). Mediated values are actually mediated when stored in views – in other words, when stored in objects, the values are not mediated.

r-code uses the following r-atoms (data are described as [*field-name* (size) | … | *field-name* (size)], sizes in bits):

Descriptor	*Value (hex)*	*Data*

float 32 bits	xx	the 3 least significant bytes of the IEEE754-32 representation, the most significant byte being the the descriptor itself. Encoding a number proceeds by right shifting it by two bits and decoding it, by left shifting the r-atom by two bits.
nil	80	[*unused* (24)]
Boolean	81	[*unused* (23) | value (1)]
wildcard (:)	82	[*unused* (24)]
tail wildcard (::)	83	[*unused* (24)]
internal pointer	84	[*unused* (8) | *index* (16)]
view pointer	85	[*unused* (8) | *index* (16)]
value pointer	86	[*unused* (8) | *index* (16)]
back pointer	87	[*unused* (8) | *index* (16)]
this	88	[*unused* (24)]
entity	89	[*unused* (24)]
sub-system	8A	[*node-identifier* (8) | *identifier* (16)]
device	8B	[*node-identifier* (8) | *class-identifier* (8) | *instance-identifier* (8)]
device function	8C	[*r-opcode* (16) | *unused* (8)]
self	8D	[*unused* (24)]
host	8E	[*unused* (24)]
chain pointer	C0	[*unused* (16) | *number-of-elements* (8)]
set	C1	[*unused* (8) | *number-of-elements* (16)]
object opcode	C2	[*r-opcode* (16) | *arity* (8)]
marker opcode	C3	[*r-opcode* (16) | *arity* (8)]
operator opcode	C4	| *unused* (8) | *r-opcode* (8) | *unused* (8)]
character string	C5	[*number-of-blocks-of-4-characters* (8) | *number-of-characters* (16)]
timestamp	C6	[*unused* (23) | *mediated* (1)]

r-atoms whose descriptors have the 6th bit set to 1 are atoms indicating structures composed of several atoms, like expressions lead by operators or by object/marker classes, sets, character strings, timestamps and also, user-defined object/marker classes (for example, n-dimensional vectors). Such atoms are thereafter called *structural atoms*.

The counter-parts of Replicode opcodes are called r-opcodes. Object/marker r-opcodes are used to specify the subscriptions of rCores (see `RView::sid` in section 1.3 in this Annex, see also Annex 2). Operator r-opcodes are indexes of operators in the Operator Register (see Annex 4).

`|nb` is coded `0x3FFFFFF`.

`|bl` is coded `0x81FFFFF`.

`|[]` is coded as a set with no elements, i.e. `0xC1000000`.

`|sid` is coded `0x8AFFFFF`.

`|did` is coded `0x8BFFFFF`.

`|fid` is coded `0x8CFFFFF`.

`|ms` is coded `0xC6FFFFF`.

Descriptor fields named *arity* contain the number of members for an object or marker class, or the number of operands for an operator. These numbers are limited to 255 for objects and markers and to 7 for operators.

The preprocessor shall use the `!class` directives to define r-atoms for object and marker classes, incrementing the r-opcode for each directive and deducing their respective arity from

the class structures. The type information associated with class members allow instantiating structures when these members are assigned undefined values. The return types are specified for operators for the sake of consistency with member declarations and are unused in this current specification. In particular, no type checking is required to be performed at runtime.

Internal pointers are indexes to retrieve r-atoms from the same array said internal pointers are located in. Indexes are absolute values (unsigned 16-bits integers).

View pointers are indexes to retrieve actual addresses in the view pointer array: these addresses are to be interpreted as *smart pointers* (class `mBrane::sdk::P`), except in the following case: the `hst` member defined by `_obj` is encoded as an index to `RView::host`, to be interpreted as a *lazy* pointer to the host sub-system, i.e. a regular address in memory (not counting references nor deallocating the sub-system).

Value pointers are like view pointers except that they do not target the view pointer array, but another array (the value array), allocated by the executive to store (a) actual template parameters and, (b) intermediate values resulting from the evaluation of the expressions in the code of reactive objects.

Back pointers are pointers located in the value array allocated for a reactive object that identify a location in the arrays of r-views that matched patterns of said reactive object.

Chain pointers are structures of r-atoms. After the leading r-atom, the first r-atom is a pointer to a view (this, view, or value pointer), followed by internal pointers holding indexes to the members of said object (see section 1.2 in this Annex). r-atoms called *this* are featured in chain pointers and indicate that the internal pointers that follow refer to the view.

r-atoms representing sub-systems and devices contain the identifier of their respective host computing nodes[11]. These identifiers are valuated dynamically by the executive. When defined in the source code, these identifiers shall be left undefined.

Timestamp r-atoms are followed by two 32 bits words containing the time (in microseconds[12]) since 01/01/1970. These words shall not be interpreted as r-atoms but as raw data. The first bit of the field called *mediated* in a timestamp r-atom indicates whether (1) or not (0) the timestamp is to be mediated by the rMem.

Replicode defines a pre-instantiated entity, `self`. The r-atom *self* is an atom behaving as a pointer: it indicates to retrieve the address of `self` in memory from the `RView` or from the `r-object` (class members `RView::Self` and `r_object::Self`). At loading time, the executive instantiates this entity as an `RView`, and valuates the address pointed to by the class members ::`Self` with the address of the instance of `self` in memory.

r-atoms are instances of the following class:

```
// file r_code.h

using namespace mBrane::sdk;

namespace r_code{
class RAtom{
public:
   uint64 atom;
   uint8 getDescriptor();
   bool  isPointer(); // returns true for all 5 pointer types.
   bool  isStructural();
   bool  isFloat();
   bool  readsAsNil(); // returns true for all undefined values
                       // and nil.
   typedef enum{
      NIL=0x80,
      BOOLEAN_=0x81,
```

[11] A cluster running a Replicode application can be composed of 256 nodes at most.
[12] a conversion between Replicode values and r-code values is necessary.

```
    WILDCARD=0x82,
    T_WILDCARD=0x83,
    I_PTR=0x84, // internal pointer
    V_PTR=0x85, // view pointer
    VL_PTR=0x86, // value pointer
    B_PTR=0x87, // back pointer
    THIS=0x88, // this pointer
    ENTITY=0x89,
    SUB_SYSTEM=0x8A,
    DEVICE=0x8B,
    DEVICE_FUNCTION=0x8C,
    SELF=0x8D,
    HOST=0x8E,
    C_PTR =0xC0, // chain pointer
    SET=0xC1,
    OBJECT=0xC2,
    MARKER=0xC3,
    OPERATOR=0xC4,
    STRING=0xC5,
    TIMESTAMP=0xC6
}Type;

// encoders
static RAtom Float(float32 f,bool mediated);
static RAtom UndefinedFloat();
static RAtom Nil();
static RAtom Boolean(bool value);
static RAtom UndefinedBoolean();
static RAtom Wildcard();
static RAtom TailWildcard();
static RAtom IPointer(int16 index);
static RAtom VPointer(int16 index);
static RAtom VLPointer(int16 index);
static RAtom BPointer(int16 index);
static RAtom This();
static RAtom Set(uint16 elementCount);
static RAtom CPointer(uint8 elementCount);
static RAtom Object(uint16 opcode,uint8 arity);
static RAtom Marker(uint16 opcode,uint8 arity);
static RAtom Operator(uint16 opcode);
static RAtom Entity();
static RAtom SubSystem(uint8 nodeID,uint16 sysID);
static RAtom UndefinedSubSystem();
static RAtom Device(uint8 nodeID,uint8 classID,uint8 devID);
static RAtom UndefinedDevice();
static RAtom DeviceFunction(uint16 opcode);
static RAtom UndefinedDeviceFunction();
static RAtom String(uint16 characterCount);
static RAtom UndefinedString();
static RAtom Timestamp(bool mediated);
static RAtom UndefinedTimestamp();
static RAtom Self();
static RAtom Host();
```

```
    // decoders
    float32 asFloat();
    int16  asIndex(); // for internal, view, and value pointers
    uint16 getAtomCount(); // arity of operators and
                           // objects/markers, number of atoms in
                           // pointers chains, number of blocks of
                           // characters in strings.
    uint16 asOpcode();
    uint8  getNodeID(); // devices and sub-systems
    uint8  getClassID(); // devices
    uint8  getDevID();
    uint16 getSysID();
    _Array<RAtom>::Iterator
    getAtom_struct(_Array<RAtom>::Iterator base,
                   int16                   index);
};
}
```

The function `getDescriptor` returns the descriptor of the atom.

The function `getAtom_struct` returns an iterator locating an r-atom member of a structure (lead by `this`). `base` is the location of the structural r-atom and `index` is the index in the structure of the atom to retrieve.

_Array is a super class of Array and thus a super class of `RView::data`, `RView::pointers`, `RObject::data` and `RObject::pointers` (see section 1.3 below in this Annex).

1.2 Encoding of Expressions

The encoding of Replicode expressions in r-code is performed according to the following rules:

I. an atom is encoded as its corresponding r-atom, except tail wildcards, atoms leading expressions, references to objects, references to members and timestamps;

II. a tail wildcard is encoded as an r-atom for the tail wildcard followed by r-atoms representing regular wildcards (`:`): this is meant to preserve the arity of the expression containing the tail wildcard. The number of inserted regular wildcards is *arity-of-the-expression* - *position-of-the-tail-wildcard-in-the-expression*;

III. an expression is encoded as the sequence of the structural r-atom corresponding to the opcode leading the expression, followed by the r-atoms corresponding to the components of the structure;

IV. when components of structures are structures themselves (i.e. sub-structures), they are encoded as internal pointers to a location (in the same array) containing the r-atoms encoding the sub-structure;

V. references to sub-systems are mapped to the *identifiers* of said sub-systems, not to view pointers: such references are encoded as single r-atoms representing sub-systems;

VI. references to variables or labels are encoded as value pointers;

VII. references to Replicode objects are encoded as view pointers;

VIII. references to members of Replicode objects are encoded as chain pointers, i.e. a structure of r-atoms. The first element is the r-atom for the chain, followed by a pointer to a view, and followed in turn by internal pointers to the members of the object. The pointer to the view is either a pointer to this, a view pointer, or a value pointer, depending on where the target object is located.

Indexing members in structures assumes the following rule: the index 0 is the location in

the array of the opcode leading the structure – the index of the first member is 1. The length of chained references to members is limited to 255.

Direct member access is also allowed on sets, i.e. the index of an element of a set is the offset or the r-atom of the element counted from the first one, 0 being the index of the r-atom indicating a set.

The allocation of r-atoms in arrays is to be performed *depth first*. The following algorithm (in pseudo code) demonstrates how to perform such allocations while encoding expressions from source code into r-code – for the sake of clarity, the algorithm is greatly simplified and does not implement all the rules given above:

```
input_stream: stream containing source code
writing index: the position where to write r-atoms in an array
in_a_structure: a flag indicating that the parsed code is an
element of either an expression or of a set.

encode(input_stream,index,writing_index,in_a_structure)
   atom=input_stream.getNextAtom()
   if atom is structural
      if in_a_structure
         write internal pointer at writing_index
         encode(input_stream,writing_index+atom.arity,false)
      else
         write atom at index
      for i=1 to atom.arity
         encode(input_stream,writing_index+1,true)
   else
      write atom at index
      encode(input_stream,writing_index+1,in_a_structure)
```

For example:

If input stream `s` contains the expression:

```
(opcode1 m1 (opcode2 m2 m3) m4 m5)
```

where `m4` is a set of 5 expressions and `m1`, `m2`, `m3` and `m5` are neither expressions nor sets, then `encode(s,0,false)` writes:

Index	*Content*
0	opcode1
1	m1
2	internal pointer to the sub-expression lead by opcode2 (5)
3	internal pointer to m4 (8)
4	m5
5	opcode2
6	m2
7	m3
8	r-atom for set (size: 5)
9	internal pointer to the first element of the set (14)
...	
13	internal pointer to the last element of the set
14	opcode leading the content of the first element of the set
15	content of the first element of the set
...	

See section 1.3 for more examples.

1.3 Views and Objects

This section gives the specification of the base classes for encoding views and objects. Each object class in Replicode corresponds to a pair of C++ classes (view and object), derived from the base classes. These base classes derive from mBrane message classes. They are defined as follows (also see examples in the next section):

```
// file r_code.h


using namespace mBrane;
using namespace mBrane::sdk;
using namespace mBrane::payloads;


namespace r_code{

class    RView;

class    dll_export     Code{
public:
         Code();
         payloads::Array<RAtom>        data;
         payloads::Array<P<RView> >    pointers;
         void    trace();
};

class    dll_export     RObject:
public   RPayload<RObject,StaticData,Memory>{
private:
         static  RView   *Self;
         Code            _code;
public:
         RObject();
         ~RObject();
         Code    *code();
         uint8           ptrCount();
         __Payload       *getPtr(uint16 i);
         void            setPtr(uint16  i,__Payload    *p);
};

class    dll_export     RView:
public   StreamData<RView,StaticData,Memory>{
private:
         static  RView   *Self;
         P<RObject>      object;
         Code            _code;
         RView           *host;  // won't be transmitted
public:
         RView(RObject  *object=NULL);
         ~RView();
         void    setSID(RAtom   a);
         RObject *getObject()   const;
         void    setObject(RObject     *o);
         Code    *code();
         uint64  &getResilience();
         uint8           ptrCount();
         __Payload       *getPtr(uint16 i);
         void            setPtr(uint16  i,__Payload    *p);
  };
}
```

Built-in markers are encoded as views that – generally - reference no object.

The `sid` function returns the r-opcode contained by the first r-atom in RView`::data`, except for views encoding `cmd`: these shall return the opcode of the device of the command.

The function `getMember` returns the address of the r-atom representing a member of the object or the view, given its index; it delegates to the object when the first member is a view pointer.

The function `ptrCount` returns the number of pointers held by `RView`, that is, the number of elements in `pointers` (i.e. `pointers.count()`) plus one (the object). For indexes grater than 0, `ptr` iterates the pointers in `RView::pointers`.

`Self` is defined as described in section 1.1 in this Annex.

`host` is the address (lazy pointer) of the view to the host sub-system. This pointer is not transmitted.

The function `getResilience` returns a reference to the resilience timestamp (using the res index contained in the r-atom leading the view). This is provided as a quick access for rMems as they have to periodically update resilience values to implement their decay.

1.4 Examples

This section describes the encoding of views and objects for programs (`pgm`) and process markers (`mk.pro`), assuming the declarations above.

pgm

An instance of `r_object` encoding the first member of a program typically contains the following r-atoms:

Index	*Content*
0	r-atom for head
1	internal pointer to tpl (4)
2	internal pointer to in_sec (s)
3	internal pointer to prods (t)
4	r-atom for set (size: n)
5	internal pointer to pattern
...	
4+n	internal pointer to pattern
5+n	r-atom for ptn (first pattern in tpl)
6+n	internal pointer to skel (8+n)
7 +n	internal pointer to guards (p)
8+n	leading opcode of skel
9+n	first member of skel
...	
p	r-atom for set (size: q)
p+1	internal pointer to guard (p+q+1)
...	
p+q	internal pointer to guard
p+q+1	leading opcode for guard
p+q+2	first member of guard
...	
r	r-atom for ptn (second pattern in tpl)
r+1	internal pointer to skel
r+2	internal pointer to guards
...	
s	r-atom for in_sec
r+1	internal pointer to patterns
r+2	internal pointer to timings
r+3	internal pointer to guards
...	
t	r-atom for set (size: u)
t+1	internal pointer to prod_sub_sec (t+u+1)
...	

```
t+u      internal pointer to prod_sub_sec
t+u+1    r-atom for prod_sub_sec
...
```

An instance of RView encoding the local data for a program typically contains the following r-atoms:

Index	*Content*
0	r-atom for pgm
1	view pointer to a pgm_object
2	r-atom for act
3	r-atom for tsc
4	r-atom for csm
5	r-atom for sig
6	r-atom for clk
7	r-atom for nfc
8	r-atom for res
9	r-atom for sln
10	internal pointer to ijt (15)
11	internal pointer to ejt (18)
12	r-atom for hst
13	r-atom for org
14	internal pointer to mks (21)
15	timestamp
16	time MSB
17	time LSB
18	timestamp
19	time MSB
20	time LSB
21	r-atom for set (size: m)
...	view pointer to a marker
21+m	view pointer to a marker

mk.pro

An instance of RView encoding `mk.pro` typically contains the following r-atoms:

Index	*Content*
0	r-atom for mk.pro
1	view pointer to a reactive object (code)
2	internal pointer to args (16)
3	r-atom for res
4	r-atom for sln
5	internal pointer to ijt (10)
6	internal pointer to ejt (13)
7	r-atom for hst
8	r-atom for org
9	internal pointer to mks (17+m)
10	timestamp
11	time MSB
12	time LSB
13	timestamp
14	time MSB
15	time LSB

```
16       r-atom for set (size: m)
...      view pointer to an input object
16+m     view pointer to an input object
17+m     r-atom for set (size: p)
...      view pointer to a marker
17+m+p   view pointer to a marker
```

2 Code Translator

The translator is composed of two C++ functions – `encode` and `decode` -, all defined in a single shared object named `r_trans.so`.

These functions take as input or output arguments, snapshots of memory (called images). These contain objects in r-code form, serialized from memory. Such images are mapped by the executive into memory to load the application. Images can also be written to and read from disk.

The internal structure of images is not part of the current specification, but their interface is. Besides the disk read/write operations, images must be able (1) to copy r-views and r-objects into images (i.e. archiving) and, (2) to map images into memory (i.e. loading code). r-code defines the prototype of images as follows:

```
// file r_code.h

namespace r_code{
class Image{
public:
   Image();
   ~Image();
   void read(std::ifstream &stream); // read from file
   void write(std::ifstream &stream); // write to file
   Image& operator << (RView *view);
   Image& operator << (RObject *object);
   Image& operator >> (RView *&view);
};
}
```

`Image& Image::operator << (RView *)` archives a view *and* the object it refers to, if any.

`Image& Image::operator << (RObject *)` archives an object; this operator is to be used when the object is not referenced by any view yet – this is often the case in the source code where objects are defined first, then views referencing them are created and injected in sub-systems.

Both << operators archive the views referenced by their respective view/object arguments.

`Image& Image::operator >> (RView *&)` loads a view in memory, the object it refers to, if any, and the views both the view and its object possibly refer to.

Images shall contain a table mapping member indexes to member names (the latter being specified by `!class` directives), to ease the work of the `decode` function (see below). For the same reason, images shall also contain a table mapping some r-atoms to literals when the original source code defined macros for these r-atoms.

The translator is meant to be used in interactive mode, i.e. `encode` in a scripting environment, and `decode` for debugging purposes. Its performance is expected to be adequate to these uses.

2.1 encode

The `encode` function translates code in the Replicode form into code in r-code form. The function has two overloads.

Signatures

```
void encode(std::ifstream &stream,
            Image         *&image,
            char          *&error_msg);

void encode(std::istringstream &stream,
            Image              *&image,
            char               *&error_msg);
```

Inputs

`stream`: an input character stream. File and string streams are valid inputs.

Outputs

`image`: an image in memory containing r-code objects.

`error_msg`: a character string containing the first syntax error encountered (`NULL` otherwise).

The stream is parsed in one single pass.

Timestamps are expressed in milliseconds in Replicode and need to be converted in microseconds.

Commands to the executive specified in the Replicode form (e.g. initialization commands) are also objects, and as such, must be included in the output buffers (so that the executive can actually execute them).

The function is supposed to output an error message at the first syntax error it encounters, and stop translating at that point in the stream. In that case, no image is produced.

2.2 decode

The `decode` function translates code in the r-code form into code in Replicode form.

Signature

```
void decode(Image              *&image,
            std::ostringstream stream);
```

Inputs

`image`: an image in memory containing r-code objects.

Outputs

`stream`: an output character stream.

Timestamps are expressed in microseconds in r-code and need to be converted in milliseconds.

In the code obtained form `decode`, r-atoms shall be replaced by their corresponding readable forms (if any), as defined in the source code using preprocessor directives:

- the member indexes shall be replaced by the corresponding member names, as defined by the `!class` directives;
- the object, marker and device functions opcodes shall be replaced by their corresponding definitions in `!class` directives;
- commands in productions shall be replaced using the aliases defined for the functions of the executive;
- literals representing constants in std.replicode shall replace their counterparts in r-code;
- the references to the sub-system *main* and the entity *self* shall be replaced by the symbols `main` and `self`.

In addition, decode shall also convert value pointers to references to variables and labels, and insert these variables and labels in the code. Names for these shall be composed of the underscore character, followed by a capital letter and at least one digit.

3 Module I/O

Modules of types I, II and V might need to communicate with the application. To do so, they would send or receive messages carrying Replicode objects, encoded in r-code.

Such modules have to convert r-code into C++ constructs (when receiving messages from the application) and vice versa (when notifying the application, i.e. sending messages).

Basically, modules shall use the view and object classes as the vectors of their I/O, that is, modules shall encode/decode information in/from arrays of r-atoms.

r-code constructs are declared in the `r_code.h` header file, the definitions being provided as a shared object `r_code.so`. The latter is to be loaded at runtime by the shared objects containing the definitions of modules of type I, II and V.

4 Dependencies

This section summarizes the various dependencies between the shared objects required to define an application.

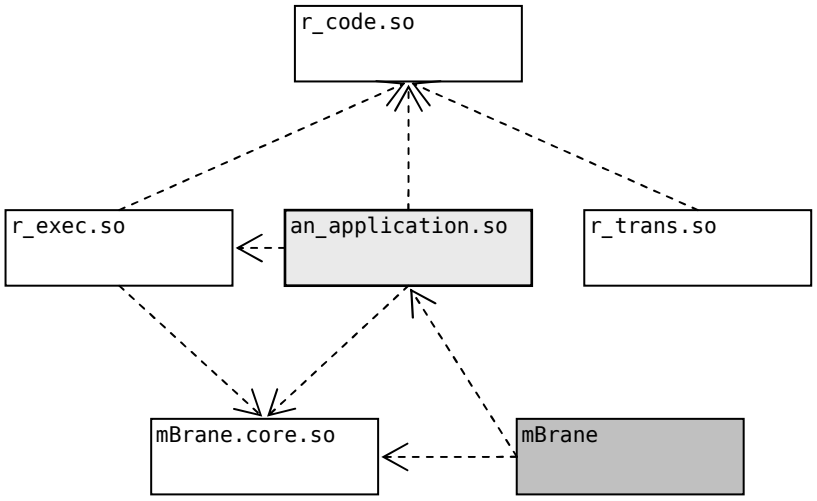

Figure 3 - Dependencies

Dependencies are indicated by the dashed lines: they correspond to the loading of pointed objects at runtime. The application is defined by the object an_application.so. mBrane is the only stand-alone executable.

The translator is not part of the application: it is loaded by the programming environment.

Annex 4 – Replicode Extension API

1 Purpose

The purpose of the Replicode Extension API is to extend Replicode with:

- new operators,
- new object classes,
- new marker classes,
- new devices,
- new device functions.

The operators and devices must be defined in a single shared object, called thereafter the application object.

2 Specification

2.1 Operators

Operators have to be registered in a table mapping the opcode of an operator to the address of the C++ function that implements it. The table is called the Operator Register and is defined as follows:

```
// file r_code.h

using namespace mBrane::sdk;

namespace r_code{

class ExecutionContext {
public:
        RAtom head();
        ExecutionContext child(uint16 index);
        int64 decodeTimestamp();
        void setResult(RAtom result);
        void setResultTimestamp(int64 timestamp);
        void beginResultSet();
        void appendResultSetElement(RAtom element);
        void endResultSet();
};

typedef void (*Operator)(ExecutionContext& context);

class OperatorRegister{
private:
   static OperatorRegister Singleton;
   void *operators[256];
   OperatorRegister();
   ~OperatorRegister();
public:
   static OperatorRegister &Get();
   Operator &operator [](uint16 r_opcode);
};
}
```

N.B.: `r_code.h` declares the functions implementing the built-in operators. The constructor of

`OperatorRegister` initializes the register with the addresses of said functions.

The r-opcode of custom operators shall be greater than or equal to 64, the first 64 being reserved for built-in operators. There can be defined up to 256 operators in total, i.e. up to 192 custom operators.

All operators are required to fail gracefully when passed unsuitable arguments.

The definitions of user-defined operator opcodes are to be written in the file `usr.operators.replicode`.

The OperatorRegister is defined in the shared object `r_core.so`.

2.2 Object and Marker Classes

Object and marker classes in Replicode are mapped to instances of `RView` and `RObject`: no particular C++ class is required to support user-defined object classes. The definition of the r-atoms that correspond to user-defined classes are to be found in the source code as preprocessor `!class` directives (file `usr.objects.replicode`).

2.3 Devices

Devices are mBrane modules and as such, are defined in the same shared object as above. For example:

```
// file some_application_modules.h

#include "an_application.h" // see section 2.5 in this Annex
#include "../core/module_node.h"

class    a_module;
MODULE_CLASS_BEGIN(a_module,Module<a_module>)
 void start(){
 }
 void stop(){
 }
 template<class T> Decision decide(T  *p){
    return WAIT;
 }
 template<class T> void react(T *p){ // to messages
 }
 template<class T> void react(uint16 sid,T *p){ // to stream data
 }
MODULE_CLASS_END(a_module)
```

The `react` and `decide` functions are to be specialized for specific incoming messages.

In r-code, devices are r-atoms whose structure is:

Descriptor	*Data*
`8F`	[*node-identifier*(1)|*class-identifier*(1)|*instance-identifier*(1)]

Devices shall be defined in the source code, using preprocessor `!def` directives (file `usr.devices.replicode`). The field called *node-identifier* is to be left to `0x0`, since it is to be valuated dynamically by the executive, depending on where an instance of the module will be instantiated in the cluster. In contrast, the fields *class-identifier* and *instance-identifier* must be assigned a correct value in the macro definition.

mBrane module instance identifiers are sequential, coded on 16 bits, starting at 0; mBrane module class identifiers are sequential, coded on 16 bits and start at 0. In the current state of the specification, developers have to determine the class identifiers of custom modules, knowing the number of module classes already defined by the executive (see Annex 2). To use explicit module class identifiers is not necessary at the C++ code level, where these identifiers are valuated automatically. However, no simple way has been currently identified to keep Replicode source code consistent with C++ source code.

2.4 Device Functions

Device functions are identifiers found in command objects (`cmd`) and used by modules to identify the operation they shall perform.

Device functions are defined as r-atoms whose structure is:

Descriptor	*Data*
`90`	[*unused*(1)|*r-opcode*(2)]

Device functions shall be defined in the source code, using preprocessor `!def` directives (file `usr.device.functions.replicode`).

2.5 The Application Object

The application object is a shared object containing the definitions for operators and modules. The prototype of the application object is:

```
// file an_application.h

#include “custom_operators.h” // declares the functions
                              // implementing the operators.

#include "../Core/application.h"

extern   "C"{
void     Init();
}
```

The application object contains also the definitions of `some_application_modules.cpp` (which includes `some_application_modules.h`).

The `Init` function shall load the user-defined operator functions in the `r_code::OperatorRegister`. User-defined overloads of existing operators shall also be loaded by `Init`, overwriting existing values.

The application object must load the shared object `r_core.so`.