Replicode: A constructivist programming paradigm and language

Version 2.0
Revision 4

Eric Nivel & Kristinn R. Thórisson

with contributions from

Nathaniel Thurston, Yngvi Bjornsson

RUTR-SCS13001
2013

Reykjavik University
School of Computer Science

Technical Report

1 Introduction

Replicode is a language designed to encode short parallel programs and executable models, and is centered on the notions of extensive *pattern-matching*[1] and *dynamic code production*.

The language is domain independent and has been designed to build systems that are *model-based* and *model-driven*, as *production systems* that can *modify their own code*. More over, Replicode supports the distribution of knowledge and computation across clusters of computing nodes.

This document describes Replicode and its executive, i.e. the system that executes Replicode constructions. The Replicode executive is meant to run on Linux 64 bits and Windows 7 32/64 bits platforms and interoperate with custom C++ code.

The motivations for the Replicode language, the constructivist paradigm it rests on, and the higher-level AI goals targeted by its construction, are described by Thórisson (2012), Nivel and Thórisson (2009), and Thórisson and Nivel (2009a, 2009b).

An overview presents the main concepts of the language. Section 3 describes the general structure of Replicode objects and describes pattern matching. Section 4 describes the execution model of Replicode and section 5 describes how computation and knowledge are structured and controlled. Section 6 describes the high-level reasoning facilities offered by the system. Finally, section 7 describes how the computation is distributed over a cluster of computing nodes.

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

The authors are grateful to Manuel Rodriguez, Carlos Hernandez Corbato, Ricardo Sanz and Bas Steunebrink for their constructive comments.

[1] Patterns target arbitrarily complex time series of objects. 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 compiled into a machine-readable form – r-code - that will be interpreted at runtime and reciprocally, be decompiled into the human-readable form. An r-code interpreter, called *rCore*, is a tiny virtual machine capable of executing r-code programs. rCores are part of the executive (the other essential part is the memory, rMem).

Systems coded with Replicode are distributed production systems, where knowledge is scattered across a cluster of computing nodes, and where programs react automatically to the occurrence of knowledge using pattern-matching to produce new knowledge. Such systems are *data-driven*, and are composed of a possibly (very) large number of concurrent programs interacting in a *global workspace*.

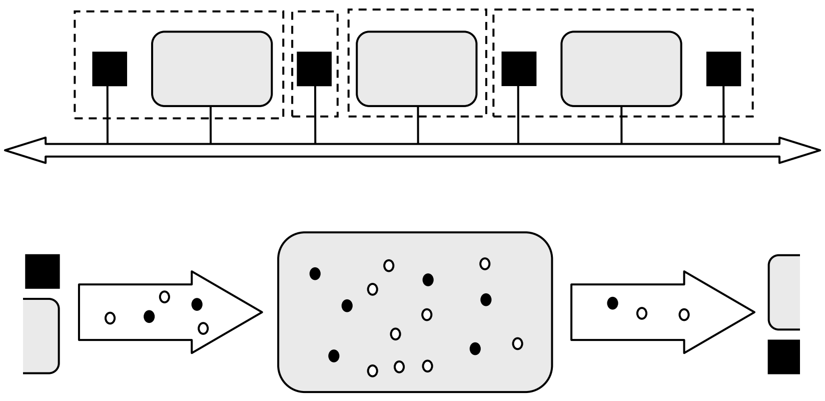

Figure 1 – Overview of a Replicode Application

Top: *an application is composed of knowledge encoded in Replicode and of external modules (custom C++ code, in black) called devices. An application is executed by several computing nodes (white rectangles, dashed outlines) interconnected by a messaging infrastructure (bidirectional arrow). Some of these nodes host an instance of the executive (grey rounded rectangles). Such instances communicate with devices and with each other, exchanging knowledge encoded in Replicode.*

N.B.: the messaging infrastructure is not part of the Replicode specification.

Bottom: *overview of the content of an instance of the executive. It contains many parallel programs (in black) that react on data (in white) in a local workspace (in grey). Programs and data are exchanged between all constituents of the system (devices and instances of the executive); these multiples instances form altogether a distributed global workspace.*

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 issued to I/O devices, process spawning and termination, etc).

Said architectures are said *model-based* in that sense that actions (productions) can result from evaluating executable models (forward models to predict, inverse models to reach goals). The selection of the models to be evaluated is not hard-coded, but is instead controlled by the execution of other models: in addition to be based on models, the architectures in question are also *model-driven*.

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

Replicode code is structured around the central notion of pattern-matching on knowledge, and enforces a *data-driven execution model*. Knowledge is represented as objects – i.e. instances of classes[2], some of which are user-defined, the others being part of the language axioms. The latter are the following:

Program Programs react to incoming objects. A program defines a pattern of time-constrained object sequences, including patterns and guards (conditions) on the individual incoming objects and produces new objects built from the incoming ones, said new objects being called productions. Programs are reactive, i.e. they perform as described whenever incoming objects match the time-pattern and guards. The transformation of inputs into productions is called *reduction*.

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

Marker Markers are tags indicating that a given object is in a specific relation with some 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".

Entity An entity 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. An entity does not carry any semantics at all: semantics are encoded in other objects that refer to entities (like markers, or models for example).

Ontology An object of this class denotes an axiom of the domain. It is a mere symbol used generally in markers to encode the fact that an entity has certain properties belonging to the class of the ontology. For example, as an alternative to the marker "has_color", we could define a more general marker class (mk.val an_object a_property a_value) where an_object is an entity, a_property is a member of the ontology of the domain, and a_value is the value associated to the property for the object; in our example we would code: (mk.val an_object color "blue").

Notifications The executive reflects the computational events by means of objects that describe reductions that have occurred in the system. Intuitively, this allows the system being aware of "what is going on". Notifications indicate, for example, that a program performed the reduction of some input objects, producing new objects at a certain time.

Device Devices are numerical identifiers for external software modules, i.e. components not written in Replicode, running independently of the Replicode executive in a given system, but interacting with code written in Replicode. Such devices are typically I/O devices (think device drivers), and generally implement functions that do not need to be modified at runtime.

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

Command Commands are the specification of a device, a device function and some arguments. They are executed by the devices.

Group The global workspace is organized as graphs of sub-workspaces called groups. Groups are objects that contain views on other objects. What distinguishes between objects and views is essentially that objects contain code (e.g. algorithms or data) whereas views contain data that qualify the code. Code is qualified by control values such as activation (controls whether or not a program can run) and saliency (controls whether or not code can be an input for a program). Groups essentially restrict the scope of the reductions that can be performed by its members: in short, the programs in a given group can only react

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

to input objects from that same group (see section 6 for more details) – however, this restriction of scope is fully controllable at runtime.

A given object can be projected onto (i.e. made a member of) many groups, and groups – being first-class citizens – can also be projected onto other groups.

Fact The Fact class is provided to support high-level first order logic programming. Essentially, a fact is a predicate pointing to other objects and carries additional qualifications: (a) a time interval specifying when the fact holds and, (b) a confidence value – a number in [0,1] that encodes an estimate of the certainty that the fact holds (notice that the confidence value is *not* a probability). Another similar class is provided, |Fact, for specifying the absence of facts.

Goal A goal object is the building block for defining a target state in a system. Goals are pointers to facts. Constraints are encoded as goal objects.

Prediction A prediction object is also a pointer to a fact and indicate that said fact has not happened yet but will happen in the future.

Composite State A composite state is a set of patterns specifying facts that shall occur at the same time. Composite states define guards (i.e. conditions) for controlling both the forward chaining and the backward chaining.

Model A model is a structure composed of two patterns: the first pattern specifies a fact that, when matched, triggers the production of a prediction, templated after the second pattern (forward chaining). Models can also be executed the other way around (backward chaining): when the second pattern is matched by a goal, the executive produces a sub-goal templated after the first pattern. Models also define guards for controlling both the forward chaining and the backward chaining.

In addition to object classes, Replicode defines structures - sets and expressions -, to be built from other structures, and eventually from atoms (see Annex 1 for the complete list):

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

Variables Identify and reference any code fragment.

Wildcards Used to ignore code fragments in patterns.

Numbers Floating point numbers.

Timestamps Integers with time semantics (in microseconds).

Identifiers Integers identifying instances of specific classes (devices, device functions and computing nodes).

Boolean values.

`nil` Nothing.

`|[]` Empty set.

`|nb` Undefined number.

`|inb` `Undefined imprecise number.`

`|us` Undefined time.

`|ius` Undefined imprecise time.

`|did` Undefined device identifier.

`|fid` Undefined device function identifier.

`|bl` Undefined Boolean value.

Replicode defines operators on structures and atoms: these are the usual arithmetic and logic operators.

Replicode allows developers extending the set of operators: this is achieved using an API, the Replicode Extension API.

Replicode also defines a preprocessor (in C++ style) for managing code definitions: this allows developers defining custom object classes and macros.

Last, Replicode defines a set of functions to control the executive. Examples of such functions are: `inj` (inject new objects), `eje` (eject existing objects to remote computing nodes) `ins` (instantiate a program).

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

Programs The code of programs has distinctive features, essentially:

- There are no explicit loops: 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: 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: disjunctions are achieved by running different programs/models targeting the different cases that would have constituted the operands of the desired disjunction.
- Programs react to salient knowledge only: present knowledge that would not be deemed salient enough still exists but will not be processed until it becomes salient. The saliency of knowledge results from a collective agreement reached by some other programs.
- Programs are prevented from executing whenever their activation is deemed too low. As for saliency, activation results from a collective agreement.
- Programs have no application-dependent internal states.
- Programs all run in parallel. Sequentiality and synchronization are implicit, as they result from objects having been produced by some programs to become the inputs of some others.
- Programs are also knowledge and can be reasoned about: their code is explicit, i.e. can match patterns defined by other programs.
- Programs can be produced dynamically and executed on the fly.
- Programs 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 exist for a reason that may no longer hold later.

Data Knowledge 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 are ephemeral, i.e. data have a limited lifetime, which can be controlled by programs. Without intervention of the latter, data are deleted from memory when their time to live expires. This time to live is called *resilience* and constitute one of the three main values used to control objects – the two others being saliency and activation.
- Once produced, data cannot be modified[3]. The memory holds the succession of occurrences of objects, 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 data acquire their relevance to programs by collective agreement. Data are said salient if they contribute noticeably to the achievement (or failure) of some function implemented by a collective of programs.
- The scope of data is controllable, i.e. data can be accessed (if they match the right patterns) by all the programs in a given (sub-)workspace. Scope restriction is however necessary to avoid the flooding of the system by possibly irrelevant

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

data. Scope is actually controlled (a) by pattern matching and, (b) membership in groups, i.e. objects can be projected onto groups, thus possibly hiding them from other groups. For a program in a given group, its inputs are chosen among salient objects in said group.

- Control over data can be propagated to other data. Objects usually reference each other, organizing knowledge in graphs. Replicode provides built-in facilities to propagate saliency changes from one object to other objects, members of the same graph.
- Application-dependent data types - and relevant operators - can be wedged in Replicode using an API.

Models Models are similar to programs, and bring additional features and semantics:

- In comparison to programs, models can be seen as higher-level constructs in the sense they add semantics to their outputs (goals, assumptions or predictions) whereas program don’t.
- Models can be executed both ways, performing either a deduction (the output is a prediction derived from a fact or a prediction taken as the input) or an abduction (the output is either (a) a sub-goal, derived from the goal taken as an input or, (b) an assumption derived from an input fact).
- Abduction and deduction are performed simultaneously, i.e. deductions and abductions can happen at the same time in a given model, as the result of the processing of different inputs.
- Models can control the execution of other models, and also, can reuse other models.
- Models are shared, i.e. they may contribute to the achievement of several goals and also, can capture the behavior of several actors in the environment.
- Models do not capture absolute truth: they contain control values that represent (a) the number of evidences, (b) the success rate and, (c) the derivative of the success rate. The success rate is the number of positive evidence divided by the number of evidence. The success rate is updated by the executive, as the model predicts events that are confirmed or not.
- As for programs, models can be active or not: the success rate is used as the activation value.
- Models that consistently perform badly are eventually deleted by the executive.
- Models that used to perform adequately and start performing badly indicate a change of context in the domain they encode knowledge for.
- With regards to pattern matching, models form a dynamic graph that produces predictions and goals. This graph is fed continuously with two simultaneous streams of, respectively, predictions and goals.
- Backward chaining can produce several sub-goals, given an input goal: these sub-goals may represent alternatives for solving the input goal and the system has to evaluate these solutions in order to commit to the best ones (according to its present knowledge). This evaluation is performed automatically by the executive: backward chaining triggers the simulation of sub-goals that runs for a limited time horizon. When the time allocated to simulation is spent, the system examines the results of the simulation (these results are predictions) and eliminates the solutions that (a) end in the failure of the goal and, (b) result in the failure of another goal that would be deemed more important than the present goal.
- Models are either given by the programmer or learned by the system, that is produced dynamically as the system accumulates experience in its environment.

3 Pattern Matching

This section describes how pattern matching is defined and performed, with respect to the general structure of objects encoded in Replicode.

3.1 Objects

Class Definition Object classes are defined as pre-processor directives (like macros in C++) that declare the identifier of the class, and class members (name and type). Replicode is *weakly typed*: types can be atomic types (numbers, timestamps, Boolean values, etc.), structured types (classes), but types can also be *less specific* (e.g. "expression", "set", "anything" are valid types).

Object classes are defined as follows:

(*class-name* m_1:t_1 … m_n:t_n)

where class-name is the name of the class, m_i are the names of the class members and t_i their types.

For example, the definition of a class encoding the position of an object in a geometrical space could be:

```
(mk.position object: position:vector3 ...)
```

where `mk.position` is the identifier of a user-defined class (e.g. "position marker") and `object` is a reference to an object having the position `position`. Notice that no specific type is specified for `object`: it can be an object of any class. The class `vector3` could be defined as:

```
(vector3 x:nb y:nb z:nb)
```

where `nb` is the atomic built-in type for floating-point numbers.

`view` is a pre-defined class holding the control values of an object. It is defined as a set:

```
[ijt:us sln:nb res:us grp:grp org: sync:nb]
```

where `ijt` is the time when the object was produced (called injection time, expressed in time units, type `us`), `sln` is the saliency of the object, `grp` is the group where the object belongs, `org` is the group or the remote computing node where the object is originating from, and sync is a number (0 or 1) that indicate if the object will be reduced when it becomes salient or, on the contrary, when it is simply salient. N.B.: Replicode defines variants of `view` for specific object classes (like programs, see section 4.1 below).

N.B.: the notation `...` is not part of the language, it is just shorthand for "other data omitted for the sake of clarity"

Class members can be accessed (read-only) using the following notation:

an-object.*member-name*

To continue with our example, to access the injection time of an object `o`, instance of `mk.position`, one can write:

```
o.vw.ijt
```

Projection Objects belong to groups – they are said to be *projected* onto groups – and possibly so to several groups. Any object class contains a standard member named `vw` of the class `view` (see example above), that holds data describing the object from the vantage point of a particular group, see figure 2 below.

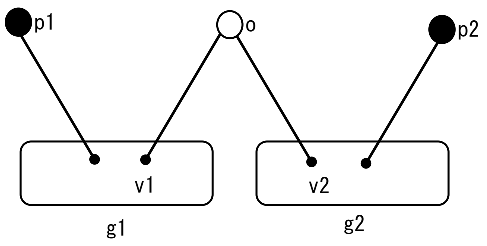

Figure 2 – Object Projections

An object o is projected onto two groups `g1` *and* `g2`*. Each group defines a particular set of control values for* `o`*, called its views in* `g1` *and* `g2` *(*`v1` *and* `v2` *in the picture). Shall* `o` *become an input for a program* `p1` *in* `g1`*,* `o.vw` *would be valuated with* `v1`*, whereas it would be valuated with* `v2` *when becoming an input for a program* `p2` *in* `g2`*. The executive is responsible for valuating the appropriate view of objects depending on which group they are accessed from.*

N.B.: objects have a member called vws for "view set", that contains all its views; for example, `o.vws` *contains* `v1` *and* `v2`*.*

Loading Declaring a class in a source code file does not create any instance of the class. One needs to specify the objects that are to be instantiated and loaded in memory. This is part of the initialization of the system (see Annex 2 for details). So, in addition to class declarations, the source code must contain initialization commands of the form:

optional-instance-name`:`*an-object a-set-of-views*

where *an-object* is an expression. The *set-of-views* argument specifies the initial projections of *an-object*.

For example:

```
g1:(grp ...) [[now 0 50 root nil ...]]
g2:(grp ...) [[now 0 forever g1 nil ...]]
m:(mk.position self (vector3 0 0 0) ...) [[now 1 10 g2 nil] [now 0.5 20 g1 nil]]
```

`now` is an operator returning the current time, i.e. the time when the expression it is embedded in is evaluated by the executive.

`g1` is a group initially projected on a default group (called `root`, see figure 3 below): its initial saliency is 0 and its resilience is 50 times the time base defined by root (see below).

`g2` is another group, initially projected onto `g1`: its initial saliency in `g1` is 1 and its resilience is infinite (`forever`).

`m` is a position marker initially projected on `g2`: `self` is a reference to a predefined instance representing the system itself, the initial saliency of `m` in `g2` is 1 and its resilience is 10 times the time base defined by `g2`.

`m` is projected again, this time onto `g1`, with another view.

Circular references of objects are not allowed.

Resilience Objects are permanently deleted from memory when (a) no view exists anymore and, (b) there is no other object referencing it. An object is referenced by another when the latter holds a pointer to it. In our previous example, a position marker holds a reference on the object it stores the position of. The resilience of a view is expressed as a multiple of a base period defined by the group where the view is defined. This base period is called the update period of the group: it the period at which the group performs some bookkeeping operations.

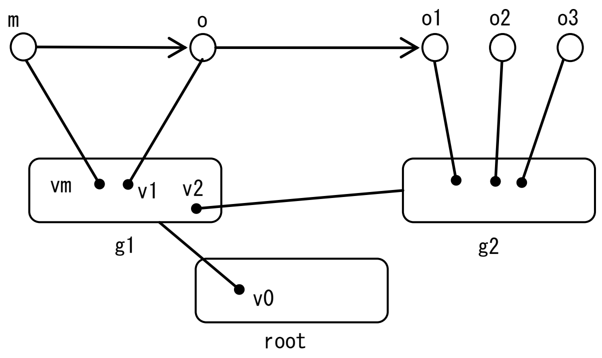

Figure 3 – Object Resilience

An object o is projected onto one group `g1`*. A marker* `m`*, also projected on* `g1` *(view* `vm`*), and holds a reference to* `o`*. The resilience control value is defined locally per object and per group, and is in fact the resilience of the view on the object. Shall* `v1` *reach a resilience of 0, it will be deleted, leaving* `o` *with a single reference from* `m`*. When the resilience of* `vm` *drops down to 0 as well, then* `m` *will be deleted, triggering the deletion of* `o`*.*

When the resilience of `v2` *drops down to 0,* `g2` *is deleted and all views it holds:* `o2` *and* `o3` *would be deleted as well – not* `o1`*, if* `v1` *is still resilient. Replicode defines an axiomatic group called* `root` *that enjoys infinite resilience:* `root` *acts as the anchor for other groups – in the example depicted, g1 would be projected on* `root`*, and would be deleted when the resilience of its view in* `root`*, drops down to 0. Deleting* `v0` *triggers the deletion of everything else.*

When an object has no views anymore but is still referenced by other objects, its view set becomes empty but the object still exists. However, lacking views, it cannot become inputs for programs.

Groups are described in section 5.

3.2 Patterns

Pattern matching is the main way to valuate formal arguments defined in programs. A pattern is specified as an expression that may contain variables and wildcards. The structure of a pattern is:

(`ptn` *skeleton guards*)

where *skeleton* defines the structure of the object to be matched, and *guards* is a set of conditions on said object. `ptn` is the opcode identifying the expression as representing a pattern. For an object to match a pattern, its structure must match the skeleton and it must satisfy *all* the guards specified in the pattern.

For example, using the class mk.position defined in the example above, such position markers match a pattern like:

```
(ptn (mk.position what: where: ::) |[])
```

This pattern will be matched only by objects of class `mk.position`. The variables `what` and `where` would then store the actual values of the object being described and its position. In other words, for a position marker `m`, `what` would refer to `m.object` and `where`, to `m.position`. Notice that the rest of the constituents of the input object (`m`) is ignored (tail wildcard `::`). In this example, the set of guards is empty (`|[]`).

Replicode allows matching patterns on sets. The operator `red` (reduction) admits three arguments: (a) the set to be filtered, (b) a positive pair containing a set of patterns and a set of productions and, (c) a negative pair of patterns and productions. The result of such a reduction is a set containing the positive productions computed from the elements that match at least one of the positive patterns and the negative productions computed from the elements that do not match at least one positive pattern but match at least one of the negative ones (see Annex 1).

Patterns can also specify the structure of objects that are *not* expected. This allows programs to output productions upon 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 patterns in programs without specifying the class of the expected object. For example, it is not allowed to write input patterns to catch “any kind of object such as …”. This restriction applies only to patterns featured in programs, not to patterns defining template arguments (defined below) or to pattern passed as arguments to the `red` operator.

Replicode allows the definition of so-called anti-patterns (`|ptn`), that is, patterns that will be matched by any object that does not match the skeleton, or does so but does not match all the guards specified in the pattern.

4 Execution Model

This section describes the behavior of 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
[patterns]; a set of template arguments
[patterns]; a set of patterns to match individual input objects
[expressions]; a set of guards applying to the inputs objects
[productions]; a set of productions
view
other-data
)
```

N.B.:

```
[]
   a
   b
```

denotes a set of two elements a and b; this also can be written `[a b]`.

A program specifies constraints on the input objects: these constraints are conditions taking as arguments any part of any of the input objects and in particular, they can express correlations *between input objects*.

A program has read-only access to a set of runtime data (maintained by the executive) including (see Annex 1 for the complete list):

- view:
  - the time of its construction: injection time (`ijt`);
  - saliency (`sln`);
  - resilience (`res`);
  - the group where the view is defined (`grp`);
  - the group that created the program (origin, noted `org`);
  - the synchronization mode (`sync`) that indicate whether the program can be an input only when becoming salient or when just being salient.
  - activation (`act`);
- a pointer to itself: `this`.
- a set of all the markers that reference it (`mks`)
- a set of all its views in the groups it is projected onto (`vws`).

Programs 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 functions of some of the template arguments. Template arguments are specified by patterns: they express constraints on the objects to be used to parameterize the program. As defined, a program cannot be executed without being passed actual template arguments. Replicode provides an operator (`ins`) to instantiate programs: it constructs another object, called an *instantiated program* (`ipgm`), that shares the code defined by the program and contains the actual template arguments. The structure of an instantiated program is:

```
(ipgm
code; a reference to the program having been instantiated
[actual-arguments]; args
...
)
```

4.2 Program Execution

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

I. whenever input objects are presented to an instantiated program by the executive and each input object individually matches its respective input pattern, then,

II. check the guards: if none of them returns `nil`, or any undefined value (e.g. `|[]`, `|nb`, etc.) then,

III. execute the code specified in the set of productions.

Notification is provided by the executive, and describes which instantiated program successfully matched which input objects and produced which objects at which time.

The executive handles the case when a program outputs an object, identical to an existing one: - identity being defined as the identity of the code: the executive eliminates any duplicates.

Replicode also defines anti-programs (`|pgm`), programs that react when *no* objects match the inputs over a specified period (the time scope `tsc`, see below).

Programs can also not define any input pattern and be meant to output their productions at specific times (like timers, using the time scope). The executive will execute input-less programs each time the time scope is elapsed.

Execution Overlays Regarding the enforcement of timing constraints (found in the guards section) on the inputs of a program, the executive performs according to rules exemplified in the following situations:

I. In the example of an instantiated program iP whose code contains only one input pattern IP and no guards, if an object o_1 is received, then the executive spawns a copy of iP to process o_1, and is then ready to spawn other copies of iP upon reception of subsequent objects o_i matching IP while o_1 is being processed: the execution is *overlaid* (and copies of an instantiated program are called its *overlays*). This rule applies in a more general way to instantiated programs featuring any number of input patterns and guards (see other rules below).

II. In the example of an instantiated program iP built from a program `P` like:

```
P:(pgm
|[]; no template arguments
[]
   ptn1:(ptn o1:skel1 guards1)
   ptn2:(ptn o2:skel2 guards2)
[]; guards
   (> o2.vw.ijt (+ o1.ijt 10)); ijt=injection time member
   (< o2.vw.ijt (+ o1.ijt 20))
   ...
...
)
```

iP receiving inputs like:

`a1` (matching `ptn1`) at time t,

`a2` (matching `ptn1`) at time t+5,

`b` (matching `ptn2`) at time t+22

triggers, upon reception of `a1`, the spawning by the executive of a copy of iP, iP1, whose code would be equivalent to:

```
(pgm
|[]
[]
   (ptn o2:skel2 guards2)
[]
   (> o2.vw.ijt (+ a1.ijt 10us))
   (< o2.vw.ijt (+ a1.ijt 20us))
      ...
...
)
```

where o1 is valuated by a1. According to the rule from the example 1 above, upon reception of a2, the executive spawns another copy of iP, iP2, where o1 is valuated by a2. iP, iP1 and iP2 now run in parallel.

Upon reception of b by iP1, the timing constraint is violated and the executive kills iP1.

Upon reception of b by iP2, the timing constraint is respected and further processing by iP2 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 iP1, the timing constraint would be respected and iP1 would output a production, a function of a1 and b. In a similar way, iP2 would output another production, a function of a2 and b.

IV. An instantiated program defines a time scope (tsc), i.e. a limit in time beyond which an overlay is automatically killed by the executive. This means that a program will not attempt to match any input older than tsc.

V. Overlays are killed when the instantiated program looses activation or dies itself. However, any ongoing computation will not be terminated before completion. In other words, if an overlay is to be killed while being currently producing objects, these will be injected before the overlay is actually killed.

From the vantage point of other programs, overlays simply *do not exist*, i.e. overlays are hidden by the original instantiated program and the notification of the reduction (see below) refers to the original instantiated program, not to any of its overlays.

Notification of Reduction Each time an overlay ouptuts productions, the reduction is notified by the executive by injecting markers (class mk.rdx) referring to the program that performed the reduction, and containing the sets of both its inputs and its productions. The structure of such a notification is:

(mk.rdx code inputs:[] prods:[] *other-data*)

where code is a reference to the instantiated program, and inputs and prods are sets of objects. For example, assuming a program P like:

```
P:(pgm
|[]
[]
   (ptn a:(...) ...); pattern1
   (ptn b:(...) ...); pattern2
[]
   (inj [(...) ...]); command1
   (inj [(...) ...]); command2
...
)
```

then, when an instantiated program iP (built from P) reduces two input objects o1 and o2 (matching, respectively, pattern1 and pattern2), then the executive injects a reduction marker as follows:

```
(mk.rdx
iP; code, i.e. a reference to (ipgm P |[])
[o1 o2]; inputs
[]; prods
   (evaluated_command1)
   (evaluated_command2)
...
)
```

The commands featured in the reduction marker are embedded expressions. These expressions are evaluated, i.e. are the result of the evaluation of the commands at production time. For example, if, in P, command1 was:

```
(inj [a_new_object:(mk.position a ...) ...])
```

then in the reduction marker, evaluated_command1 would be:

```
(inj [a_new_object ...])
```

where `a_new_object` would be a reference to:

`(mk.position` *reference-to-o1* `...)`

Dynamic Program Production Programs are state-less, that is, programs do not encapsulate private persistent application-dependent data[4].

This means that the only way for programs to memorize application-dependent states is for them to refer to common knowledge, i.e. objects that are accessible by any other program. In other words, programs refer to states by pointing to external objects.

To memorize a state, there is also the possibility to *alter* a program's code dynamically that is - more accurately - to produce a *variant* that would embed some prior state: the variant would be built as a function of some previous input objects (the so-called state) caught by the original program. For example, let's consider the following case: an instantiated program iP0 continuously outputs the position of an entity – a ball, under the control of yet another program. We need a program `iP1` that outputs an object `L` holding the last known position of the ball – and also, there is no need to produce `L` when the ball is stationary. `L` is an instance of a class defined as `(mk.last_known data:mk.position ...)`. Here is a possible encoding for `P1` (which would be instantiated to form `iP1`) – `P1` depends on another program `P2`:

```
P1:(pgm
[]
   (ptn (mk.position ref_b: ref_pos: [ref_t: ::] ::) |[])
[]; inputs
   (ptn m:(mk.position b: pos: [t: ::] ::) []
      (= b ref_b)
      (> t ref_t)
      (<> pos ref_pos)
   )
|[]; guards
[]; productions
   (inj [(mk.last_known m ...)]); builds L
   (inj [(ins P1 [b pos t])]); instantiates P1 with b, pos ..
and t as actual template arguments
   (set [this.vw.res 0]); kills the original
)

P2:(pgm
[(ptn ref_b: |[])]
[]; inputs
   (ptn m:(mk.position b: pos: [t: ::] ::) [(<> b ref_b)])
|[]; guards
[]; productions
   (inj [(ins P1 [m]) ...])
   (inj [(ins P2 [b]) ...])
   (set [this.vw.res 0])
)

iP2:(ipgm P2 [nil] ...) [[...]]; initial injection of iP2
```

Notation:

`;` 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 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 `iP1` to zero. The executive accumulates potential similar requests on `iP1` issued in a (tunable)

[4] Programs however, do hold control values in their views (saliency, activation, resilience, injection time) but only have limited access to these: the executive maintains (writes) these values. For this reason, and also because these values are not related to the application domain, they are not considered internal states.

time window and decides a final value accordingly.

P1 is parameterized by the last known position marker referring to *a* ball. When it catches another position marker m for the same ball, it produces a new instance of mk.last_known, instantiates P1 with m and kills itself. In this respect, notice that this in P1 refers to the instance of P1 being executed – this evaluates to an instantiated program.

P2 is parameterized by a ball, ref_b. The role of instances of P2 is to create instantiations of P1 for any new ball b – different from ref_b - for which the position is reported, and to produce another instance of P2 that would react on balls other than b.

Manual injection of objects is performed by defining the code of the object, followed by a set of views.

Notice that P1 could be augmented so that it would also kill previous occurrences of L; in our example such occurrences would die "naturally", i.e. after the delay specified by their resilience value (not shown in the code).

Explicit program instantiation from template arguments is not the only way to generate programs. Any program can build new code from scratch and generate corresponding new programs. For example:

Assuming a vector class vec3 and a position marker defined as:

```
!class (vec3 x.nb y:nb z:nb)

!class (mk.position (_obj {obj:ent pos:vec3}); points to an.. entity and holds a
vector.

P1:(pgm
|[]
[]
   (ptn (mk.position e1: v1: ::) |[])
|[]
[]
   (inj []
      P2:(pgm
      |[]
      []
         (ptn m:(mk.position e2: v2: ::) [<> e1 e2])
      |[]
      []
         (set [m.vw.res 0])
         (inj []
            (mk.position e2 (¥ (+ v1 v2))) 1)
            [(¥ now) 1 forever root nil]
      ...
      )
      [now 1 forever root nil]
   (inj []
      (ins P2 |[] [now 0 forever root nil 1])
   )
...
)
```

P1 builds a program P2 that adds P1's input vector v1 to P2's own input vector v2. Notice the use of the ¥ operator that prevents evaluation (similar to the quote operator in LISP). The expressions in P2 that follow the \ operator are not evaluated when P1 is. They will be evaluated when P2 is actually injected, instantiated and performing a reduction.

Batched Productions When a program outputs some productions, the executive stores them in memory and make them available for further referencing by other productions. This is performed as a *batch*: first, all productions are computed, and then they are made available. This allows productions to refer to each other. For example, a program can create a new object n and a marker `m` on `n`:

```
(pgm
...
[]; productions
   (inj [n:(...) ...])
   (inj [(mk.whatever n ...) ...])
...
)
```

Circular references are forbidden – caveat: no check is enforced at runtime in this version of the specification and the behavior of the executive in such cases is undefined.

The general form of a production is either a command to an external device:

`(cmd device function [`*arguments-of-the-function*`])`

or an internal command (`icmd`), i.e. a command to be performed by the executive itself. The injection commands used in the examples are actually macros and are expanded as follows:

`(icmd _inj [`*object view*`])`

where `_inj` is the actual function, *object* is the object to be injected and *view* is a view for the object[5]:

```
[ijt:us sln:nb res:nb grp:grp org: sync:nb]
```

it contains an injection time `ijt` - to be interpreted as “inject the object not before `ijt`” – and a destination group (`grp`), where the object will be projected using the specified view. `org` will be valuated by the executive, and can initially be set to `nil`. It is possible to set injection times in the future, i.e. later than `now`.

Commands are not system objects, i.e. they don’t exist freely in memory: they are always embedded in code fragments. When the fragment is a production, the executive interprets these commands as actual function calls in case the specified device is the executive, or as messages to an external module otherwise. In the latter case, the messages carry the r-code corresponding to the command, and the device shall transform this r-code into its own internal representation.

Execution Eligibility When an object is produced, its producer assigns it an *initial* saliency value which can be changed over time by other programs. An object is eligible to become an input for any program in one of two cases, depending on its views’ `sync` parameter value:

> 0 - Synchronization on front: 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 by the executive as a potential input to all active programs. This means that objects become potential input data when they *cross a saliency threshold*: once salient, they cannot be processed by programs anymore – to do so, they would have to lose and regain saliency.
>
> 1 - Synchronization on state: an object becomes an input for any program whenever it is salient, regardless of its crossing the threshold.

When a program becomes deactivated (see section 6.1), 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 *are* or *become* salient after it wakes up – depending on the objects’ views’ respective sync parameters.

Expressivity Combining the two kinds of programs (`pgm` and `|pgm`) and the two kinds of patterns (`ptn` and `|ptn`) yields for main modes of execution. For the sake of simplicity let’s consider a pattern P designed to match objects having the structure S. We denote by |P the corresponding anti-pattern. let’s also ignore timings, guards and multiple patterns. Then:

> I. a program P1 that uses P will reduce any object of structure S as soon as it is eligible for reduction;
>
> II. a program P2 that uses |P will reduce any object that does not have the structure S, as soon as it is eligible for reduction.
>
> III. an anti-program P3, with a time scope T and using P will build its productions any time when, after T, no object having the structure S has been eligible for

[5] There are specific kinds of views for instantiated programs and for groups that add, respectively, an activation or a visibility value to the set of view members.

reduction;

IV. an anti-program P4, with a time scope T and using |P will build its productions any time when, after T, no object having not the structure S has been eligible for reduction; in other words, P4 will build its production if it observed either nothing or only objects that had the structure S.

Notice that only P3 and P4 can react to the total absence of objects: P1 and P2 need actual inputs to trigger their reductions. Notice also that P4 is *not* equivalent to P1: P4 imposes the constraint of maintaining its requirements (|P) over a period of time (T), whereas P1 would react immediately upon the occurrence of an object having the structure S.

5 Groups

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 a few individuals, up to the scale of an entire system. This section presents the facilities offered by Replicode to control computation at both the individual and collective scales.

5.1 Individual Scale

Objects in Replicode are controlled individually by a set of control values held by their views, namely, resilience, saliency and – for instantiated programs – activation.

The resilience is a delay (expressed as a multiple of a period defined by the group where the view lives) after which, if left unchanged, the view will be permanently removed from the memory by the executive – although, at the developer's convenience, objects can also 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 instantiated programs or not, whereas the activation is a value (also in [0,1]) that controls whether an instantiated program will receive inputs or not (i.e. as discussed in section 4.1, whether it is “asleep” or “awake”).

Activation and saliency are evaluated by the executive against thresholds defined in groups: below (or equal to) its respective threshold a control value is considered “off”, above “on”.

Any program in a given group can request modifications on any control value of a particular object. The executive will gather these requests and, at a certain frequency, will use the average as the final value: each group specifies a particular update period (`upr`) for the control values of the objects it contains. 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. This way of gathering requests and computing final values from them is called *mediation*. Notice that thresholds defined in groups are also control values and are also mediated.

5.2 Collective Scale

Basically, a group is a set of views on some objects: these objects are said to be *projected* onto the group. A group gathers objects that collectively perform some function and/or represent some particular knowledge. From an architectural perspective, groups bring the following capabilities to a system (the underlying mechanisms are presented after this overview):

- **Sub-systems:** a group behaves like a small production (sub-)system in itself. In particular, each group defines its own thresholds for controlling locally the saliency and activation values. Generally, objects projected onto a group are not visible from programs that are not projected in this group, nor can they access objects that are not in the same group.

  Distributing objects amongst groups allows reducing the number of candidates for pattern matching: this is not only a performance issue but also a design issue, in the sense that such a distribution allows preventing some reductions to occur that would have otherwise been possible but not desired.
- **Controllable scope:** the scope of objects in groups can be adjusted dynamically. Since groups are regular objects, they can be projected onto other groups: when a group G1 is projected onto another G2, under certain conditions, the content of G1 can be made visible to the programs in G2. This relation is called *group visibility* and can be adjusted at runtime. It is also possible to inject objects into other groups, and groups can be parameterized so that the salient content of a group G1 is copied into another group G2 when G1 becomes visible from G2 (this feature is called *copy upon visibility*).

  Controlling the scope of objects allows defining dynamically reduction pathways across the system.
- **Local time scales:** each group defines its own update period as multiples of the system’s base period.

The update period is the period at the end of which all mediated values are assigned a final value by the executive. For example, it is at the end of each update period that saliency and control values change; it is also when views are deleted from memory when their resilience expires.

- **Control over collectives:** the content of groups can be controlled as a whole in two ways.

  First, a group defines two additional control values – *c-saliency* (content saliency) and *c-activation* (content activation) – that allow changing the saliency/activation status of *all* objects in the group. These values are controlled using thresholds also defined locally for a group.

  Second, groups also implement automatic decay on either the saliency of its members or on its own saliency threshold: this allows programming cycles of changes in saliency, this being equivalent to defining cyclic eligibility of knowledge (synchronized on front) as inputs for programs.

  Control over collectives achieves the same results as controlling individually each group member, but reduces drastically the number of operations involved and the number of programs that would have been needed to perform these operations.
- **Propagation of saliency:** under certain conditions, saliency changes on an object are propagated by the executive to its markers – and, possibly, to the objects referred to by these markers.

  In short, saliency can be propagated within graphs of related objects: these graphs would then constitute graphs of semantically related objects (the prototype of such semantic clustering is an associative memory).

  Notice that this current specification does not defines the propagation of values other than saliency. In particular, the propagation of activation values is not defined.

Sub-systems Sub-systems are implemented as groups. They are first class objects and as such, are injected into other groups using the injection command (`inj`). Groups define their own mediated values (for example, thresholds on activation and saliency) and these values can be modified by programs, as they would do for the control values on individual objects.

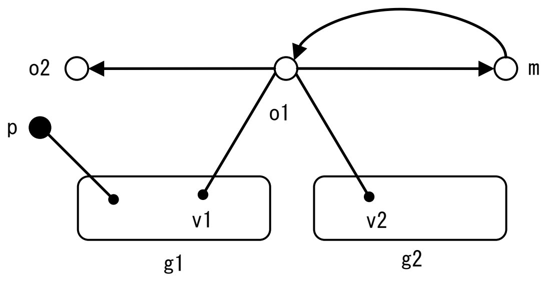

Figure 4 – Sub-systems

An object o1 *is projected onto two groups* g1 *and* g2*. As defined earlier (section 3.1),* o1 *will be viewed from* g1 *and* g2 *with, respectively, different sets of control values, i.e. the views* v1 and v2*.* o1 *also holds a reference to another object* o2*. If a program* p *in* g1 *acesses* o2 *via* o1*, then it would see* o2 *with a nil view, regardless of the actual projections of o2. This also holds for a marker* m *on* o1*. If* p *accesses* m *via* o1.mks*, m would appear to* p *with a nil view.*

g1 *and* g2 *define their own thresholds on these control values and, as a consequence,* o1 *can be salient in* g1 *but not in* g2*.*

As depicted, g1 and g2 are not related and their content is hidden from each other. This means that no program in g1 *can process inputs from* g2 *and vice-versa.*

If an object is injected in a group for the first time, and is salient in that group, this has the same effect as for an existing object to cross the saliency threshold upward. In other words, the newly injected object is eligible for processing as an input. In additon, the executive notifies such a first-time injection (mk.new).

When an object is created, the executive checks for identical existing objects among all objects present in the same node. The identity that is checked for is the identity of the code of the object and of the references it holds to other objects - its markers and views are not checked. When the new object turns out to be identical to an existing one, the latter replaces the former. This check applies to objects and therefore does not depend on group membership.

When an object is injected in a group, the executive performs another identity check, this time among all the views exisitng in the group. In case the object is already projected onto the group (a view on the object is already defined), then the executive replaces the old view by the new one.

Replicode defines a special group called root: it is a group with infinite resilience meant to gather - directly or indirectly – all objects in the system. root is never deleted, regardless of the number of references to it. N.B.: there exit two other default groups that are never deleted either (see section 7).

Controllable Scope The injection command admits an argument that references a target group, that is the group where an object is to be injected (i.e. projected). Either this target group is caught by a program as an input, or it is a hard-coded reference. For example, in figure 4 above, a program p in g1 could inject an object directly in g2: since g2 is not projected on g1, g2 cannot be an input for p, and p would have to explicitly refer to g2 (i.e. hold a pointer to it).

Groups are not programs and thus, their views do not include activation values. Instead, they contain another mediated value called visibilty (vis) that controls whether or not the content of a group, when projected onto another, is visible from the latter group. Like other mediated values, this value is compared against the visibility threshold (vis_thr) defined by the group. Visibility is *not* transitive.

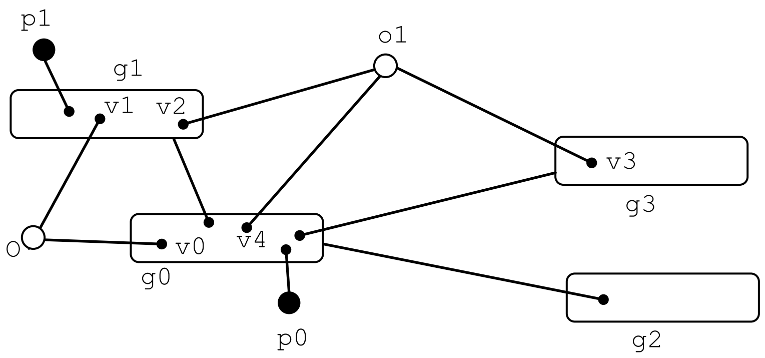

Figure 5 – Visibility

An object o *is injected in* g1 *with a view* v1*; it is also a member of* g0*, with a view* v0*. Assuming two active programs* p0 *and* p1 *respectively in* g0 *and* g1*:*

- *if* o *is salient in* g0*,* p0 *can react to* o *with the saliency v0.sln;*
- *if* o *is salient in* g1*,* p1 *can react to* o *with the saliency v1.sln;*
- *if* g1 *is visible in* g0*,* p0 *can react to* o *with the saliency v0.sln;*

For another object o1 *projected onto* g0*,* g2 *and* g3*:*

- *if* o1 *is not salient in* g0*, but is salient in* g2 *and* g3 *and* g1 *and* g3 *are visible in* g0*,* p0 *can process* o1 *twice, each time with a different saliency, i.e.* v2.sln and v3.sln *– assuming* o1 *is or becomes salient in its respective groups and matches one of the inputs of* p0*.*

Visibility is not transitive: if g0 *is visible in* g2 *and* g1 *in* g0*, then* g1 *is still not visible in* g2*.*

In case an object is visible from multiple groups (thereafter called sources), it will be eiligible for becoming an input if it is or becomes salient in at least one of these sources, and if it is actually processed by programs its saliency will be taken as the value held by its views from the sources where it is visible from. This means that said saliency values must be comparable, i.e. be expressed as a ratio to the threshold defined in their respective sources. In such cases, the executive applies a correction to convert the saliencies defined in the sources into values that can be compared against the threshold defined in the group where the object is processed.

Shall an object become visible from a group G for the first time, this would be treated exactly as if the object was injected in G for the first time: notification would occur in the groups where the object is being injected, and said notifications would be visible from G.

Groups define a view member called cov (for "copy on visibility"), not mediated. When a group (thereafter called source) is visible from another group (the target) and in this group, the source's `cov` is set to 1, then its content will be copied into the target. The following rules apply:

- the target group must be c-salient and c-active (see sub-section Control over collectives).
- only the salient content of the source is copied into the target.
- as for visibility, the control values of the objects being copied are converted into values suitable for comparison in the target group.

Local Time Scales Mediated values are computed by the executive over a period called update period, defined by each group (member `upr`). They are computed as follows: at the end of each update period, the requests for change (issued by programs using the commands `mod` and `set`) are averaged and the result is added to the value to be controlled. Some objects (synchronized on front) can be processed when they cross the saliency threshold in the group and this is checked at the end of each update period. As a consequence, modifying the update period of a group has a direct impact on the reactivity of its active content.

When visibility is involved – like when a group `g1` is visible from another group `g2` – the final mediated values of visible objects are computed at the end of the period specified by the group performing the reduction (e.g. `g2`).

Control Over Collectives As for any object, a group `g1` - projected onto another group `g2` - is said salient when its saliency in `g2` is above the saliency threshold defined by `g2`. The group's saliency controls the saliency of the group as an object, *not the saliency of its content*. To control the saliency or activation of the content of a group (thereafter referred to as c-salinecy and c-activation), four additional control values are provided:

- `c_sln` and `c_sln_thr`: define respectively the saliency of the content of the group and the threshold to which the former is compared to determine the saliency of all group members;
- `c_act` and `c_act_thr`: the same as above, defined for the activation of the members.

When a group is not c-salient, none of the objects it contains can be inputs for any program in the group. When a group is c-salient, the objects it contains can be inputs if they are salient themselves.

When a group is not c-active, none of the programs it contains can run. When a group is c-active, its programs can run if they are active themselves.

A group's c-saliency takes precedence over the saliency of its members. This means that c-saliency acts like a mask on the individual saliencies of the group members. The same observation holds for activation vs c-activation.

When multiple changes to saliency, saliency threshold, c-saliency, and c-saliency threshold are made during a single update, an object is presented as an input to programs if, before the update, the object was not salient and/or the group was not c-salient; and after the update, the object is salient and the group is c-salient.

When, at an update period, both the group c-activation changes and the activation of a member changes, then the member's activation is computed, then the c-activation of the group. If the group is c-active, then the member starts receiving inputs. If the group is not c-active, then the object does not receive any inputs regardless of its own activation.

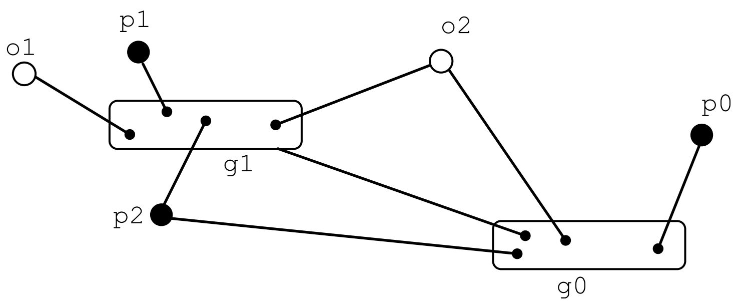

Figure 6 – c-saliency and c-activation

If g1 *is not c-salient,* o1 *cannot be an input for* p1*, even if* o1.vw.sln > g1.sln_thr*. If* g1 *is not c-salient,* o2 *cannot be an input for* p1*, but can still be an input for* p0 *and* p2*, if* o2.vw.sln > go.sln_thr*.*

If g1 *is not c-active,* p1 *cannot process anything, even if* p1.vw.act > g1.act_thr*. If* g1 *is not c-active,* p2 *cannot process anything in* g1 *but can in* g0 *if it is active in* g0*.*

As mentioned earlier, an object synchronized on front is eligible for processing when its saliency crosses the saliency threshold upward. Consequently, once salient, such an object will never become an input again, unless its saliency crosses the threshold downward and upward again, or the threshold itself gets higher than the object's saliency and lower again. Another consequence can also be deduced from the eligibility rule: if an object that is already salient in a given group is injected again, it will not be processed again. However, depending on the programmer's intention, it could be advantageous to treat this renewed injection as an indication of saliency, intuitively "the object occurs again to the mind of the system". As a convenience, groups offer extra parameters to define an automatic decay of the saliency values of all its members – this way, the saliency of an object will decrease over time and a new injection of the same object is likely to pass the eligibility check. This decay can be applied either to the saliency values of the group's members or to the saliency threshold of the group. The following parameters specify the decay function:

- dcy_per the target percentage of the control value: if 0, no decay will be performed; if negative, the percentage is interpreted as a decrease (decay); if positive, as an increase.
- dcy_tgt what the decay shall apply to: if 0, applies to the saliencies of the members, if 1 to the threshold of the group.
- dcy_prd the period of the decay, expressed as a multiple of the update period defined for the group.
- dcy_auto a flag indicating whether or not the group shall resume decay after the decay period has expired.

For example, a decay can specify a behavior like: "decrease saliencies down to 80% of their value (taken at the last injection time), within 4 times the update period". In this example, if an object o is injected with a saliency s1 during a period P, 4 periods after P, the decay will not be applied anymore, and at P+5, o.vw.sln=0.2*s1 - assuming no explicit changes by mod/set have been issued by some programs. If during these 4 periods, say during period P+3, there is a new injection of o with a saliency s2, two cases can arise:

a - if s2>o.vw.sln at P+2, the decay is reset, using s2 as the initial saliency for o, and will thus apply until P+8.

b - if s2<o.vw.sln at P+2, the decay continues on as if nothing happened.

Technically, this means that during each update period, the saliency is decreased by 20% of what it was the last time some program injected the object in the group. In other words, this is equivalent to one execution of:

(mod [o.vw.sln ((* o.vw.*sln-at-last-injection-time* -0.2)])

per update period. Such a modification competes with the explicit calls to set and mod issued by any program.

Decay parameters (as any other control parameters in groups) are not mediated (i.e. the last to talk wins).

Propagation of Saliency When some objects are related to eachother (think of Proust's cake), it can be useful to express this relation in terms of concrete control: if two objects o1 and o2 are in a semantic relation, then when o1 becomes salient one may expect o2 to also become salient. Replicode addresses this issue in a general way: changes of an object's view saliency can propagate to the objects it is related to, namely its other views and the views of its markers.

Each object class defines one additional member (psln_thr) a threshold to control the propagation of saliency changes. Basically, when an object sustains a change of its saliency and this change is (in absolute value) greater than the threshold, then it is propagated, inhibited otherwise.

Propagation applies only to the markers of the object. If some of these allow propagation themselves, then the propagation will be applied to their own markers and also to the objects they may explicitly refer to in their code.

Propagation applies to *all views* of the recipients. When propagated to other objects, saliency changes are adjusted (as mentioned in the sub-section Controllable scope) to the respective thresholds of the groups where the recipients of the change are projected.

The executive detects cycles in the object graph and prevents recursive propagation.

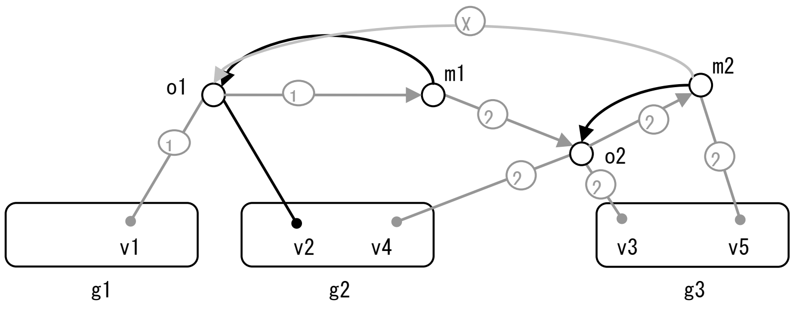

Figure 7 – Propagation of Saliency

The links in grey show the path of saliency propagation. When the saliency of an object o1 *changes in a group* g1*, it is propagated to all the markers referencing* o1 *(here* m1*) – path marked 1.*

If the propagation is enabled in m, then the change is propagated to all markers attached to m1 *(*m2*) and to all objects referenced by* m1 *– path marked 2. Changes are applied to all views of the recipients (*v3*,* v4 *and* v5*), if they have some (*m1 *does not).*

The executive blocks recursive propagation: in this example, m2 *hold a reference to* o1*, but* o1 *is part of the propagation path, and thus, no saliency changes incurred by* m2 *will be applied to* o1 *(path marked X).*

Notifications Groups provide several notifications to reflect the status of the various control values defined for its members. These notifications are (see the definition of groups in Annex 1 for the details):

- changes of saliency, activation and resilience, monitored over a period specified for the group.
- control values reaching peak values (high or low, with respect to some thresholds defined for the group). This applies to saliency, activation; for resilience, only the low peak value is notified[6].
- first time appearence of an object in the group (see sub-section Sub-systems above).
- reductions: each time an instantiated program outputs productions, this is notified (mk.rdx, see Annex 1), and such notification includes references to both the inputs and

[6] The notification of a low resilience warns programs of the possibly imminent deletion of an object.

the outputs of the instantiated program which performed the reduction.

Each group contains a set of references to destination groups where these notifications are to be injected – when a reference is `nil`, the destination is the group itself.

Groups also expose "statistics", i.e. the highest/average/lowest saliency/activation values among its members. These are computed at the end of each update period, and are accessible as members of the group class.

6 High-Level Reasoning

Replicode provides support for non axiomatic logical reasoning. In Replicode, reasoning means inducing and deducing. As mentioned in the overview, the language defines several classes of objects (facts, goals, predictions) upon the production of which the executive can perform forward and backward chaining using models. This is called high-level reasoning - as opposed to the low-level reductions that have been presented in the sections above – because of (a) the semantics ascribed to the production of models (predictions and goals) and, (b) the non-axiomatic nature of the encoded knowledge (confidence values for facts, predictions and goals, success rates and number of evidences for models).

6.1 Predicates

The classes of predicates are defined as follows[7]:

Facts Facts are objects that points to other objects and indicate two timestamps (defining the time interval where the fact is believed to hold), and a confidence value (in [0,1]) indicating how reliable the fact is; this value is *not* a probability.

Facts come in two classes `fact` and `|fact`, the latter (so called “anti fact”) denotes the absence of a fact. The syntax of a fact is:

```
(fact an-object after before confidence-value)
```

where `after` denotes the time after which the fact holds and `before` denotes the time before which the fact holds: these two timestamps define the interval of validity of the fact. At time `t`, the fact holds if and only if `after<=t<=before`.

Inputs coming from the environment (technically, data produced by the I/O devices) come in the form of facts for which the time interval is reduced to a single point in time (the time the input was produced) and for which the confidence value is one (since the fact represents an observation in the environment).

For matching purposes, the executive defines two parameters – `time_tolerance` (in microseconds) and `float_tolerance` (in [0,1]). Two numerical values or two timestamps are deemed to match if they are within the same tolerance interval, i.e. one of the value plus/minus the tolerance.

Goals Goals are objects either given by the programmer or produced by programs or by the executive as the result of backward chaining. A goal is the specification of a target state, in the form of a reference to a fact, and of an actor, that is the entity that pursues the goal. The syntax is:

```
(goal a-fact an-actor)
```

the confidence value held by `a-fact` is interpreted by the executive as an estimate of the chances of success for the goal, i.e. the likelihood that `a-fact` will actually be observed.

A goal constitutes knowledge the system can reason about and thus is pointed to by a fact. This fact encodes the likelihood of the goal to be valid (that is the interpretation of the confidence value) and the time interval in which the goal holds. These values (confidence and time interval) are distinct from the values found in the fact pointed to by the goal.

The confidence value of a goal is interpreted by the executive as the saliency value of the fact holding the goal. In other terms, the saliency of the fact holding the goal is set by the executive to the confidence value of the goal target. As a consequence, goals with low confience values i.e. low chances of success) will have a low saliency and, depending on the relevant saliency thresholds, may be hidden from programs, models and composite states.

Predictions Predictions are objects produced by the executive as the result of forward chaining. The syntax is:

```
(pred a-fact)
```

where `a-fact` is a pointer to the predicted fact: the confidence value held by this

[7] As any object, predicates have their last member encode the saliency change propagation threshold (`psln_thr`). This member is not indicated in the syntax for clarity. See Annex 1 for the full definition of these objects.

fact is interpreted by the executive as an estimate of the chances of success for the prediction, i.e. the likelihood that a fact will confirm the prediction.

As for goals, a prediction constitutes knowledge the system can reason about and thus is also pointed to by a fact. This fact encodes the likelihood of the prediction to be valid (that is the interpretation of the confidence value) and the time interval in which the prediction itself holds. These values (confidence and time interval) are distinct from the values found in the fact pointed to by the prediction, i.e. the predicted fact.

As for goals, the confidence value of a predicted fact is interpreted by the executive as the saliency value of the fact holding the prediction. In other terms, the saliency of the fact holding the prediction is set by the executive to the confidence value of the predicted fact. As a consequence, predictions with low confidence values (i.e. low chance of confirmation) will have a low saliency and, depending on the relevant saliency thresholds, may be hidden from programs, models and composite states.

Notifications of Success or Failure The executive notifies the success or failure of goals and predictions in the form of objects of class `success`. Notifications are held by facts or anti-facts. In the latter case, what is notified is a failure (i.e. an absence of success). The syntax is:

```
(success a-fact an-input)
```

where `a-fact` is a pointer to a fact that points in turn to either a prediction or a goal; `an-input` is the actual input that matched the goal or prediction (case of a success) or the negation of the fact that should have occurred to secure success (case of failure).

A notification is issued for a prediction when, at the predicted deadline (the `before` time value held by the predicted fact) a fact has occurred during the predicted time interval (defined by the `after` and `before` values held by the predicted fact) and matches the predicted fact (success) or no such fact has happened during the predicted time interval (failure). In the case of a failure the executive asserts the absence of the predicted fact in the form of anti-fact pointing to the same object the predicted fact was referring to.

A notification is issued for a goal when, at the goal deadline (the `before` time value held by the goal target, i.e. the fact the goal refers to) a fact has occurred during the time interval specified by the goal (defined by the `after` and `before` values held by the goal target) and matches the goal target (success) or no such fact happened in the prescribed time interval (failure).

6.2 Composite States

A composite state is a set of patterns of facts that shall happen at the same time. Composite states are intended to group synchronous inputs into one single construct. The executive scans the memory and attempts to match the patterns. When all are matched, the executive produces an instance of the composite state (`icst`). The syntax of a composite state is:

```
(cst [set-of-tpl-arguments] [set-of-patterns] [set-of-forward-guards] [set-of-
backward-guards] [set-of-ouput-groups])
```

The first set contains template parameters (in a similar fashion as in the case of programs): these parameters are variables.

Patterns used in composite states (and models) differ from patterns found in programs in that they do not define any set of guards: they are merely facts that contain variables.

Guards are expressions, as found in programs, that are evaluated by the executive to control chaining. In contrast to programs, Replicode allows assignment operations in guards.

The set of output groups is the set of the groups where the instantiated composite state is to be injected upon successful match.

The syntax of an instantiated composite state is:

```
(icst a-cst [set-of-template-values] [set-of-values])
```

where a-cst is a pointer to a composite state, the set of template values is the set of the actual values assigned to the template arguments and the set of values contain the values assigned to the variables during matching.

As in the case of programs, a composite state defines several patterns that can be matched by any combination of facts, each matching one pattern. The executive catches all these combinations and outputs as many instantiated states as it found combinations. This is technically achieved in the same way as for programs, using overlaid reductions.

When all patterns are matched, the forward guards are evaluated and if all of them succeed, an instantiated state is injected. The confidence of the instantiated state is the lowest of its inputs.

Shall at least one input be a prediction, then the instantiated state will be injected as a prediction itself.

For clarity, let's consider an example:

```
s:(cst |[] []
(fact (mk.val an_object: essence  "cube"  ::) t: t: ::)
(fact (mk.val an_onject: color  "blue" ) t: t: ::)
|[]; no forward guards
|[]; no backward guards
[stdin])
```

> Basically, `s` encodes a state of the environment that asserts the presence of a blue cube, i.e. a state where there is an object whose essence (defined by the programmer in the ontology) is a cube and whose color (also part of the ontology) is blue. Whenever such a state is discovered by the executive, say a particular object `x` matches the state at time `t0`, an instance of `s` is injected in the `stdin` group. This instance should, in principle, be encoded as follows:
>
> ```
> (fact (icst s |[] [t0 x] 1) t0 t0 1); confidence is 1
> ```
>
> Notice that the actual values (`t0` and `x`) appear in the value set in the same order as the corresponding variables (`an_object` and `t`) appear in the composite state.
>
> Notice also that `t0` appears redundantly (it is both in the instantiated state and in the fact): the language eliminates this unnecessary redundancy and, instead of the fact mentioned above, actually injects:
>
> ```
> (fact (icst s |[] [x] 1) t0 t0 1)
> ```
>
> This example does not make use of guards: these will be exemplified in the next section.

Composite states can be composed of other states: one would simply define patterns of instantiated states instead of patterns of raw inputs (like the markers in the example given above).

Commands cannot be part of a composite state (they can only appear in models) – see next section.

A composite state can also produce sub-goals, upon catching a goal targeting an instance of the state. In case there exist no model able to produce the targetted instance, the executive produces as many sub-goals as there are patterns in the state, provided all the backwards evaluate successfully. Otherwise, it is the duty of the models to induce their own sub-goal from the targeted instance. If the targeted instance is held by an anti-fact and there is no model able to produce said instance, then the sub-goals are produced as above, excepted that all facts are inverted (facts tunr into anti-facts and reciprocally).

6.3 Models

Knowledge is essentially encoded as models. It is these models that operationally encode the semantics of the phenomena observed in the environment. Models deduce and induce. In Replicode, deduction is the production of predictions given an input fact (actual or predicted), whereas abduction refers to the production of sub-goals, given an input goal.

As mentioned in the introduction, models are either given by the programmer or learned by the system, i.e. acquired from observation and revised. Replicode does not define any model acquisition sub-system: it is expected to be provided by the programmer as an extension of the executive. However, Replicode provides the infrastructure for revising models: the executive monitors the execution of models and updates their success rates accordingly, and these success rates are used in turn to control the execution of models (they are interpreted as activation values, as the ones defined to control programs). The success rate of a model is the number of positive evidences divided by the total number of evidences. By positive evidences we mean the number of times a model has predicted well (e.g. the predicted fact actually came true); the total number of evidences is the number of times the model produced a prediction, regardless of its outcome (success or failure).

A model is essentially a pair of patterns and two sets of guards (as for composite states), and encodes a production rule of the form A → B where A and B are patterns of facts as defined for composite states. Thus a model has only one input and one output: during forward chaining the input is the object matching A and the output is the object produced as an instance of B, whereas during backward chaining, the input is either (a) the goal matching B and the output is a sub-goal produced as an instance of A or, (b) an instance of B and the output is an *assumption*, i.e. an instance of A whose confidence value is calculated exactly as for goals (see below). Using instantiated states in the patterns of a model allows said model to operating on (or producing) conjunctions of facts.

During backward chaining, if a goal matches the right-side pattern with an inverted target fact, then the sub-goal target is also inverted.

When the left-side pattern of a model is a command, sub-goals built from this pattern are sent to the I/O devices. The command sent shall be considered a *request* to the device rather than an order. Devices have internal states that may be hidden from the system (the system would not have models of some devices) and thus, the system shall be informed of the command that was *actually* executed (if any) or of exceptions that were raised during the attempt to execute the command. Either the devices themselves or user-defined programs shall a copy of the command they actually executed back in the memory or, in case of failure, an exception. Commands can only appear as the left-side pattern of a model. Goals on commands that contain unbound values are not passed to the I/O devices: they are just lost.

Forward and backward chaining are performed concurrently, i.e. at any time a model may produce a sub-goal and a prediction simultaneously.

The confidence values for goals and predictions are computed as follows:

- The confidence value of a sub-goal is the product of the success rate of the model and the confidence value of the input goal
- The confidence value of a prediction is the product of the success rate of the model and the confidence value of the input fact or of the predicted fact if the input is a prediction.

Accordingly, the more models involved in a reduction (abduction or deduction) path, the lesser the confidence of the final product (goal or prediction), assuming some models are not perfect (e.g. their success rates are below 1). By reduction path we mean the set of models that, from an initial input (goal or prediction), output objects that constitute inputs for other models in the set.

The syntax of a model is:

```
(mdl [set-of-tpl-arguments] [set-of-patterns] [set-of-forward-guards] [set-of-
backward-guards] [set-of-ouput-groups] strength number-of-evidences success-
rate derivative-of-the-success-rate)
```

The five first members are defined exactly as for composite states. The strength of a model is a number in {0,1} that indicates if the model has performed well, consistently in the past. The so-called derivative of the success rate is actually the success rate at the time of the former to last evidence. This value is provided to control learning as it indicates the progression of the success rate.

When a model produces an output, the executive notifies this operation, using a reduction marker (mk.`rdx`), (as in the case of programs) where the `code` member is set to the instantiated model that produced the goal or prediction. The syntax of an instantiated model is:

```
(imdl a-mdl [set-of-template-values] [set-of-values])
```

As for instantiated composite states, the values appear in the instantiated model in the same order as their corresponding variables in the model. More over, the time redundancy is also eliminated, as for composite states - in the case of models, the time in question is the time of the matching of the left-side pattern.

Models can encode knowledge about other models, in other words, models can reuse and control other models.

Reuse Reusing a model M0 in a model M1 means having M1 predict a fact when M0 outputs its own prediction. The left-side pattern of M1 would then be encoded as a pattern on an instance of M0 (imdl), in other words, M1 can be sketched as: imdl M0 → B (imdl stands for a fact pointing to an imdl).

During forward chaining, M1 reuses the prediction produced by M0 to trigger its own predictions, in effect, adding its own prediction to M0's.

During backward chaining, M1 needs M0 to achieve its goal: assuming M0 is A → C, when M1 receives a goal b matching B, then the executive produces a sub-goal c matching C in M0.

Control To control M0 by M1 means that M1 allows or blocks the execution of M0 (for both chaining modes). M1 can be sketched as: A → imdl M0, and another model M3 as: B → |imdl M0 (|imdl stands for an anti-fact pointing to an imdl). M1 reads "when some a matches A, then I predict that if M0 predicts something, it will succeed" whereas M2 reads: "when some b matches B, then I predict that if M0 predicts something, it will fail". M2 is called a strong requirement (because without M2, M0 will fail) and M1 is called a weak requirement (because without M1, M0 may still succeed).

During forward chaining, M0 collects all predictions about its own success, from both its strong and weak requirements. When M0 matches its input, it will output a prediction only if (a) there is no requirements at all or, (b) there is only weak requirements and at least one has predicted the success of M0 or, (c) there are only strong requirements and none of them has predicted anything about M0 or, (d) there are both kinds of requirements, both having predicted, and one prediction of success of M0 has higher a confidence than any other. Otherwise, chaining is not allowed and M0 will not predict anything. in case chaining is not allowed, the executive runs M0 silently, i.e. it makes the prediction M0 would have made without injecting it: this allows the executive monitoring the potential outcome of running M0 and thus check if the predictions of the strong requirements were correct or not.

During backward chaining, M0 also collects all predictions about its own success as above (actually, such predictions are collected continuously and discarded when the current time leaves their time interval). When M0 matches its right-side pattern, the executive checks if chaining is allowed, using the same rules as for forward chaining. If chaining is allowed, then the backward guards are evaluated *using the values stored in the instantiated model that allowed chaining*. This instantiated model (of M0) is the right-side term of a model that predicted the success of M0 with greater confidence. Once the backward guards have been successfully evaluated, M0 produces a sub-goal, as if there were no requirements.

Requirements are counted by the executive among the active models: when a model loses activation, and if it was a requirement for some other model, then the executive will count one less requirement for the latter model. Symmetrically, when a model being a requirement for another one gains activation, then the executive will count one more requirement for the latter model.

To illustrate the use of models, let's consider the example of a simple motor control. Let's assume a command lift_hand taking two arguments, the hand and the distance the device shall move. The following set of models and composite states encodes the consequences of lifting a hand – or, from the goal generation point of view,, encode a command generator for lifting a hand:

```
M0:(mdl [p0:] []
   (fact (cmd lift_hand [a_hand: a_distance:] :) t0: t0: ::)
   (fact (mk.val a_hand position p1: ::) t1: t1: ::)
[]; forward guards
   t1:(+ t0 SAMPLING_PERIOD)
   p1:(+ p0 a_distance)
 []; backward guards
   a_distance:(- p1 p0)
   t0:(- t1 SAMPLING_PERIOD)
[stdin])
```

where `SAMPLING_PERIOD` is a user-defined constant. Notice the assignment operations in the forward guard, for example, during forward chaining, `t1` will be set to `t0+SAMPLING_PERIOD`.

```
S:(cst |[] []
(fact (mk.val an_object: essence "hand" ::) t: t: ::)
(fact (mk.val an_onject: position p0:) t: t: ::)
|[]; forward guards
|[]; backward guards
[stdin])

M1:(mdl []
   (fact (icst S |[] [a_hand: p0:] ::) t2: t2: ::)
   (fact (imdl M0 [p0] [a_hand: a_distance: t1: p1:] ::) t0: t0: ::)
[]
   t0:(+ t2 SOME_CONSTANT)
|[]
[stdin])
```

where `SOME_CONSTANT` is user-supplied.

Forward chaining: at T0, say the right hand is at a position P0, S is instantiated (values: the_right_hand and P0); M1 predicts the success of M0 with the template value P0 and the values the_right_hand and a_distance: (the latter unbound). In turn, when M0 matches a command, chaining is allowed (thanks to the prediction from M1), then M0 evaluates its forward guards and predicts the next position of the hand.

Backward chaining: let's assume a goal G0 trying to get the right hand at a position P1 at time T1. The right-side pattern of M0 matches G0, but as M0 is controlled by some requirements (here, M1), the system must ensure that there is a prediction of success for M0. This produces a sub-goal G1, targetting an instance of S with the values the_right_hand and an unbound value for p0. Assume now that the right hand is at a position P0; S will be instantiated with the values the_right_hand and P0; M1 will predict the success of M0, with P0 as the template value, and the values the_right_hand and undefined values for a_distance and p1. At this point, chaining is allowed for M0: the backward guards are evaluated, which produces another sub-goal G2, actually an instance of the command, with the parameters the_right_hand and P1-P0.

Each group a model live in must be paired with another group and is actually active in either one or the other. The first group is called the primary group and it is where a model runs according to the rules mentioned above. Shall a model loose activation in the primary group, then it is activated in the secondary group. The reason for this is as follows: a model loses activation when its success rate goes below the activation threshold of the primary group, and thus will never execute again, therefore there would be no chance for the model to getter better a success rate. In the secondary group, no abduction is performed and only successes are monitored; also, predictions are not injected in the secondary group (they are still silently produced for model rating purposes). When a model in the secondary group gets its activation value above the activation threshold of the primary group, it is deactivated in the secondary group and gets back to the regular reduction mode allowed in the primary group. Shall a model in the secondary group loose activation in that very group, it is deleted permanently, excepted for strong models which live forever: strong models starting to misbehave indicate a context change, that is, the environment has entered a state where what was predictable by known methods is not so anymore, but this does not mean the models are bad – they are just out of the context where they use to make sense.

Replicode defines a construct for pairing groups; its syntax is:

```
(mk.grp_pair primary-group secondary-group)
```

Notice that such a marker shall *not* be injected in any of the two groups to avoid circular dependencies[8].

[8] group -> view -> marker -> group, where -> denotes a reference counting smart pointer.

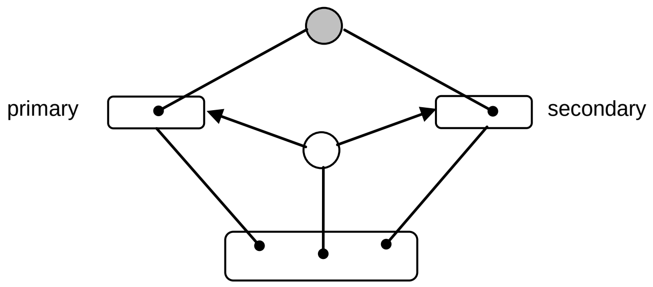

Figure 8 – Paired Groups Hosting a Model

A model (grey circle) is projected on two groups: these are paired using a `mk.grp_pair` *object (white circle). A model can live in an arbitrary number of such paired groups.*

Composite states live only in the primary group.

Goals pursued by the system are injected by models and states only in the primary group. Predictions are injected in the output groups defined for states and models.

The method for calculating the success rates brings some undesirable inertia to the activation of models. Consider for example the case of a model having succeeded for a long time: it will take a large amount of failures to bring its success rate below the activation threshold of the primary group. In other words, good models that start performing badly remain active for too long, and then keep injecting wrong predictions unnecessarily. The executive take two control parameters - `mdl_inertia_sr_thr` and `mdl_inertia_cnt_thr`: the first parameter defines a threshold on the success rate and the second, a threshold on the number of evidences. When a model reaches both thresholds (good success rate over a large number of evidences) then the model is marked as being strong and the number of evidences is trimmed down to 1 if the number of evidence was exactly the same as the threshold, or is reduced by the value of the threshold otherwise. In addition, the success rate of such a model is reset to 1.

6.4 Pattern Extraction

This section describes how models and states are induced from input data. Acquiring new knowledge contributes to learning alongside model revision (the latter has been described earlier: it is the maintenance of the success rates of the models and possibly their deletion).

Replicode offers three ways of acquiring models/states, by means of three different components, called Targeted Pattern Extractors (TPX),that attempt to model causal relationships between inputs and their target. Targets are different with regards to the type of TPX:

- GTPX, Goal Targeted Pattern Extractor: a GTPX is instantiated each time a goal is produced, and the target is the success of said goal;
- PTPX, Prediction Targeted Pattern Extractor: a PTPX is instantiated each time a prediction is produced and the target is the failure of said prediction;
- CTPX, Change Targeted Pattern Extractor: a CTPX is instantiated for each known state and the target is a change of said state.

TPX operate on time-bounded buffers of inputs – a system parameter defines a time horizon (TPX_THZ) for limiting the size of the buffers (inputs older than now-TPX_THZ are discarded).Upon observing it's a target, a TPX trims its buffer down then proceeds (if possible) to build new models and states. We now describe these procedures in details for each kind of TPX.

GTPX:

Trimming down the buffer:

1 if one input is a trace of the execution of one model that solved the goal, abort – a trace of model execution is called a reduction marker and contains (a) a reference to

the executed model, (b) a reference to its input and (c) a reference to its production. This step ensures that no two identical models can be produced;

2 remove any inputs that triggered any model execution – model execution is traced solely by imdl constructs;

3 remove any inputs that were assembled in composite states – only high-level state instances will be traced in the form of icst constructs.

N.B.: the input buffer is composed of inputs satisfying the time requirements and sharing at least one value with the goal.

Induction:

1 starting from the younger input, if a plausible cause C is found, scan for an icst containing C; if none, scan for synchronous inputs and build a new composite state. Remove the inputs from which the new state was build.

2 use C (no state found) or the identified icst as the left side of a new model, the goal as the right side. Build guards.

3 goto 1 until the buffer is empty.

PTPX:
Trimming down the buffer:

1 if one input is a trace of the execution of one model that predicted the failure of the prediction, abort – a trace of model execution is called a reduction marker and contains (a) a reference to the executed model, (b) a reference to its input and (c) a reference to its production. This step ensures that no two identical models can be produced;

2 remove any inputs that triggered any model execution – model execution is traced solely by imdl constructs;

3 remove any inputs that were assembled in composite states – only high-level state instances will be traced in the form of icst constructs.

N.B.: the input buffer is composed of inputs satisfying the time requirements and sharing at least one value with the prediction.

Induction:

1 starting from the younger input, if a plausible cause C is found, scan for an icst containing C; if none, scan for synchronous inputs and build a new composite state. Remove the inputs from which the new state was build.

2 use C (no state found) or the identified icst as the left side of a new model, the instance of the model that failed to predict as the right side. Build guards. Notice that the new model is a strong requirement on the model that failed to predict correctly.

3 goto 1 until the buffer is empty.

CTPX:
Triiming down the buffer:

1 if one input is the trace of the execution of a model that predicted the change, abort.
2 if the state does not change, abort. A state is deemed not to change when one input matches it at time t_0+SAMPLING_PERIOD, where t_0 is the time of the last injection of the state.

N.B.: the input buffer of a CTPX is the union of the buffers of all the currently active GTPX and PTPX.

Induction:

1 starting from the younger input, if a plausible cause C is found, scan for iS containing C; if none, scan for synchronous inputs and build a composite state. Remove the

inputs from which the new state was build.
2 use C (no state found) or the identified icst as the left side of a new model, the new state as the right side. Build guards.
3 build a weak requirement on the new model, left side: the state that changed. Notice that this induction procedure actually buils two new models,the first capturing the cause of the new value of the state, the second being a weak requirement on the first. These two models actually encode a state transition.
4 goto 1 until the buffer is empty.

To control model production Replicode maintains a model base, composed of two lists:

- Black List: models that failed are stored in the black list: TPX cannot inject new models that are already in the list. This prevents producing models that have proven to be inadequate.
- White List: strong models (good success rate, numerous evidence count) are stored in the white list: TPX cannot inject new models that are already in the list. This prevents producing new models identical to exisitng ones that are not active but still good performers.

6.5 Simulation

Deriving a sub-goal from a goal (abduction) can have the following undesirable consequences: (a) the sub-goal can, in fine , have impacts on the environment that would cause a failure of the initial goal (the goal would then be self-defeating) or, (b) the impacts of the sub-goal on the environment would cause another goal to fail.

Moreover, as discussed in the previous section, weak requirement are actually alternatives for reaching the goal of the model they control and the system shall select the most appropriate alternative instead of pursuing them all.

For all of these reasons, the executive simulates the impacts of sub-goals before committing to them. Simulation is a process hidden to the system itself, i.e. the system has no knowledge about what is being simulated and about the results. It is a strictly internal process of the executive.

Simulation is a process that performs for a limited time before committing to sub-goals. the executive defines three parameters – `min_sim_time_horizon`, `max_sim_time_horizon` and `sim_time_horizon` – that control the time allowance for simulation. The simulation time horizon is computed as a percentage of: the goal deadline minus the current time, clamped by the min and max values. In a nutshell, simulation is performed as a time-bound succession of abduction steps followed by simulated deductions:

Abduction Step 1 For a given goal the system already committed to (thereafter called an actual goal as opposed to a simulated goal), several models induce sub-goals as described in the previous section. Each of these sub-goals is tagged (internally) with a simulation object that refers to the actual goal.

Abduction Step 2 For each simulated goal, models induce more sub-goals: these are tagged (internally) with a solution object that refers to the simulation objects produced in the previous step.

Abduction Step 3 For each solution goal, models induce furthermore and the resulting sub-goals are tagged with the same solution objects produced in the previous step.

Abduction Step 4 + Each subsequent step performs as step 3.

Abduction steps end at `sim_time_horizon/2`. From then on, simulated predictions of the deepest sub-goal targets are injected (with a confidence value of 1). The deepest goals are the goals for which further abduction steps were prevented. The simulated deduction is performed as for regular predictions, using forward chaining and stops at `sim_time_horizon`, counting from the beginning of the simulation.

After `sim_time_horizon`, the system stops the simulation and makes a decision about which sub-goals to actually pursue, based on the outcome of the simulated predictions.

Notice that simulated goals and predictions are injected *only in the primary group*.

The rest of this section describes in details how the simulation is performed, and for clarity we will examine three cases: (a) a set of models containing no commands, with no requirements, (b) the same set where a model contains a command and, (c) the same set (no commands) where a model is controlled by some requirements.

Case 1 – no command, no requirement

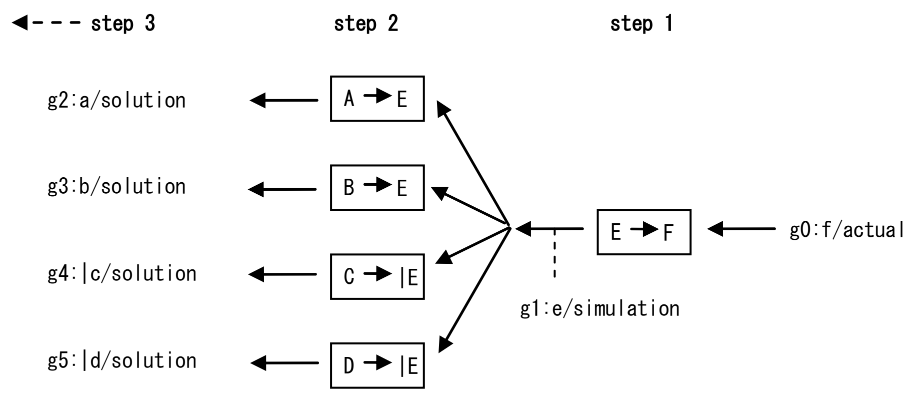

Figure 9 – Abduction Steps (no command, no requirement)

Key: models are represented by boxes containing patterns, noted in capital letters. Goals are represented as gi: target/type, where a target matching a pattern X is written x.

See text for details.

As mentioned in the simulation overview, the sub-goals produced from g2 to g5 will be marked by the exact same solution marks as for g2 to g5.

When the abduction steps stop, predictions are simulated, i.e. they are marked with the solution markers of the goals that could not be simulated (the deepest goals).

each actual goal is monitored by the executive and, at the end of the simulation, a decision is made depending on the simulated predictions. The predictions of interest are the predictions of success or failure of *any* goal, simulated or not; for g0, only the predictions holding a solution mark that refers to the simulation mark that in turn refer to g0 will be considered.

The objective of the decision is to choose the sub-goals to pursue in order to achieve g0. In the case depicted above success for g0 requires pursuing *both* g4 and g5, and *the best* of g2 and g3, if no conflicts are anticipated. Notice that the system will *not* commit to g1, but to g1's sub-goals instead.

Conflicts are identified by simulated predictions of the failure of another goal; decision is performed as follows:

- if a simulation causes the failure of an actual goal, the simulation and its solutions are discarded;
- if a simulation causes the failure of a simulated goal, say g, the simulation is discarded if g is more important than g0, kept otherwise. The importance of a goal is a value, computed internally, defined as the product of the confidence of the goal, a unction of the time left between the current time and the "before" deadline of the goal and the highest confidence of the predictions that warned about the success or failure of the goal.
- if after the evaluation of conflicts, there are still some simulations, then the executive commits to all the solutions that are required (like g4 and g5 in the picture above) and to the best of the others (e.g. g2 or g3), chosen according to their importance value.
- if there is no simulation left, then the executive will start another simulation.

Case 2 – one command, no requirement

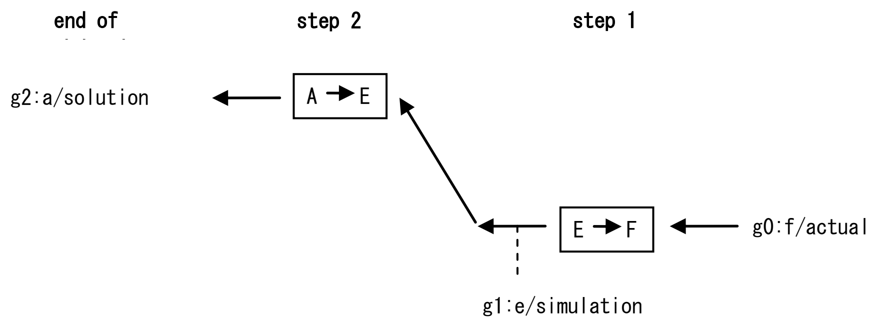

Figure 10 – Abduction Path (one command, no requirement)

This case is similar to the one presented in figure 9 above, except that A is now the pattern of a command (and also, other solutions have been omitted for clarity).

See text for details.

Inducing from a command goal cannot yield sub-goals, as commands constitute the interface between the system and its devices (and through these, to the environment). Consequently, when a goal matches the right-side pattern of a model containing a command, the simulation stops as if half the time allowance would have spent.

At the end of the simulation, assuming the simulation g1 is chosen, the system commits to the command, and said command is sent to the devices.

Shall the command be patterned after E instead of A, then upon matching of g0, the simulation returns immediately and the system will commit to g2 – assuming again a positive conflict resolution.

Case 3 – no command, some requirements

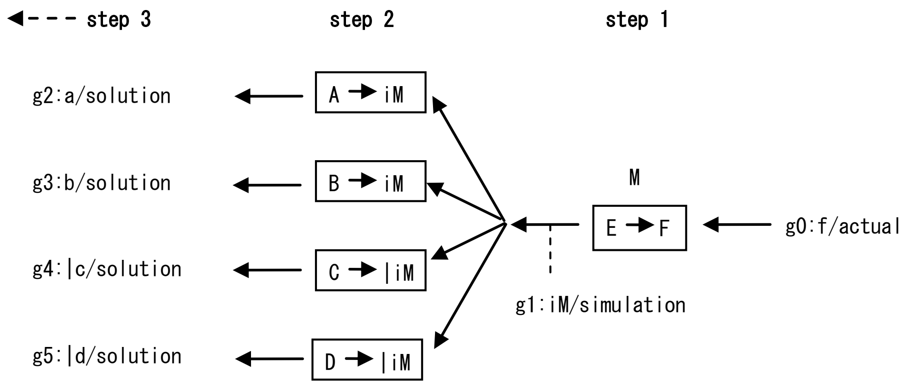

Figure 11 – Abduction Path (no command, some requirements)

Key: iM denotes (fact (imdl M arguments) …), |imdl stands for (|fact (imdl M arguments) …).

See text for details.

If for model M, there are some requirements and the chaining is allowed, then the executive proceeds as in Case 1. Otherwise, it will simulate a goal for allowing chaining, i.e. g1, and will perform the simulation as in Case 1. In case of a positive conflict resolution, the executive will commit to both g4 and g5, and to the best of g1 and g2. At the next simulation period, the executive will induce again from g0 until chaining is allowed, or g0 fails. If at some point chaining is allowed, then the executive will perform as in Case 1.

There is a particular case that is handled specifically: assume g0 is actually g☹f instead of g:f. Then if chaining is not allowed, then there is no chances that M will execute and thus will not produce f, which is the goal. In that case, the executive will not simulate g1 at all, and shall g0 be a simulation or a solution itself, then the abduction would stop (as if the time allowance was spent).

General Rules

These rules apply at the end of each simulation time, for a given goal:

- in case conflicting predictions have been received during the simulation time, the one with the highest confidence is retained, the others discarded;
- when an actual goal fails, all its sub-goals and simulations are terminated, then a new simulation starts;
- for an actual goal that has already committed to sub-goals, nothing particular is performed if no prediction of failure have been received during the simulation time. Otherwise, the sub-goals and their simulations are terminated and a new simulation starts;
- when, for an actual goal that has not committed to sub-goals yet (i.e. these are being simulated), a prediction of success is received, then the simulations are terminated; the rationale is that for such a goal, success can be predicted without having to commit to anything;
- for an actual goal that has not committed yet, if no simulation can be chosen (either none have been received or the one received have been discarded), a new one is started;

6.6 Drives

Drives are goals whose target is not observable. Drives constitute the very core of a system – they can be considered “innate” goals. A system shall be designed to learn how to solve its own drives, most typically, with the help of user-defined models that give it some headways.

Each drives is to be injected periodically in a system by a set of programs reacting upon the success or failure of the previous issue of the drive (let’s call these programs “drive injectors”). To allow the control of the activation of these drives, drives injectors are isolated in one specific group, defined by the application. Models that take a drive as their right-hand side are called top-level models and must specify the drives group as the last parameter in their output group set.

Since drives are not observable, top-level models do not issue predictions: instead they inject the outcome of their left-hand side goal (sub-goals of their drive) into the drives group, thus allowing drive injectors to perform (for example, re-inject the drive after a failure, or delay its re-injection after a success).

There can exist only one top-level model per drive, and no requirements (weak or strong) are allowed to control top-level models.

7 Self-Monitoring of Time Resources

The executive periodically assesses its available time resources (i.e. its time-wise execution performance). A parameter – `perf_sampling_period` - defines the sampling period of the monitoring upon which performance objects are injected in the memory. These objects are of the class `perf` (see Annex 1 for details) and carry four numbers:

- `rj_ltcy`: reduction jobs latency;
- `d_rj_ltcy`: reduction job latency at the last sampling time;
- `tj_ltcy`: time job latency;
- `d_tj_ltcy`: time job latency at the last sampling time.

All the tasks performed by the executive fall into two categories: immediate reduction jobs and time-delayed reduction jobs (i.e. timer-locked processing). Each kind of jobs is injected into a dedicated pipeline and picked up by two separate pools of threads. The measurements carried by the `perf` objects mean:

- reduction job latency: the delay between the injection time of a job in the pipeline and the time it was processed by a thread, averaged over the sampling period: the lower the better.
- time job latency: the delay between the time the thread that picked up the job had to wait, averaged over the sampling period: the higher the better.

The courses of action that can be taken to bring the system back within an acceptable operational envelope are:

- raise the primary/secondary group activation/saliency thresholds: fewer models will be executed, goals and predictions will flow for a reduced range (as their saliency will get under the threshold more rapidly).
- reduce the simulation time allowance;
- raise the activation threshold in the drives group.

8 Distributed Computation

This section describes how computation is distributed over a cluster of computing nodes.

Inter-node communication is *not a transparent operation* carried out by the executive. Instead, programs have to explicitly transmit code to remote nodes. This is the role of the ejection function (`eje`) defined by the executive: it admits four arguments, (a) the object to be ejected, (b) a view (see below), (c) the destination node and (d) a selector indicating the group in the destination node where the object shall be projected (see below).

In addition to `root`, Replicode defines two other default groups to facilitate communications:

- `stdin`: the standard input group. When some program on a node ejects an object to another node, the object can be projected on the stdin of the destination node, with the view passed as the second argument to `eje`.
- `stdout`: the standard output group. If code on a node `n1` needs to detect patterns of events occurring on another node `n2`, then the way to proceed is to send to `n2` some programs able to perform such detection and when successful, to send the productions to `n1`. Such deported detectors are actually proxies of programs from `n2` sent to `n1`. The standard output is the group where proxies shall be ejected to.

The last argument of the `eje` function selects whether the ejected object shall land on `stdin` or on `stdout` in the destination node. For communication purposes, `stdin` and `stdout` are the only groups that are guaranteed to exist at any time (`root` is private to a node).

When an object is received by a node, its view contains an injection time representing the reception time on the node.

When an object is ejected to a remote node, *neither* its markers nor its views are transmitted. To be transmitted, markers have to be explicitly ejected. However, if the object contains explicit references to other objects, these will be transmitted. Groups cannot be transmitted.

Objects sent by devices to nodes hosting an instance of the executive always land in `stdin`.

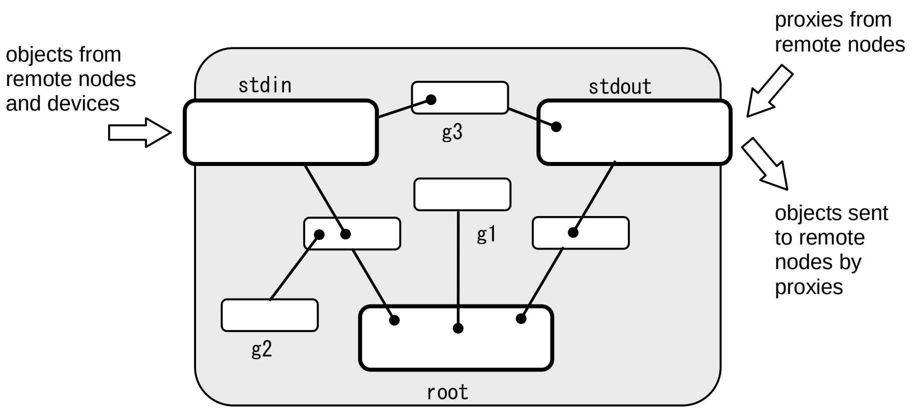
objects from remote nodes and devices
proxies from remote nodes
stdin
stdout
g3
g1
g2
root
objects sent to remote nodes by proxies

Figure 12 – Structure of a Node

The node's three default groups - `root`*,* `stdin` *and* `stdout` *– are always present in memory. Other groups have to be projected (directly like* `g1` *or indirectly, like* `g2`*) onto at least one of the default groups.*

Remote nodes send detectors to `stdout`*, and other objects to* `stdin`*. Devices always send data to* `stdin`*. Once received, objects can be injected freely in any group on the node, by programs in said node.*

Any program in the node can send objects to devices and remote nodes, regardless of its location in the node (it does not have to be projected on any of the default groups to do so).

The objects flow from left (`stdin`*) to right (*`stdout`*) as instrumented by group* `g3` *– it is assumed here that* `stdin` *is visible in* `g3` *and* `g3` *in turn, is visible in* `stdout`*. In* *other words, there must exist a reduction path from* `stdin` *to* `stdout`*.*

9 References

Thórisson, K. R. (2012). A New Constructivist AI: From Manual Construction to Self-Constructive Systems. In P. Wang and B. Goertzel (eds.), Theoretical Foundations of Artificial General Intelligence. Atlantis Thinking Machines, 4:145-171.

Thórisson, K. R. & E. Nivel (2009a). Holistic Intelligence: Transversal Skills and Current Methodologies. Proc. of the Second Conference on Artificial General Intelligence, 220-221. Arlington, VA, USA, March 6-9.

Thórisson, K. R. & Nivel, E. (2009b). Achieving Artificial General Intelligence Through Peewee Granularity. Proc. of the Second Conference on Artificial General Intelligence, 222-223. Arlington, VA, USA, March 6-9.

Nivel, E. & K. R. Thórisson (2009). Self-Programming: Operationalizing Autonomy. Proc. of the Second Conference on Artificial General Intelligence, 150-155. Arlington, VA, USA, March 6-9.

Annex 1 – Replicode Specification

1 Overview

This annex defines the syntax and semantics of Replicode language constructs. Annex 2 specifies the semantics of the executive.

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.rdx a [b c d] [e f] 1)`, an expression denoting an object of class `mk.rdx` followed by one object `a`, a set of three objects `b`, `c`, `d`, a set of two objects `e`, `f`, and a value, 1. (meaning the instantiated program a has reduced the objects `b`, `c` and `d` to produce the new objects `e` and `f`).
- **Directives**: instructions to specify how the compiler shall handle code. Directives are not subject to rewriting, they are commands defined by the pre-processor, interpreted at compilation 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 “undefined expression”.
`[]`	opcode of sets.
`|[]`	constant identifying an empty set. N.B.: `[ ]` is illegal, and `[nil]` is a set containing one element.
`|nb`	constant meaning “undefined number”.
`|us`	constant meaning “undefined time”.
`|did`	constant meaning “undefined device identifier”.
`|fid`	constant meaning “undefined device function identifier”.
`|nid`	constant meaning “undefined node identifier”.

`|bl`	constant meaning “undefined Boolean value”.
|st	constant meaning “undefined character string”.
`forever`	time constant (infinite time).
`true`	Boolean constant.
`false`	Boolean 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 program’s code. *a-name* is an alphanumerical identifier.
a-name#a-class-name:	declaration of a variable inside a program’s code with the indication of a cast to the specified class. This allows referencing members of the structure that valuates the variable, according to the definition of the class into which the variable is cast.
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 31 bits (see Annex 3 for details); also used to represent integers (lesser than or equal to 2^{24}). Whenever a class member or an operator argument is expected to be an integral value and is valuated by a non-integral value, said value is converted to the nearest lower integral value.
a-floating-point-number|*a-floating-point-number*	an imprecise number. The left-hand number is the center of an interval, the right-hand number represents half the size of the interval: a|b defines the interval [a-b, a+b].
a-timestamp	an unsigned integer on 64 bits (in microseconds since 01/01/1970), post-fixed with “us”.
a-timestamp|*a-timestamp*	an imprecise time value, post-fixed with “us”. a|bus defines the time interval [a-b, a+b], in microseconds.
“*a-character-string*”	character string (8 bits per character).
`this`	reference to an instantiated program in the code of the program (similar to C++).
`this.`*a-member*	reference to a value identified by the instantiated programs’s member *a-member* in the code of the program 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. Daisy chaining of member access is permitted (e.g. *a-name*`.`*a-member*`.`*a-member*).
a-member	an alphanumerical identifier indexing a member in an object’s structure.
opcode	an operator or an object/marker class.
identifier	an integral unsigned value identifying a device or a device function. Such values are typically defined as macros mapping symbols into r-code constructs.
index	an integral unsigned value identifying an index in a set viewed as an array. This is meant to access members in sets like sets of template arguments which are actually arrays (see the `at` operator section 4.1 below).

Replicode is case-sensitive.

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

Atoms of types named variable, label and member 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 functions of the executive and label/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 opcode 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 limit 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 3 spaces.

Examples:

```
(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*	
	{list-of-member-declarations}	Only allowed to instantiate a variable in a template class.
	`:~`	A class featuring members declared as ~ 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.
	`us`	Timestamp.
	`bl`	Boolean value.
	`sid`	Group 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	`::`*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*`)`	Anonymous instantiation. Expands into a list of members where the variables are replaced piece-wise by the member declarations passed as actual arguments. No structure is created and

		the opcode of the class template is therefore ignored. See section 4.2.1 of this Annex for details.
	`member-name:`(*class-template list-of-member-declarations*)	Named instantiation. In contrast with the method above, this creates a sub-structure led by the opcode of the class template; member substitution is performed as in the case above. See section 4.2.1 of this Annex for details.
Operator directive	`!op` (*operator-name list-of-anonymous-member-declarations*)`:`*type*	Exposes the signature of an operator, i.e. the type of its 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.

The type information associated with class members allow instantiating structures when these members are assigned undefined values (see Annex 1, section 4.2.2). Return types are also specified for operators for the sake of consistency with class member declarations; however, they are not exploited at runtime in this current specification: *no type checking is to be performed at runtime*. Nevertheless operator return types are used by the compiler to enforce minimal static type checking (see Annex 3, section 2.1).

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: code and a set of views.

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 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 arguments.
`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.
`red`		scans a set for objects matching some patterns, returns a set of productions.
rnd		returns a pseudo-random number.
`fvw`		finds a view on an object in a group.

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*
`view`	view on an object. All views have the same core constituents (saliency, resilience, etc.); in addition to these some views define some specific parameters (for instantiated programs, models and group).

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.
|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.
pgm	program.
|pgm	anti-program. Behaves as a program, but reacts when no objects match the pattern within a period of time.
ipgm	instantiated program.
icpp_pgm	instantiated C++ program.
grp	group.
fact	fact.
|fact	negation (or absence) of a fact.
mdl	model.
cst	composite state.
imdl	instantiated model.
icst	instantiated composite state.
pred	prediction.
goal	goal.
success	notification of the succes of a prediction or a goal.
icmd	internal command.
cmd	command to a device, i.e. invocation of a device function.
dev	device.
nod	computing node.
perf	performance counter.
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 an object in a group.
	eje	ejects an object into a remote node.
	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.
	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).
	stop	terminates the application.

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

3.4 Marker Classes

Opcode	*Description*
`mk.rdx`	notification of a reduction.
`mk.grp_pair`	indicates that two groups are paired for hosting models.
`mk.low_sln`	notification of an object getting low saliency.
`mk.high_sln`	notification of an object getting high saliency.
`mk.low_act`	notification of an object getting low activation.
`mk.high_act`	notification of an object getting high activation
`mk.low_res`	notification of an object getting low resilience.
`mk.sln_chg`	notification of an object's saliency change.
`mk.act_chg`	notification of an object's activation change.
`mk.new`	notification of the injection or visibility of an object for the first time.

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 Groups

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

Group	*Description*
`root`	the standard root group.
`stdin`	the standard input group.
stdout	the standard output group.

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 formal overloads are not part of the language, the signatures of such overloads are given when appropriate for readability.

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

_now

Signature

`!op (_now):us`

Description

Returns the current time.

Return value

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

equ

Signature

`!op (= : :):bl`

Description

Checks for the equality of the two arguments. When the arguments are objects, `equ` compares the content of said objects (it ignores views and markers).

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. When the arguments are objects, `neq` compares the content of said objects (it ignores views and markers).

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 (+ : :):`

Description

Adds the two arguments.

Return value

Depending on the class of the arguments, a value representing the sum of the two arguments.

sub

Signature

`!op (- : :):`

Description

Subtract the second argument from the first.

Return value

Depending on the class of the arguments, a value representing the difference between the first argument and the second.

mul

Signature

`!op (* : :):`

Description

Multiplies the two arguments.

Return value

Depending on the class of the arguments, a value representing the product of the two arguments.

div

Signature

`!op (/ : :):`

Description

Divides the first argument by the second.

Return value

Depending on the class of the arguments, a value 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 (`ipgm`, `ifun`). The set of arguments in the instantiated object is filled with the actual values of the template arguments. In case at least one of the actual arguments does not match its corresponding pattern, `ins` returns `nil`, and thus the instantiation fails.

`red`

Signature

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

positive-section is a set containing (a) a set of patterns and (b) a set of productions. When an element of the input set matches *at least one* of the patterns in the positive section, all the productions specified in the section are computed for this element and added to the result set. A production is an expression that possibly refers to variables expressed in the patterns.

negative-section is a set containing (a) a set of patterns and (b) a set of productions. When an element of the input set matches *at least one* of the patterns in the negative section *but fail to match at least one pattern in the positive section*, all the productions specified in the section are computed for this element and added to the result set. A production is an expression that possibly refers to variables expressed in the patterns.

Description

Fills a set with productions computed as functions of the elements in the input set that match at least one of the patterns in the positive section, and productions computed as functions of the elements that do not match any patterns in the positive section but match at least one the patterns in the negative section.

Patterns can as general as may be, for example: `(ptn o: |[])` is valid in both sections.

Return value

A set whose elements result from the transformation of the elements in the input set by the operations specified in the positive and negative section.

Shall the resulting contain commands, these will not be executed. To do so, the resulting set shall be referred to in the production section of a program.

The resulting set may contain duplicates, depending on the patterns and associated productions. Such duplicates are *not* eliminated.

rnd

Signature

`!op (rnd :nb):nb`

Description

Computes a pseudo-random number in a given range.

Return value

A pseudo-random number within the specified range.

fvw

Signature

!op (fwv *obj*: *grp*:grp):

Description

Retrieves the view of the object *obj* in the group *grp*.

Return value

A view. If the object has not been projected on the specified group, `nil` is returned.

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:

```
(ptn (ipgm code: ::) |[])
...
(... code ...)
```

can be written:

```
(ptn an_ipgm:(ipgm ::) |[])
...
(... an_ipgm.code ...)
```

where `code` ia a member of the class `ipgm`.

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 process, and there is no need to specify a pattern for the members of a process.

Member identifiers are alphanumerical and represented internally as 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 exists a notation for defining members of 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 m7:c1) m5:)`
>
> then `c3` embeds an instantiation of `c2` with `c1`. `(c2 m7:c1)` is translated into:
>
> `m3 m7 m4` where m7 has the structure: `(`*opcode-of-c1* `m1 m2)`.
>
> The structure of an object of class `c3` is:
>
> `(`*opcode-of-c3* `m3 m7 m4 m5); m3=1 m7=2 m4=3 m5=4`
>
> Shall we have `c3` defined as `!class (c3 m8:(c2 m7:c1) m5:)` then c3 would have the following structure:
>
> `(`*opcode-of-c3* `m8 m5)` and `m8` would have the structure:
>
> `(`*opcode-of-c2* `m3 m7 m4)`
>
> `!class (c4 (c2 {m6: m7:}))` has the following structure:
>
> `(`*opcode-of-c4* `m3 m6 m7 m4);m3=1 m6=2 m7=3 m4=4`.

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.

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 `|[]`, `|us`, `|bl`, `|st`, `|did`, `|fid` and `|nid` depending on the context.

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; class names starting with an underscore denote convenience constructs used to build other classes;
- 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.

view

Structure

```
!class (view[] ijt:us sln:nb res:us grp:grp org: sync:nb)
```

`ijt` – injection-time: a timestamp representing the creation time of the object. When a view originates from a remote node, the injection time is the time of reception of the object by the target node.

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

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

`grp` – group: a reference to the group the view lives in. This value cannot be `nil`, i.e. no view can exist outside of a group.

`org` – origin: a reference to the group (if any) that produced the object locally (i.e. on the same computing node), or a reference to the remote node that ejected the object.

`sync` – synchronization mode: a number in {0,1} defining the rule for eligibility for reduction. 0 means that the view will be eligible when it becomes salient (synchronization on front), 1 means that the view will be eligible when it is salient (synchronization on state).

Development

```
[ijt sln res grp org sync]
```

Description

A class defining the data for any view on any object, except instantiated programs and groups.

grp_view

Structure

```
!class (grp_view[] ijt:us sln:nb res:us grp:grp org: sync:nb vis:nb cov:nb)
```

`ijt` – defined as for `view`.

`sln` – defined as for `view`.

`res` – defined as for `view`.

`grp` – defined as for `view`.

`org` – defined as for `view`.

`sync` – defined as for `view`.

`vis` – visibility: a number in [0,1] representing the final visibility of the group.

`cov` – copy on visibility: a number in {0,1}. When set to 1, whenever the group is visible from another group G, then all the salient group's members are copied into G. Predefined macros: `COV_ON` and `COV_OFF`.

Development

`[ijt sln res grp org sync vis cov]`

Description

A class defining the data for any view on groups.

`pgm_view`

Structure

`!class (pgm_view[] ijt:us sln:nb res:us grp:grp org: sync:nb act:nb)`

`ijt` - defined as for `view`.

`sln` – defined as for `view`.

`res` – defined as for `view`.

`grp` – defined as for `view`.

`org` – defined as for `view`.

`sync` – defined as for `view`.

`act` – activation: a number in [0,1] representing the final activation of the instantiated reactive object.

Development

`[ijt sln res grp org sync act]`

Description

A class defining the data for any view on instantiated programs.

`_obj`

Structure

`!class (_obj :~ psln_thr:nb)`

`psln_thr` – a threshold to control the propagation of saliency changes: this is a number in [0,1] such as if the saliency change is lesser than the threshold, then the propagation of the change is inhibited. In particular, if `psln_thr` is set to 1, then no propagation will ever occur, and if `psln_thr` is set to 0, then any change will be propagated. The changes of saliency are propagated to all the markers in the object’s marker set. This means that the change will be applied to all said markers in their respective views. The propagation is not applied further (e.g. to the markers of the markers) unless the markers have their own `psln_thr` value set to a value lesser than 1. When a marker enables propagation, the saliency change does not only propagate to the marker’s own markers, but also to the objects it may refer to. Propagating saliency to an object is treated by the executive as if there had been issued a call to `mod` on the saliency of the object requesting (`sln` is mediated) the same change, transformed into the range of the target object. The executive detects and prevents recursive propagation.

Development

`... psln_thr`

Description

A template class defining common data for all objects.

`ptn`

Structure

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

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

`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

A pattern.

|ptn

Structure

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

Same constituents as in a pattern.

Development

Same development as `ptn`.

Description

An anti-pattern.

pgm

Structure

`!class (pgm (_obj {tpl:[::ptn] inputS:[] guards:[::xpr] prods:}))`

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

`inputs` – a set of patterns defining the inputs.

`guards` - a set of expressions.

`prods` – set of productions: a production is a call to an internal command (`icmd`).

Development

```
(pgm
[patterns]; tpl
[patterns]; inputs
[expressions]; guards
[productions]; prods
psln_thr)
```

Description

Program. Patterns in programs *must specify the opcode of the expression specified in their skeletons*. It is not allowed to define variables on said opcodes (in contrast with the patterns found in the `red` operator). This applies to the patterns specified in both the template arguments and the inputs of the 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 (_obj {tpl:[::ptn] inputS:[] guards:[::xpr] prods:}))`

Same constituents as in a program.

Development

Same development as for a program.

Description

An anti-program, i.e. a program that outputs its productions when *no* object matches its inputs. No production shall contain any reference to variables in the inputs. To be executable, an anti-program needs to be instantiated, i.e. passed actual template parameters.

_grp

Structure

`!class (grp (_obj {upr:nb spr:nb sln_thr:nb act_thr:nb vis_thr:nb c_sln:nb c_sln_thr:nb` c_act:nb c_act_thr:nb dcy_per:nb dcy_tgt:nb dcy_prd:nb sln_chg_thr:nb sln_chg_prd:nb act_chg_thr:nb act_chg_prd:nb avg_sln:nb high_sln:nb low_sln:nb avg_act:nb high_act:nb low_act:nb high_sln_thr:nb low_sln_thr:nb sln_ntf_prd:nb high_act_thr:nb low_act_thr:nb act_ntf_prd:nb ntf_new:nb low_res_thr:nb ntf_grps:[] :~}))

upr - update-period: an integral number representing the period with which the group (actually, the rMem managing the group) updates the saliency and activation values of the objects in the group. This value is actually a multiple of the base period defined for the entire system (see Annex 2). N.B.: the resilience values are updated at a fixed period, defined for all groups, system-wide.

spr – signaling period: an integral number representing the period with which the group triggers the execution of programs and models. This value is actually a multiple of the base period defined for the entire system (see Annex 2).

sln_thr – saliency threshold: a number in [0,1] specifying the saliency threshold of the group. Any object projected on the group with a saliency lesser than or equal to the threshold will be considered non salient, salient otherwise. Thus, when sln_thr is set to 1, no object can be salient in the group, and when sln_thr is set to 0, only objects with a strictly positive saliency will be salient.

act_thr – activation threshold: a number in [0,1] specifying the activation threshold of the group. Any instantiated reactive object projected on the group with an activation lesser than or equal to the threshold will be considered not active, active otherwise. Thus, when act_thr is set to 1, no object can be active in the group, and when act_thr is set to 0, only objects with a strictly positive activation will be active.

vis_thr – visibility threshold: a number in [0,1] specifying the visibility threshold of the group. Any group projected on the group with a visibility lesser than or equal to the threshold will be considered not visible, visible otherwise. Thus, when vis_thr is set to 1, no group can be visible in the group, and when vis_thr is set to 0, only groups with a strictly positive visibility will be visible in the group.

c_sln – content saliency: a number in [0,1] specifying the saliency of the content of the group as a whole. This value is compared against the c_sln_thr value (see below) to determine whether the group is c-salient or not.

c_sln_thr – content saliency threshold: a number in [0,1]. Similar to sln_thr but applies to c_sln instead of sln. When a group is not c-salient, none of the objects it contains can be inputs for any reactive object in the group. When a group is c-salient, the objects it contains can be inputs if they are salient themselves.

c_act – content activation: a number in [0,1] specifying the activation of the content of the group as a whole. This value is compared against the c_act_thr value (see below) to determine whether the group is c-active or not.

c_act_thr – content saliency threshold: a number in [0,1]. Similar to act_thr but applies to c_act instead of act. When a group is not c-active, none of the instantiated reactive objects it contains can run. When a group is c-active, its instantiated reactive objects can run if they are active themselves.

dcy_per – decay percentage: a number in [-1,1] representing the decay of a given value (dcy_tgt below) in percentage.

dcy_tgt – decay target value: a number in {0,1} specifying the value to apply the decay on: 0 for the saliencies of all objects in the group, 1 for the saliency threshold of the group. Predefined macros: DCY_SLN and DCY_SLN_THR.

dcy_prd – decay period: an integral number specifying a multiple of the update period defined for the group (upr). The decay will be performed as follows: the target value will undergo a decay specified by dcy_per over a period equal to the upr*dcy_prd. Decay is implemented as calls to mod on the target value over the specified period. For example, if dcy_prd is set to 4 and dcy_per to 50, then this is equivalent to calling mod.*target-value* 12.5 four times in a row, once per update period.

sln_chg_thr – threshold on saliency changes: a number in [0,1] under which a change of an object's saliency will not be notified.

sln_chg_prd – saliency change monitoring period: an integral number specifying a multiple of the group update period (upr). This defines the period over which the group accumulates the changes in saliency of its members. At the end of each such periods, the accumulated changes are compared to sln_chg_thr and changes greater than this threshold are notified using a marker mk.sln_chg (see section 4.3).

act_chg_thr – threshold on activation changes: a number in [0,1] under which a change of an object's activation will not be notified.

act_chg_prd – activation change monitoring period: an integral number specifying a multiple of the group update period (upr). This defines the period over which the group accumulates the changes in activation of its members. At the end of each such periods, the accumulated changes are compared to act_chg_thr and changes greater than this threshold are notified using a marker mk.act_chg (see section 4.3).

avg_sln – average saliency: a number in [0,1] representing the average saliency of the group's member.

high_sln – highest saliency: a number in [0,1] representing the highest saliency of the group's member.

low_sln – lowest saliency: a number in [0,1] representing the lowest saliency of the group's member.

avg_act – average activation: a number in [0,1] representing the average activation of the group's member.

high_act – highest activation: a number in [0,1] representing the highest activation of the group's member.

low_act – lowest activation: a number in [0,1] representing the lowest activation of the group's member.

high_sln_thr – high saliency notification threshold: a number in [0,1]. Whenever an object's saliency becomes higher than the threshold, this event is notified using a marker mk.high_sln (see section 4.3).

low_sln_thr – low saliency notification threshold: a number in [0,1]. Whenever an object's saliency becomes lower than the threshold, this event is notified using a marker mk.low_sln (see section 4.3).

sln_ntf_prd – saliency notification period: an integral number specifying a multiple of the group update period (upr). This defines the period at the end of which notification will occur for any object having become more salient than the high_sln_thr or less salient than low_sln_thr, and whose saliency is still in the same relationship with regards to the thresholds.

high_act_thr – high activation notification threshold: a number in [0,1]. Whenever an object's activation becomes higher than the threshold, this event is notified using a marker mk.high_act (see section 4.3).

low_act_thr – low activation notification threshold: a number in [0,1]. Whenever an object's activation becomes lower than the threshold, this event is notified using a marker mk.low_act (see section 4.3).

act_ntf_prd – activation notification period: an integral number specifying a multiple of the group update period (upr). This defines the period at the end of which notification will occur for any object having become more active than the high_act_thr or less active than low_act_thr, and whose activation is still in the same relationship with regards to the thresholds.

ntf_new – a flag in {0,1} that turns on (1) or off (0) the notification of objects being injected in the group for the first time.

low_res_thr – low resilience notification threshold: a number in [0,1]. Whenever an object's resilience becomes lower than the threshold, this event is notified using a marker mk.low_res (see section 4.3).

ntf_grps – notification groups: a set of references to the destination groups where notifications are to be injected. When a reference is nil, the destination is the group itself. This set cannot be altered once the group is created.

Development

```
upr spr sln_thr act_thr vis_thr c_sln c_sln_thr c_act c_act_thr dcy_per dcy_tgt dcy_prd
sln_chg_thr sln_chg_prd act_chg_thr act_chg_prd avg_sln high_sln low_sln avg_act high_act
low_act high_sln_thr low_sln_thr sln_ntf_prd high_act_thr low_act_thr act_ntf_prd ntf_new
low_res_thr ntf_grps
...
psln_thr
```

Description

A template class defining data for groups (`grp` and `rgrp`).

grp

Structure

```
!class (grp (_grp nil))
```

Development

```
(grp
upr spr sln_thr act_thr vis_thr c_sln c_sln_thr c_act c_act_thr dcy_per dcy_tgt dcy_prd
sln_chg_thr sln_chg_prd act_chg_thr act_chg_prd avg_sln high_sln low_sln avg_act high_act
low_act high_sln_thr low_sln_thr sln_ntf_prd high_act_thr low_act_thr act_ntf_prd ntf_new
low_res_thr ntf_grps
psln_thr)
```

Description

A group.

_fact

Structure

```
!class (_fact (_obj {obj: after:us before:us cfd:nb :~}))
```

`obj` – a reference to an object.

`after` – the low end of the time interval when the fact holds (in microseconds).

`before` – the high end of the time interval when the fact holds (in microseconds).

`cfd` – confidence: a number in [0,1] indicating the plausibility that the fact actually holds .

Development

```
_fact obj after before cfd psln_thr ...
```

Description

A template class for defining facts and similar classes.

fact

Structure

```
!class (fact (_fact nil))
```

Development

```
(fact obj after before cfd psln_thr)
```

Description

A fact.

|fact

Structure

```
!class (|fact (_fact nil))
```

Same constituents as in a fact.

Development

Same development as for a fact.

Description

The negation of a fact.

pred

Structure

`!class (pred (_obj {obj:}))`

`obj` – the predicted fact.

Development

```
(pred obj psln_thr)
```

Description

A .prediction

goal

Structure

`!class (goal (_obj {obj: actor:ent }))`

`obj` – the target fact.

`actor` – the system pursuing the goal (it may be self, or any entity in the environment that manifests intentions).

Development

```
(goal obj actor psln_thr)
```

Description

A goal.

cst

Structure

`!class (cst (_obj {tpl_args:[] objs:[] fwd_fun:[] bwd_fun:[] out_grps}))`

`tpl_args` – set of variables used as template arguments..

`objs` – the patterns of a composite state.

`fwd_fun` – set of expressions used as guards during forward chaining.

`bwd_fun` – set of expressions used as guards during backward chaining.

`out_grps` – set of groups where to inject instances of the composite state.

Development

```
(cst
[variables]; tpl args
[expressions]; patterns
[expressions]; forward guards
[expressions]; backward guards
[groups]; output groups
psln_tr)
```

Description

A composite state.

mdl

Structure

`!class (mdl (_obj {tpl_args:[] objs:[] fwd_fun:[] bwd_fun:[] out_grps str:nb cnt:nb sr:nb dsr:nb}))`

`tpl_args` – set of variables used as template arguments..

`objs` – the patterns of a model (only two).

`fwd_fun` – set of expressions used as guards during forward chaining.

`bwd_fun` – set of expressions used as guards during backward chaining.

`out_grps` – set of groups where to inject the predictions made by the model.

`str` – strength of the model, in [0,1].

`cnt` – number of evidences for a model (i.e. the number of times it predicted something).

`sr` – success rate of the model: number of times the mdoel predicted well divided by `cnt`.

`dsr` – derivative of the success rate, i.e. the success rate as it was at the former to last evidence.

Development

```
(mdl
[variables]; tpl args
[expressions]; patterns
[expressions]; forward guards
[expressions]; backward guards
[groups]; output groups
str cnt sr dsr
psln_tr)
```

Description

A model.

icst

Structure

```
!class (icst (_obj {cst:cst tpl_args:[] args:[]}))
```

Development

```
(icst cst
[values];tpl_args
[values-or-variables];args
psln_thr)
```

Description

An instantiated composite state.

imdl

Structure

```
!class (imdl (_obj {mdl:mdl tpl_args:[] args:[]}))
```

Development

```
(imdl mdl
[values];tpl_args
[values-or-variables];args
psln_thr)
```

Description

An instantiated model.

icmd

Structure

```
!class (icmd function:fid args:[])
```

`function` – function of the executive.

`args` – arguments of the function: a list of arguments, i.e. a set where the order of its members matches the order of the formal arguments of the function.

Development

```
(icmd function
[expressions]; args
)
```

Description

A call to an internal command (also called a function of the executive), sent in asynchronous mode - there is no return value.

cmd

Structure

`!class (cmd (_obj {function:fid device:did args:[]}))`

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

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

`args` – arguments of the function: a list of arguments, i.e. a set where the order of its members matches the order of the formal arguments of the function.

Development

```
(cmd function device
[expressions]; args
psln_thr
)
```

Description

A call to a device function, i.e. a command, sent in asynchronous mode - there is no return value.

ent

Structure

`!class (ent (_obj nil))`

Development

`(ent psln_thr)`

Description

An entity. Entities do not hold any code: they are meant to be referenced by other objects to differentiate from each other. For example, entities are commonly used to indicate what markers apply to - like in `(mk.position self` *a-position* `...)` where `self` is a predefined entity.

ont

Structure

`!class (ont (_obj nil))`

Development

`(ont psln_thr)`

Description

A member of an ontology. Such objects do not hold any code: they are meant to be referenced by other objects to differentiate from each other.

dev

Structure

`!class (dev (_obj nil))`

Development

`(dev psln_thr)`

Description

A device. Device identifiers have to match the ones used by the messaging infrastructure.

nod

Structure

`!class (nod (_obj id:nid))`

`id` – identifier*:* a numerical identifier[9] (see Annex 3).

Development

`(nod` *id* `psln_thr)`

Description

A computing node, i.e. an rMem. Node identifiers have to match the ones used by the messaging infrastructure.

ipgm

Structure

`!class (ipgm (_obj {code: args:[] run:bl tsc:us nfr:bl}))`

`code` – the instantiated code*:* either a program or an anti-program.

`args` – the instantiation values: a set of expressions or references to objects.

run – a Boolean value indicating if the program shall execute only once (false) or whenever possible (true).

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

`nfr` – notify flag: a number in {0,1} representing whether or not the rCores executing an instantiated reactive object notifies the reduction of input objects (`mk.rdx`). Predefined macros: `NOTIFY` and `SILENT`.

Development

`(ipgm` code
`[`*expressions-or-references-to-objects*`]`; args
`run tsc nfr`
`psln_thr`
`)`

Description

An instantiated program or anti-program.

icpp_pgm

Structure

`!class (icpp_pgm (_obj {code:st args:[] run:bl tsc:us nfr:bl}))`

`code` – the instantiated code*:* the name of a C++ program to load.

`args` – the instantiation values: a set of expressions or references to objects.

run – a Boolean value indicating if the program shall execute only once (false) or whenever possible (true).

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

`nfr` – notify flag: a number in {0,1} representing whether or not the rCores executing an instantiated reactive object notifies the reduction of input objects (`mk.rdx`). Predefined macros: `NOTIFY` and `SILENT`.

Development

`(icpp_pgm` code
`[`*expressions-or-references-to-objects*`]`; args
`run tsc nfr`
`psln_thr`
`)`

Description

An instantiated C++ program; the program is supplied by the programmer as a member of the usr.operators shared library.

[9] N.B.: a cluster running a Replicode application can be composed of 256 nodes at most.

perf

Structure

```
!class (perf (_obj {rj_ltcy:nb d_rj_ltcy:nb tj_ltcy:nb d_tj_ltcy:nb}))
```

`rj_ltcy` – the delay between the time of injection of a reduction job and it!s processing time, averaged over the `perf_sampling_period`.

`d_rj_ltcy` – the value above, at the last sampling time.

tj_ltcy – the delay a thread had to wait for processing the job, averaged over the sampling period.

`d_rj_ltcy` – the value above at the last sampling time.

Development

```
(perf rj_ltcy d_rj_ltcy tj_ltcy d_tj_ltcy
psln_thr
)
```

Description

A performance counter. All values in us encoded as floats.

4.3 Marker Classes

`mk.rdx`

Structure

```
!class (mk.rdx (_obj {code: inputs:[] prods:[::cmd]}))
```

`code`: a reference to the instantiated program or model having performed the reduction.

`inputs`: a set containing references to the input objects that were reduced. Is emtpy in case of anti-programs and input-less programs.

`prods`: a set containing the commands that result from the reduction.

Development

(mk.rdx code
[*references-to-input-objects*]; inputs
[*commands*]; prods
psln_thr)

Description

Notification of a reduction. Values in `prods` are expressions resulting from the evaluation of the commands defined in the reactive object. The commands `mod` and `set` are not notified since they are requests for change and not actual changes. Groups notify the actual changes of control values using dedicated markers (see below).

`mk.grp_pair`

Structure

```
!class (mk.grp_pair (_obj {primary:grp secondary:grp}))
```

`primary`: a reference to a primary group.

`secondary`: a reference to a secondary group.

Development

```
(mk.grp-pair primary_group secondary_group psln_thr)
```

Description

The primary group hosts models that have high a success rate enough to be considered meaningful for the operation of the system. When models are no longer considered good enough, they are run in the secondary group, to give them a chance to get a better success rate, and thus, to be moved back to the primary group.

mk.low_sln

Structure

```
!class (mk.low_sln (_obj obj:))
```

`obj` - object: a reference to the object having reached a low saliency.

Development

```
(mk.low_sln obj psln_thr)
```

Description

Notification of an object having reached a low saliency. This notification is issued by rMems following an object having reached a saliency lower than the low saliency threshold in a group. This group is accessible using the `org` member of the marker's view.

mk.high_sln

Structure

```
!class (mk.high_sln (_obj obj:))
```

`obj` - object: a reference to the object having reached a high saliency.

Development

```
(mk.high_sln obj psln_thr)
```

Description

Notification of an object having reached a high saliency. This notification is issued by rMems following an object having reached a saliency greater than the high saliency threshold in a group. This group is accessible using the `org` member of the marker's view.

mk.low_act

Structure

```
!class (mk.low_act (_obj obj:))
```

`obj` - object: a reference to the object having reached a low activation.

Development

```
(mk.low_act obj psln_thr)
```

Description

Notification of an object having reached a low activation. This notification is issued by rMems following an object having reached an activation lower than the low activation threshold in a group. This group is accessible using the `org` member of the marker's view.

mk.high_act

Structure

```
!class (mk.high_act (_obj obj:))
```

`obj` - object: a reference to the object having reached a high activation.

Development

```
(mk.high_act obj psln_thr)
```

Description

Notification of an object having reached a high activation. This notification is issued by rMems following an object having reached an activation greater than the high activation threshold in a group. This group is accessible using the `org` member of the marker's view.

mk.low_res

Structure

`!class (mk.low_res (_obj obj:))`

`obj` - object: a reference to the object having reached a low resilience.

Development

`(mk.low_res obj psln_thr)`

Description

Notification of an object having reached a low resilience. This notification is issued by rMems following an object having reached a resilience lower than the low resilience threshold in a group. This group is accessible using the `org` member of the marker's view.

mk.sln_chg

Structure

`!class (mk.sln_chg (_obj {obj: chg:nb}))`

`obj` - object: a reference to the object whose saliency changed.

`chg` – saliency change: a number in [0,1] representing the change of saliency.

Development

`(mk.sln_chg obj chg psln_thr)`

Description

Notification of a change of an object's saliency in a group. This group is accessible using the `org` member of the marker's view. This notification is issued by rMems when the object's saliency has changed by an amount greater than the saliency change threshold defined by the group.

mk.act_chg

Structure

`!class (mk.act_chg (_obj {obj: chg:nb}))`

`obj` - object: a reference to the object whose activation changed.

`chg` – activation change: a number in [0,1] representing the change of activation.

Development

`(mk.act_chg obj chg psln_thr)`

Description

Notification of a change of an object's activation in a group. This group is accessible using the `org` member of the marker's view. This notification is issued by rMems when the object's saliency has changed by an amount greater than the saliency change threshold defined by the group.

mk.new

Structure

`!class (mk.new (_obj obj:))`

`obj` - object: a reference to the newly injected object.

Development

`(mk.new obj psln_thr)`

Description

Notification of the injection of a new object in a group. This group is accessible using the `org` member of the marker's view. This notification is issued by rMems whenever an object is injected in a group for the first time, or is visible from that group for the first time.

4.4 Entities

`self`

Description

The application, i.e. the system as an actor in an application domain.

4.5 Groups

`stdin`

Description

The standard input group. This is the group that receives object from remote nodes (i.e. rMems).

`stdout`

Description

The standard output group. This a group from which objects are typically ejected to remote nodes. Notice however, that it is allowed to eject objects from any other group.

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

; utilities.
!class (_obj :~ psln_thr:nb)
!class (_grp (_obj {upr:nb sln_thr:nb act_thr:nb vis_thr:nb c_sln:nb c_sln_thr:nb c_act:nb
c_act_thr:nb dcy_per:nb dcy_tgt:nb dcy_prd:nb dcy_auto:nb sln_chg_thr:nb sln_chg_prd:nb
act_chg_thr:nb act_chg_prd:nb avg_sln:nb high_sln:nb low_sln:nb avg_act:nb high_act:nb
low_act:nb high_sln_thr:nb low_sln_thr:nb sln_ntf_prd:nb high_act_thr:nb low_act_thr:nb
act_ntf_prd:nb ntf_new:nb low_res_thr:nb ntf_grps:[::grp] :~}))
!class (_fact (_obj {obj: after:us before:us cfd:nb :~}))
!class (val (_obj val:))
!class (val_hld val:)

; mapping low-level objects -> r-code.
!class (ent (_obj nil))
!class (ont (_obj nil))
!class (dev (_obj nil))
!class (nod (_obj id:nid))
!class (view[] sync:bl ijt:us sln:nb res:nb grp:grp org:)
!class (grp_view[] sync:bl ijt:us sln:nb res:nb grp:grp org: cov:bl vis:nb)
!class (pgm_view[] sync:bl ijt:us sln:nb res:nb grp:grp org: act:nb)
!class (ptn skel:xpr guards:[::xpr])
!class (|ptn skel:xpr guards:[::xpr])
!class (pgm (_obj {tpl:[::ptn] inputs:[] guards:[::xpr] prods:}))
!class (|pgm (_obj {tpl:[::ptn] inputs:[] guards:[::xpr] prods:}))
!class (grp (_grp nil))
!class (icmd function:fid args:[])
!class (cmd (_obj {function:fid args:[]}))
!class (cmd_arg (_obj {function:fid arg:ont}))
!class (ipgm (_obj {code: args:[] run:bl tsc:us nfr:bl}))
!class (icpp_pgm (_obj {code:st args:[] run:bl tsc:us nfr:bl}))

; mapping low-level markers -> r-code.
!class (mk.rdx (_obj {code: inputs:[] prods:[]}))
!class (mk.low_sln (_obj obj:))
!class (mk.high_sln (_obj obj:))
!class (mk.low_act (_obj obj:))
!class (mk.high_act (_obj obj:))
!class (mk.low_res (_obj obj:))
!class (mk.sln_chg (_obj {obj: chg:nb}))
!class (mk.act_chg (_obj {obj: chg:nb}))
!class (mk.new (_obj obj:))

; mapping high-level objects -> r-code.
!class (mk.val (_obj {obj: attr:ont val:}))
!class (mk.act (_obj {actr:ent cmd:cmd}))
!class (fact (_fact nil))
!class (|fact (_fact nil))
!class (var (_obj nil))
!class (cst (_obj {tpl_args:[] objs:[] fwd_fun:[] bwd_fun:[] out_grps:[::grp]}))
!class (mdl (_obj {tpl_args:[] objs:[] fwd_fun:[] bwd_fun:[] out_grps:[::grp] str:nb cnt:nb
sr:nb dsr:nb}))
!class (icst (_obj {cst:cst tpl_args:[] args:[]}))
!class (imdl (_obj {mdl:mdl tpl_args:[] args:[]}))
!class (pred (_obj {obj:fact}))
!class (goal (_obj {obj:fact actr:ent}))
!class (success (_obj obj:))
!class (mk.grp_pair (_obj {primary:grp secondary:grp}))
```

```
; internal control structures.
!class (sim (super_goal:)); simulation head.
!class (sol (super_goal: contribution:)); solution.
!class (cmt (requirement:)); commitment.

; mapping operator opcodes -> r-atoms.
!op (_now):us
!op (rnd :nb):nb
!op (equ : :):
!op (neq : :):
!op (gtr : :):
!op (lsr : :):
!op (gte : :):
!op (lse : :):
!op (add : :):
!op (sub : :):
!op (mul :nb :nb):nb
!op (div :nb :nb):nb
!op (dis : :):nb
!op (ln :nb):nb
!op (exp :nb):nb
!op (log :nb):nb
!op (e10 :nb):nb
!op (syn :):
!op (ins : :[] :bl :us :bl):
!op (red :[] :[] :[]):[]
!op (fvw : :):

; operator aliases.
!def now (_now)
!def = equ
!def <> neq
!def > gtr
!def < lsr
!def <= gte
!def >= lse
!def + add
!def - sub
!def * mul
!def / div
!def ¥ syn

; mapping devices -> r-atoms.
!def exe 0xA1000000; the executive.

; mapping device functions -> r-atoms.
!dfn (_inj :
!dfn (_eje : : :nid)
!dfn (_mod : :nb)
!dfn (_set : :nb)
!dfn (_new_class
!dfn (_del_class
!dfn (_ldc :st)
!dfn (_swp :nb)
!dfn (_prb :nb :st :st :[])
!dfn (_stop)

; device functions aliases.
!def (inj args) (icmd _inj args)
!def (eje args) (icmd _eje args)
!def (mod args) (icmd _mod args)
```

```
!def (set args) (icmd _set args)
!def (new_class args) (icmd _new_class args)
!def (del_class args) (icmd _del_class args)
!def (ldc args) (icmd _ldc args)
!def (swp args) (icmd _swp args)
!def (prb args) (icmd _prb args)
!def (stop args) (icmd _stop args)

; various constants.
!counter __constant 0
!def OFF __constant
!def ON __constant

; parameters for tuning the behavior of reactive objects.
; member nfr.
!def SILENT false ; no notification upon production
!def NOTIFY true ; notification upon productions

; parameters for tuning the behavior of groups.
!def DCY_SLN 0
!def DCY_SLN_THR 1
!def COV_ON true
!def COV_OFF false

!def SYNC_FRONT true
!def SYNC_STATE false

!def RUN_ALWAYS true
!def RUN_ONCE false
```

Notice the last five directives for loading user-defined object classes, operators, devices and device functions. 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 instances of rMem distributed over the cluster:

- rMems are managing groups (see section 6.2 of the main document);
- each rMem instantiates an rCore. rCores are responsible for the execution of instantiated programs and models (see section 4.2 of the main document);

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

The executive defines two values to be used for defining the time periods that parameterize cyclic behaviors. These values are specified in microseconds:

- a base period: the periods `upr` and `spr` defined by groups are expressed as multiple of this base value;
- the update period of the resilience values. This period is defined per system, not per group.

These two values are initialized by the function `_start`.

2 rCore

The rCore is completely hidden by the rMem that instantiates it: there is no need to include it in the language specification. It suffices to describe its role The role of an rCore is:

- to execute programs – i.e. to perform reductions;
- to build reduction markers – i.e. used by the rMem to notify reductions.

N.B.: the communication protocol between an rCore and its rMem is not part of the specification.

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 command objects (`cmd`). These objects are passed to the rMem for execution. The latter can result in:

- injection of objects in the group specified by the command;
- ejection of objects to a remote computing node;
- request for modification of mediated values;
- sending of a command to an external device;
- other actions (loading code, creation of new object classes, etc.);

An rCore is expected to define an array called a value storage, to store template arguments and intermediate results when evaluating expressions in code – value storages are instances of `r_code::ReductionInstance` (see Annex 4).

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

Marker sets are volatile: this means that during the execution of a program, the marker sets of the inputs may change *after* a successful match. In other words, if an input object is scrutinized for its markers in code fragments located *after* the guards of the pattern that was matched, then said markers may not be the same. This practice – i.e. a many step pattern matching – is however strongly discouraged for efficiency reasons (avoiding too many invocations of `red`) and for preserving machine-readability of code (it is easier to craft code that focuses on one pattern construct (`ptn`) rather than having to search code for conditions located in many places).

View sets are also volatile.

Reduction Markers Upon a successful reduction, the rCore shall construct a reduction marker (mk.rdx defined in the Annex 1) and pass it to the rMem. This construction can happen only if the program has its `nfr` (for “notify reduction”) member set to 1.

3 rMem

3.1 Role

The role of an rMem is:

- to update periodically mediated values, as 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 group – except the resilience values, which are updated at a fixed period defined for the entire system.
- to detect instantiated programs gaining activation, and loading them in the rCore for execution; symmetrically, the rMem unloads an instantiated program from an rCore when it loses activation;
- to signal active input-less instantiated programs at each of the periods specified by their `tsc` member.
- to detect object gaining saliency, and passing them to its rCore as input objects;
- to manage the allocation in memory, by 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 programs anymore. It will however still be visible in patterns targeting objects that hold a reference to it;
- to detect identical objects and eliminate duplicates - identity is the identity of r-code regardless of the mediated values. When duplicate exist in a given group, the view is rebuilt using the greatest values available among the different views of the duplicates;
- to notify programs of key events occurring in the group, depending on the parameters of the group;
- to implement the behavior of groups as defined in section 6.2 of the main document (decay, exposition of statistics, etc.).

3.2 Functions

All the functions of the executive are implemented by the rMems. The prototype of these functions is:

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

See Annex 4 for the definition of `ExecutionContext`.

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 – see Annex 3 for the definition of r-code.

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.

`_inj`

Arguments

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

1: the r-atom leading the definition of a view for the object.

Description

Injects a new object inside the group specified in the accompanying view. The `ijt` member of the view holds a value defining a lower boundary on the injection time. For example, the value `(+ now 100000)` means that the object shall be injected *after* 100 milliseconds, counted from the time of the invocation of `inj`. The `grp` member specifies the group where to inject the object. The member org shall be set eventually by the rMem to the group where the reduction occurred.

Injection commands are batched: this is to make sure that injected objects can only be available as input objects when their markers produced in the same batch are also available. A batch is defined as the set of productions resulting from the inputs processed at a given time.

rMems are responsible for eliminating object duplicates: shall a batch deliver an object that already exist in the system, then the new object is not injected; however, if the object was to be injected in a group where it was not already present, then the object is actually projected in the group, with the specified view. If a view for the object already exist in the group, then the view is adjusted to hold the highest values available among the two views (the old one and the new one).

_eje

Arguments

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

1. the r-atom leading the definition of a view.

2. the r-atom identifying a destination node.

Description

Ejects a new object in the remote node specified by the third argument, in a group specified by the view. Only stdin and stdout are allowed for specifying the destination group. Upon reception, the member ijt shall be set to the reception time. Elimination of duplicates on the destination group in the remote node applies as for _inj.

_mod

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}} U {forever}.

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 is either a saliency, and activation, a visibility or a resilience value), 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 group. The resilience values are a special case: such values 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.

_set

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. If, among the invocations of _set, there is no -1 value, the resilience changes to the value to the newly computed average of the requests `_new_class`

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 group where the call is issued.

`_del_class`

Arguments

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

Description

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

`_ldc`

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 binary input stream (r-code form). `_ldc` loads an image from disk and maps it contents into memory.

`_swp`

Arguments

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

Description

Swaps the content of the entire node to/from disk. When swapped to disk. The r-code is stored as an image (see section 2 of the Annex 3). Images stored/loaded by `_swp` bear the name `image.`<nid>.rcode where `nid` is the identifier of the node.

`_stop`

Arguments

None.

Description

Terminates the rMem and its rCore, on the node where the function is executed. The memory is flushed and the rMem and its rCore are terminated.

3.3 Notifications

Notifications by an rMem are carried by different object classes, depending on the event to be notified:

- process objects (pro): used to notify the beginning of the execution of a device function. Execution termination markers (`mk.xet`) are attached to process objects upon termination of the device function;
- reduction markers (`mk.rdx`), as built by rCores upon successful reduction;
- `mk.new`: notification of the injection of a new object in a group. This group is the `org` member of the marker's view. This notification is issued by rMems whenever an object

is injected in a group for the first time, or is visible from that group for the first time.

- mk.low_sln: Notification of an object having reached a low saliency. This notification is issued by rMems following an object having reached a saliency lower than the low saliency threshold in a group. This group is the org member of the marker.
- mk.high_sln: Notification of an object having reached a high saliency. This notification is issued by rMems following an object having reached a saliency greater than the high saliency threshold in a group. This group is the org member of the marker.
- mk.low_act: Notification of an object having reached a low activation. This notification is issued by rMems following an object having reached an activation lower than the low activation threshold in a group. This group is the org member of the marker.
- mk.high_act: Notification of an object having reached a high activation. This notification is issued by rMems following an object having reached an activation greater than the high activation threshold in a group. This group is the org member of the marker's view.
- mk.low_res: <description>
- mk.sln_chg. Notification of a change of an object's saliency in a group. This group is the org member of the marker's view. This notification is issued by rMems when the object's saliency has changed by an amount greater than the saliency change threshold defined by the group.
- mk.act_chg: Notification of a change of an object's activation in a group. This group is the org member of the marker's view. This notification is issued by rMems when the object's saliency has changed by an amount greater than the saliency change threshold defined by the group.

N.B.: some of these notifications are enabled by parameters found in groups – see Annex 1 for details.

Notification of events occurring in a group are to be injected by the rMem in the notification group (ntf_grp) specified by the group.

3.4 Interface

rMems implement the following interface (defined in a shared object, r_code.so, the rMem itself being defined in another shared object, r_exec.so):

```
// file r_core.h

using namespace std;
namespace r_code{

struct dll_export ObjectReceiver{
   enum{
      INPUT_GROUP=0,
      OUTPUT_GROUP=1
   }Destination;
   virtual void receive(
      Object *object,
      vector<Atom> viewData,
      uint32       nodeID,
      Destination  destination)=0;
   virtual void receive(
      Object *object,
      vector<Atom> viewData,
      uint32       nodeID,
      Group        *destination
   )=0;
};

struct dll_export Asynchronous{
   virtual void start()=0;
   virtual void stop()=0;
   virtual void resume()=0;
   virtual void suspend()=0;
};

struct dll_export Mem:
public ObjectReceiver,
public Asynchronous{
   static Mem* create(ObjectReceiver* output);
   virtual ~Mem();
};
}
```

The functions `receive` load objects and views in memory. The classes `Atom` and `Object` are defined respectively in Annex 3 and Annex 4.

Annex 3 – R-code Specification

1 Overview

r-code is the name of the internal representation of Replicode constructs.

The content of objects (for example, the code of a program) is shared amongst the groups hosted by the same node. However, the values used to control objects in groups (e.g. `ijt`, `res`, `sln`, `act`/`vis`) must be defined locally per group and cannot be shared. This calls for differentiating the content of the object from the control data.

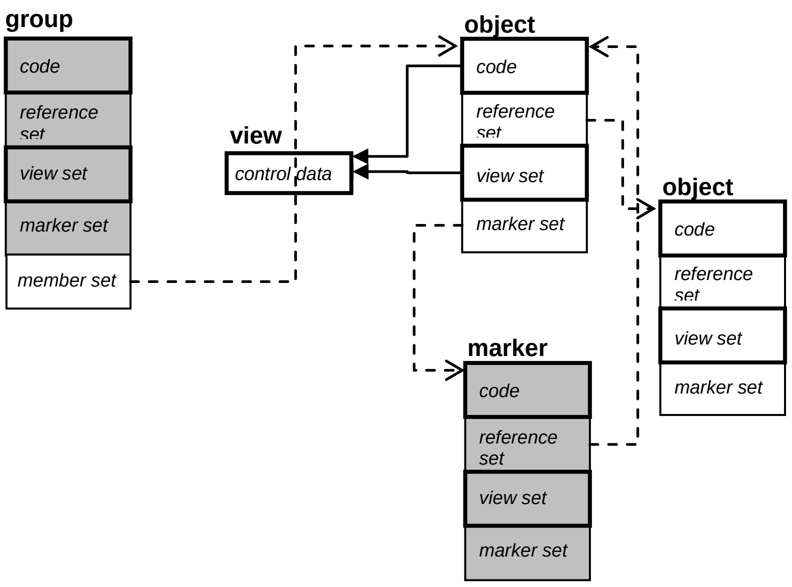

Figure 1 – Overview of Views and Objects

A view is local to a group and defines local data describing an object in the group. Views do not hold any pointer: they are embedded in the code of objects (plain arrows).

An object holds pointers (dashed arrows) to (a) any objects its code may explicitly refer to (reference set) and, (b) the markers attached to it (marker set).

When an object o is an input for a program p in a group g, the executive copies the view of o in g into the code of o. In other words, the overlays of p have each a private copy of o, patched with the correct view.

Groups and markers are also objects (grey segments).

The view data, the object code and the view set are stored in respective contiguous arrays, and encoded in r-code (bold outlining). In contrast, the references held by objects (i.e. markers and explicit references) are stored as raw pointers – this also holds for the set of group members.

When an object is created, its code and reference set are fixed and will not change. However, its view and marker set can change over time (as can the groups' member sets).

2 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 4 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.

The counter-parts of Replicode opcodes are called r-opcodes. Operator r-opcodes are indexes of operators in a table - the Operator Register (see Annex 4).

The preprocessor uses the `!class` directives to define r-atoms for object and marker classes, incrementing the r-opcode when parsing each directive, and deducing their respective arity from the class structures.

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.

2.1 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 1 bit*. The significand part of such a number is thus reduced to 22 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.

r-code defines 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 1 bit and decoding it, by left shifting the r-atom by 1 bit.
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)]
reference pointer	`85`	[*unused* (8) | *index* (16)]
value pointer	`86`	[*unused* (8) | *index* (16)]
ipgm pointer	87	internal use
input obj. pointer	88	internal use
value array pointer	89	internal use
prod. pointer	8A	internal use
output obj. pointer	8B	internal use
deleg. in. obj. ptr	8C	internal use
assignment pointer	8D	internal use
this	`90`	[*unused* (24)]
view	`91`	[*unused* (24)]
mks	`91`	[*unused* (24)]
vws	`91`	[*unused* (24)]
node	`A0`	[*unused* (16) | *identifier* (8)]
device	`A1`	[*node-identifier* (8) | *class-identifier* (8) | *instance-identifier* (8)]
device function	`A2`	[*r-opcode* (16) | *unused* (8)]
chain pointer	`C0`	[*unused* (16) | *number-of-elements* (8)]
set	`C1`	[*unused* (8) | *number-of-elements* (16)]
structured set	`C2`	[*r-opcode* (16) | *arity* (8)]
object opcode	`C3`	[*r-opcode* (16) | *arity* (8)]

marker opcode	C4	[*r-opcode* (16) | *arity* (8)]
operator opcode	C5	[*unused* (8) | *r-opcode* (8) | *arity* (8)]
character string	C6	[*number-of-blocks-of-4-characters* (8) | *number-of-characters* (16)]
timestamp	C7	[*unused* (24)]
group	C8	[*r-opcode* (16) | *arity* (8)]
ipgm	C9	[*r-opcode* (16) | *arity* (8)]
icpp_pgm	CA	[*r-opcode* (16) | *arity* (8)]
ipgm (no inputs)	CB	[*r-opcode* (16) | *arity* (8)]
ipgm (|pgm)	CC	[*r-opcode* (16) | *arity* (8)]
cst	CD	[*r-opcode* (16) | *arity* (8)]
mdl	CE	[*r-opcode* (16) | *arity* (8)]

Constant values are encoded as follows:

`|nb`	`0x3FFFFFFF`
`|bl`	`0x81FFFFFF`
`true`	`0x81000001`
`false`	`0x81000000`
`|[]`	`0xC1000000` (a set with no elements)
`|nid`	`0xA0FFFFFF`
`|did`	`0xA1FFFFFF`
`|fid`	`0xA2FFFFFF`
`|st`	`0xC6000000` (a string with no characters)
`|us`	`0xC7FFFFFF`
`forever`	`0xC7FFFFF0`

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. Such atoms are thereafter called *structural atoms*.

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, markers and set objects, and to 7 for operators.

Structured sets are used to encode structures (think C-style structures) in sets of fixed arity like for example, `view` or `_in_sec` (see Annex 1, section 4.2.2). Semantically, structured sets are identical to object classes. Structured sets are a convenience for allowing writing structures at the source code level without having to specify their opcodes (as required when using object classes). For example, using a structured set for encoding a view allows writing (in the source code) `[ijt sln res grp org]` instead of `(view ijt sln res grp org)`. However, the internal representation specifies the opcode of the view structure using a r-atom of type structured set.

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).

Reference pointers are indexes to retrieve addresses of objects from the reference set (including markers if they have been referenced explicitly). Reference pointers are used to refer to objects that are not derived from computation (e.g. pattern matching, guards and production). For example, an explicit reference to `stdout` is encoded using a reference pointer.

Value pointers are indexes to access the value storage, that is, a structure allocated by the executive to hold (a) actual template parameters for instantiated objects, (b) objects that matched the inputs of a program and, (c) intermediate values resulting from the evaluation of expressions in the code of programs. A value storage is allocated for each overlay and contains the same structure as the program the overlay is an execution of. Value storages are instances of the `class r_code::ReductionInstance` (see Annex 4).

Chain pointers are structures of r-atoms for accessing named members of structures. After the leading r-atom (identifying the chain pointer type), the first r-atom is a pointer to an object (*this*, a reference pointer, or a value pointer), followed by an internal pointer to a member of said object (see section 1.2 in this Annex). In case this member has a deep structure (i.e. is a structure itself), it can be followed by other pointers to members.

The r-atoms *this*, *mks, vws,* and *view* indicate to the executive how to retrieve objects from memory, respectively, the instantiated program currently being executed (*this*), the marker set (*mks*), the view set (*vws*), and the view of an object in the group where the reduction is performed.

Timestamp r-atoms are followed by two 32 bits words containing the time (in microseconds) since 01/01/1970. These words shall not be interpreted as r-atoms but as raw data.

r-atoms are instances of the following class:

```
	// file r_code.h

using	namespace	core;

namespace	r_code{

// Opcodes on 12 bits.
// Indices on 12 bits.
// Element count on 8 bits.
// To define bigger constructs (e.g. large matrices), define hooks to RAM (and derive classes
from Object).
class	dll_export	Atom{
private:	//	trace utilities.
	static	uint8	Members_to_go;
	static	uint8	Timestamp_data;
	static	uint8	String_data;
	static	uint8	Char_count;
	void	write_indents()	const;
public:
	typedef	enum{
		NIL=0x80,
		BOOLEAN_=0x81,
		WILDCARD=0x82,
		T_WILDCARD=0x83,
		I_PTR=0x84,		// internal pointer.
		R_PTR=0x85,		// reference pointer.
		VL_PTR=0x86,		// value pointer.
		IPGM_PTR=0x87,		// r_exec internal: index of data of a tpl arg held by an
ipgm.
		IN_OBJ_PTR=0x88,	// r_exec internal: index of data held by an input object.
		VALUE_PTR=0x89,		// r_exec internal: index of data held by the overlay's
value array.
		PROD_PTR=0x8A,		// r_exec internal: index of data held by the overlay's
production array.
		OUT_OBJ_PTR=0x8B,	// r_exec internal: index of data held by a newly produced
object.
		D_IN_OBJ_PTR=0x8C,	// r_exec internal: index of data held by an object referenced
by an input object.
		ASSIGN_PTR=0x8D,	// r_exec internal: index of a hlp variable and to be assigned
index of an expression that produces the value.
		THIS=0x90,		// this pointer.
		VIEW=0x91,
		MKS=0x92,
		VWS=0x93,
		NODE=0xA0,
		DEVICE=0xA1,
		DEVICE_FUNCTION=0xA2,
		C_PTR =0xC0,		// chain pointer.
		SET=0xC1,
		S_SET=0xC2,		// structured set.
		OBJECT=0xC3,
		MARKER=0xC4,
		OPERATOR=0xC5,
		STRING=0xC6,
            TIMESTAMP=0xC7,
            GROUP=0xC8,
            INSTANTIATED_PROGRAM=0xC9,
            INSTANTIATED_CPP_PROGRAM=0xCA,
            INSTANTIATED_INPUT_LESS_PROGRAM=0xCB,
            INSTANTIATED_ANTI_PROGRAM=0xCC,
            COMPOSITE_STATE=0xCD,
            MODEL=0xCE
                  }Type;
                  // encoders
                  static     Atom     Float(float32 f); //       IEEE 754 32 bits
encoding; shifted by 1 to the right (loss of precison).
                  static     Atom     PlusInfinity();
                  static     Atom     MinusInfinity();
                  static     Atom     UndefinedFloat();
                  static     Atom     Nil();
                  static     Atom     Boolean(bool value);
                  static     Atom     UndefinedBoolean();
                  static     Atom     Wildcard(uint16   opcode=0x00);
                  static     Atom     TailWildcard();
                  static     Atom     IPointer(uint16 index);
                  static     Atom     RPointer(uint16 index);
                  static     Atom     VLPointer(uint16 index,uint16
       cast_opcode=0x0FFF);
                  static     Atom     IPGMPointer(uint16 index);
                  static     Atom     InObjPointer(uint8         inputIndex,uint16
index);           //         inputIndex: index of the input view; index: index of data in
the object's code.
                  static     Atom     DInObjPointer(uint8        relativeIndex,uint16
index);//         relativeIndex: index of an in-obj-ptr in the program's (patched) code;
index: index of data in the referenced object code.
                  static     Atom     OutObjPointer(uint16 index);
                  static     Atom     ValuePointer(uint16 index);
                  static     Atom     ProductionPointer(uint16 index);
                  static     Atom     AssignmentPointer(uint8    variable_index,uint16
index);
                  static     Atom     This();
                  static     Atom     View();
                  static     Atom     Mks();
                  static     Atom     Vws();
                  static     Atom     Node(uint8 nodeID);
                  static     Atom     UndefinedNode();
                  static     Atom     Device(uint8 nodeID,uint8 classID,uint8 devID);
                  static     Atom     UndefinedDevice();
                  static     Atom     DeviceFunction(uint16 opcode);
                  static     Atom     UndefinedDeviceFunction();
                  static     Atom     CPointer(uint8 elementCount);
                  static     Atom     SSet(uint16 opcode,uint8 elementCount);
                  static     Atom     Set(uint8 elementCount);
                  static     Atom     Object(uint16 opcode,uint8 arity);
                  static     Atom     Marker(uint16 opcode,uint8 arity);
                  static     Atom     Operator(uint16 opcode,uint8 arity);
                  static     Atom     String(uint8 characterCount);
                  static     Atom     UndefinedString();
                  static     Atom     Timestamp();
                  static     Atom     UndefinedTimestamp();
                  static     Atom     InstantiatedProgram(uint16 opcode,uint8 arity);
                  static     Atom     Group(uint16 opcode,uint8 arity);
                  static     Atom     InstantiatedCPPProgram(uint16 opcode,uint8 arity);
                  static     Atom     InstantiatedAntiProgram(uint16 opcode,uint8 arity);
                  static     Atom     InstantiatedInputLessProgram(uint16 opcode,uint8
arity);
```

```
                static      Atom      CompositeState(uint16 opcode,uint8 arity);
                static      Atom      Model(uint16 opcode,uint8 arity);

                static      Atom      RawPointer(void   *pointer);

                Atom(uint32  a=0xFFFFFFFF);
                ~Atom();

                Atom        &operator  =(const  Atom&    a);
                bool        operator   ==(const Atom&    a)        const;
        bool    operator    !=(const   Atom&    a)       const;
        bool    operator    !()        const;
        operator size_t     ()         const;
                uint32      atom;
                // decoders
        bool    isUndefined()          const;
        uint8   getDescriptor()        const;
        bool    isStructural()         const;
        bool    isFloat()              const;
        bool    readsAsNil() const;    // returns true for ll undefined values.
        float32 asFloat()              const;
        bool    asBoolean()            const;
        uint16  asIndex()              const;   // applicable to internal, view, reference,

        uint8   asInputIndex()         const;   // applicable to IN_OBJ_PTR.
        uint8   asRelativeIndex()      const;   // applicable to D_IN_OBJ_PTR.
        uint16  asOpcode()             const;
        uint8   asCastOpcode()         const;   // applicable to VL_PTR.
        uint8   getAtomCount()         const;   // arity of operators and




        uint8   getNodeID()  const;    // applicable to nodes and devices.
        uint8   getClassID() const;    // applicable to devices.
        uint8   getDeviceID()          const;   // applicable to devices.
        uint8   asAssignmentIndex()    const;

        template<class         C> C *asRawPointer() const{ return (C *)atom; }

        void    trace()     const;
        static  void        Trace(Atom *base,uint16      count);
        };
}
```

2.2 Encoding of Expressions

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 variables are encoded as value pointers;

VI. explicit references to objects (including markers that are explicitly referenced) are encoded as reference pointers;

VII. a reference to the view of an object is encoded as an internal pointer to the view. The view itself is encoded as the r-atom *view*, i.e. a place-holder where the executive will insert the correct view, depending on which group the reduction takes place;

VIII. references to the marker set and to the view set are encoded as internal pointers to, respectively, the r-atom *mks* and *vws* - this is similar to the encoding of the view (rule VII above).

IX. explicit references to object members are encoded as chain pointers, i.e. structures of r-atoms. The first element of a chain pointer is the r-atom for the chain, followed by a pointer to an object (*this*, value or reference pointer), and followed in turn by internal pointers to the members of the object.

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.

Explicit member access is also applicable to sets and structured sets: 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 Replicode atoms (i.e. 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 1.4 for more examples.

2.3 Examples

This section describes the encoding of views and objects for programs (pgm) and process objects (pro), assuming the declarations above.

pgm

The code segment of an object encoding a program contains the following r-atoms:

Index	*Content*
0	r-atom for pgm
1	internal pointer to tpl (12)
2	internal pointer to inputs (13+n)
3	internal pointer to prods (23+n+m+p+q)
4	internal pointer to tsc (25+n+m+p+q+r)
5	r-atom for csm
6	r-atom for sig
7	r-atom for nfr
8	r-atom for view
9	r-atom for mks
10	r-atom for vws
11	r-atom for psln_thr
12	r-atom for the tpl set (size: n)
13	internal pointer to first pattern in tpl
...	
12+n	internal pointer to last pattern in tpl
13+n	r-atom for the input set (size: 3)
14+n	internal pointer to input patterns
15+n	internal pointer to timings
16+n	internal pointer to guards
17+n	r-atom for the pattern set (size: m)
18+n	internal pointer to first pattern in inputs
...	
18+n+m	internal pointer to last pattern in inputs
19+n+m	r-atom for the timings set (size: p)
20+n+m	internal pointer to the first expression in timings
...	
20+n+m+p	internal pointer to the last expression in timings
21+n+m+p	r-atom for the guards set (size: q)
22+n+m+p	internal pointer to the first expression in guards
...	
22+n+m+p+q	internal pointer to the last expression in guards
23+n+m+p+q	r-atom for the prods set (size: r)
24+n+m+p+q	internal pointer to the first command
...	
24+n+m+p+q+r	internal pointer to the last command
25+n+m+p+q+r	r-atom for the tsc timestamp
26+n+m+p+q+r	most significant 32 bits of the timestamp
27+n+m+p+q+r	least significant 32 bits of the timestamp
...	tpl patterns
...	input patterns
...	timings

```
...            guards
...            productions
```

Notice that the program’s view, marker set and view set are not encoded in the code segment: they are encoded as place holders, to be valuated dynamically by the executive.

A segment of r-code encoding the local data for a program contains the following r-atoms:

Index	*Content*
0	r-atom for view
1	r-atom for ijt
2	r-atom for sln
3	r-atom for res
4	reference pointer to a group
5	reference pointer to a group, or r-atom for node
6	r-atom for act

The r-atom for the view refers to the Replicode class, i.e. is encoded as `0xC3xxxxxx`, not as the *view* r-atom (`0x9100000`).

The `reference` pointers are indexes in an array holding the actual addresses of the groups. These pointers are valuated by the executive.

3 Compiler

The compiler is composed of two C++ functions – `compile` and `decompile` -, all defined in a single shared object named `r_comp.so`.

These functions take as input/output arguments, snapshots of memory called *images*.

N.B.: the compiler is meant to be used in interactive mode, i.e. to encode in a scripting environment, and to decode for debugging purposes. Its performance is expected to be adequate to these uses.

3.1 Images

Images contain objects in r-code form, serialized from memory. Such images are mapped by the executive into memory to load the application. Images can be written to and read from disk, and can also be transmitted by the messaging interface.

An image contains four structures, arranged in a contiguous memory area – in the specified order:

- definition segment: an array containing class definitions - member names, offsets and types.
- object map: an array containing the indexes of objects in the code segment.
- code segment: an array holding the code of objects. Each object is stored as an array of r-atoms, followed by an array of pointers to other objects (the reference set), and last, by an array of views. Each view has its own code array and its own reference set.

The format for encoding the definition segment is private to the compiler and is not part of this specification.

```
...
!def (a_class ...)
...
P0:(pgm ...)
P1:(pgm ... P0 ...)
...
(inj [iP0:(ins P0 [...]) [iP0-view-data]]); inj0
(inj [iP1:(ins P1 [...]) [iP1-view-data]]); inj1
(inj [obj:(... iP1 ...) [obj-view-data]]); inj2
```

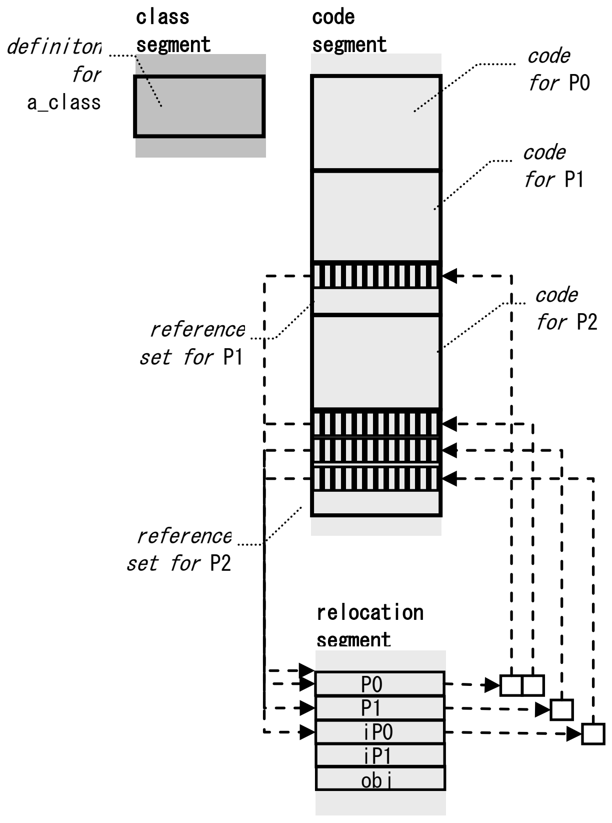

Figure 2 – Structure of an Image

Here is represented the structure if an image resulting from the compilation of a source code defining among other things, a class, two programs and three initial injections. Indexes are represented by dashed arrows.

The code for `P0` *does not reference any object: it has no reference set. The code of* `P1` *references* `P0`*, and thus, its reference set contains one index (vertical pattern) to the relocation segment corresponding to* `P0`*.*

Each entry in the relocation segment contains a list of the indexes where the executive shall substitute indexes (to the relocation segment) for actual addresses.

r-code defines the prototype of images as follows:

```
// file r_code.h

using namespace std;

namespace r_code{
```

```
class dll_export Image{
friend r_comp::Image;
private:
   uint32 def_size;
   uint32 map_size;
   uint32 code_size;
   word32 *data; // [def segment|object map|code segment]
   word32 *getDefSegment() const;
   uint32 getDefSegmentSize() const;
public:
   Image();
    ~Image();
   uint32 getSize() const; // size of data in word32
   uint32 getObjectCount() const;
   word32 *getObject(uint32 i) const; // points to the code size of
                                      // the object; the first atom
                                      // is at getObject()+2
   word32 *getCodeSegment() const; // equals getObject(0)
   uint32 getCodeSegmentSize() const;
   void read(ifstream &stream);
   void write(ofstream &stream);
   void trace() const;
};
}
```

The class segment is not part of the image specification: the compiler is free to use any subclass of `Image` appending the class segment to the existing structure. The only requirement is to keep the definition segment aligned on 4 bytes.

Reference pointers are included in the code segment.

An image contains the following:

- sizes of what follows
- def segment: list of class, operators and function definitions, and object names
- object map: list of indexes (4 bytes) of objects in the code segment
- code segment: list of objects
    - object:
        - size of the code (number of atoms)
        - size of the reference set (number of pointers)
        - size of the marker set (number of pointers)
        - size of the view set (number of views)
        - code: indexes in internal pointers are relative to the beginning of the object, indexes in reference pointers are relative to the beginning of the reference set
        - reference set:
            - number of pointers
            - pointers to the relocation segment, i.e. indexes of relocation entries
            - marker set:
                - number of pointers
                - pointers to the relocation segment, i.e. indexes of relocation entries
            - view set: list of views
                - view:
                    - size of the code (number of atoms)
                    - size of the reference set (number of pointers)
                    - list of atoms

- reference set:
    - pointers to the relocation segment, i.e. indexes of relocation entries

RAM layout of Image::data:

data[0]:	first word32 of def segment
...	...
data[def_size-1]:	last word32 of def segment
data[def_size]:	index of first object in data
...	...
data[def_size+map_size-1]:	index of last object in data
data[def_size+map_size]:	first word32 of code segment
...	...
data[def_size+map_size+code_size-1]:	last word32 of code segment

An image defines everything needed to instantiate objects and groups in given node. In this version of the specification, as many images as there are nodes in a system shall be generated by the compiler (i.e. one shall compile as many source files as there are nodes).

3.2 COMPILE

The `compile` function compiles code in the Replicode form into the r-code form. The function has two overloads.

Signatures

```
void compile(ifstream &stream,
             Image    *&image,
             char     *&error_msg);
```

```
void compile(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.

The function is supposed to output an error message at the first syntax error it encounters, and stop compiling at that point in the stream. In that case, no image is produced.

Operator return types and class member types are used by the function to enforce minimal static type checking.

3.3 DECOMPILE

The `decompile` function decompiles code in the r-code form into the Replicode form.

Signature

```
void decompile(Image         *&image,
               ostringstream stream);
```

Inputs

`image`: an image in memory containing r-code objects.

Outputs

`stream`: an output character stream.

In the source code obtained from `decompile`, 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 the `!class`, `!dev` and `!dfn` directives;
- commands in productions shall be replaced - when possible - using the aliases defined for the functions of the executive;
- literals representing constants in std.replicode shall replace their counterparts in r-code;
- references to the groups *root*, *stdin* and *stdout* and references to the entity *self* shall be replaced by the symbols `root`, `stdin`, `stdout` 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.

4 Communication With External Modules

External modules are user-defined C++ code that might need to communicate with the Replicode application. To do so, they would send or receive messages carrying Replicode objects, encoded in r-code. The messaging infrastructure is not part of the specification.

External 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, external modules shall use the r-view and r-object classes as the carriers of their I/O, that is, modules shall encode/decode information into/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 implementing the external modules.

The executive is specified independently of the messaging interface. The executive is meant to be wrapped into a module that complies with the messaging interface. This wrapper module is responsible for:

- having the executive load images when received from the network: it shall implement a function like:

```
void batch_receive(){
   <for each group in the image>
      Object* o = Object::create(<arguments>)
      Mem::receive(o, <other arguments>)
      Group* g = o->asGroup()
   <for each object in the image>
      Object* o = Object::create(<arguments>)
      Mem::receive(o, <other arguments>)
}
```

- notifying the rMem of incoming messages carrying r-code: this translates into a call to Mem::receive();
- sending objects to remote nodes upon the execution of ejection commands by the rMem.
- controlling the rMem, upon reception of commands *start*, *stop*, *pause* and *resume*.

5 Dependencies

This section summarizes the dependencies between the various components defining an application.

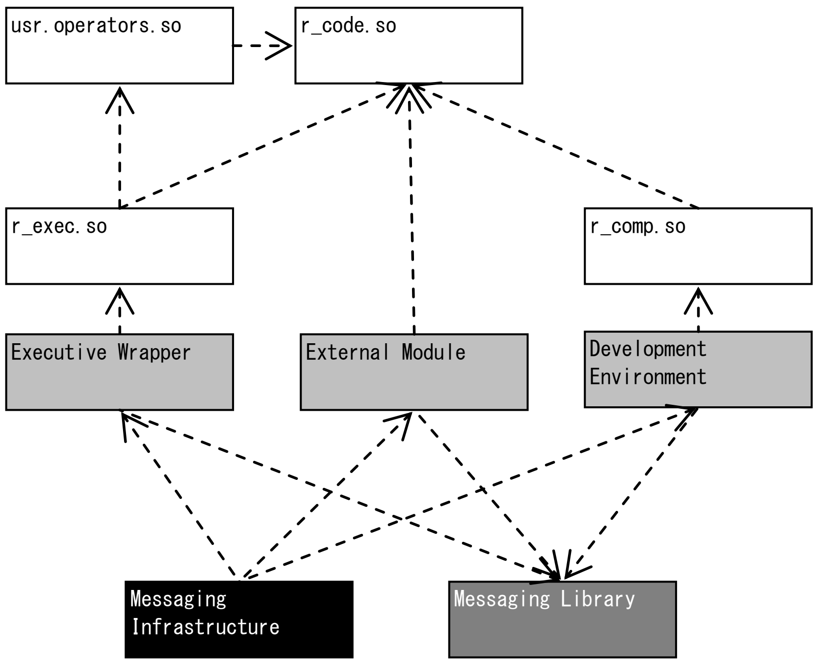

Figure 3 - Dependencies

Dependencies are indicated by dashed lines: they correspond to the loading of the pointed objects at runtime.

The executive is wrapped into a module that complies with the messaging infrastructure (see Annex 2).

`usr.operators.so` *is shared object that implements user-defined operators (see section 2.1. of the Annex 4 for details).*

Annex 4 – Replicode Extension API

1 Overview

Replicode allows programmers extending the language with:

- custom operators,
- custom object classes,
- custom marker classes.

In addition, programmers also have to define and implement:

- custom devices,
- custom device functions.

Custom object or marker classes do not require any API, as they are to be declared as preprocessor directives (see section 3.1).

Devices are external modules and their specification has to comply with the requirements of the messaging infrastructure. Thus, Replicode does not include an API to define these devices (see section 3.2).

For the same reason, the implementation of device functions does not depend on Replicode (see section 3.3).

To summarize, the purpose of the Replicode Extension API is to allow programmers extending Replicode with custom operators.

2 Specification

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 std;

namespace r_code{

class dll_export Expression{
friend class ReductionInstance;
protected:
   ReductionInstance *instance;
   int32 index;
   bool isValue;
public:
   RExpression(ReductionInstance *instance_,
               int32             index_,
               bool              isValue_);
   Atom& head() const;
   Expression child(int index) const;
   int64 decodeTimestamp() const;
   void setValueAddressing(bool isValue_);
   bool getValueAddressing() const:
   int32 getIndex() const;
   Expression dereference() const;
   Expression copy() const;
   Expression copy(ReductionInstance* dest) const;
   Atom iptr() const;
   Atom vptr() const;
```

```
};

class dll_export ExecutionContext:
public Expression{
private:
   vector<Atom> resultSet;
public:
   ExecutionContext(RExpression& e);
   Expression evaluateOperand(int index);
   Expression evaluate();
   ExecutionContext xchild(int index) const;

   void setResult(Atom result);
   void setResultTimestamp(int64 timestamp);
   void beginResultSet();
   void appendResultSetElement(Atom element);
   void endResultSet();
   void undefinedResult();
};

struct dll_export Group{
   virtual void receive(ReductionInstance *reduction)=0;
   virtual Expression copyContent(ReductionInstance& dest) const=0;
};

class dll_export Object{
public:
   static Object* create(
      vector<Atom> atoms,
      vector<const Object*> references
   );

   virtual void retain()=0;
   virtual void release()=0;
   virtual Expression copy(ReductionInstance& dest) const=0;
   virtual Expression copyMarkerSet(
                       ReductionInstance& dest) const=0;
   virtual Expression copyViewSet(
                       ReductionInstance& dest) const=0;
   virtual Expression copyVisibleView(
                       ReductionInstance& dest,
                       const RGroup* group) const=0;
   virtual Expression copyLocalView(
                       ReductionInstance& dest,
                       const RGroup* group) const=0;
   virtual Object* getReference(int index) const=0;
   virtual const Group* asGroup() const=0;
};

class dll_export ReductionInstance{
friend class Expression;
friend class ExecutionContext;
private:
   struct CopiedObject {
      Object *object;
      int32   position;
   };
   int32 referenceCount;
   vector<Atom> input;
   vector<Atom> value;
   vector<CopiedObject> references;
   size_t hash_value;
```

```
public:
   ReductionInstance();
   void retain();
   void release();

   struct ptr_hash {
      size_t operator()(const ReductionInstance *ri) const;
   };

   ReductionInstance* reduce(Object *input);
   ReductionInstance* reduce(vector<ReductionInstance*> inputs);
   void merge(Expression location, ReductionInstance* ri);
};

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.

User-defined operator opcodes are to be specified in the file `usr.operators.replicode`. The implementation of user-defined operators is to be provided in a shared object called `usr.operators.so`. Said shared object shall expose a function having the following prototype:

```
extern "C"{
   uint32 Load();
}
```

This function is to be called by the executive upon loading the object. Load is supposed to:

- first, perform initialization tasks, shall the operators need some,
- then, register the operators in the OperatorRegister.

OperatorRegister is defined in the shared object `r_core.so`.

Here is given a sample implementation for the operator `add` – in this code, the admissible inputs are restricted to floats and timestamps:

```
void operator_add(ExecutionContext& context){

   Expression lhs(context.evaluateOperand(1));
   Expression rhs(context.evaluateOperand(2));
   if(lhs.head().isFloat()){

      if(rhs.head().isFloat()){

         context.setResult(
            RAtom::Float(
               rhs.head().asFloat()+lhs.head().asFloat()
```

```
                )
            );
            return;
        }else if(rhs.head().getDescriptor()==Atom::TIMESTAMP){

            context.setResultTimestamp(
                int(lhs.head().asFloat()*1e6)+rhs.decodeTimestamp()
            );
            return;
        }
    }else if(lhs.head().getDescriptor()==Atom::TIMESTAMP){

        if(rhs.head().isFloat()){

            context.setResultTimestamp(
                lhs.decodeTimestamp()+int(rhs.head().asFloat()*1e6)
            );
            return;
        }
    }
    context.setResult(Atom::Nil());
}
```

3 Other Extensions

3.1 Object and Marker Classes

Object and marker classes in Replicode are mapped to instances of `RObject`: no particular C++ class is required to support user-defined object classes.

The definitions of the r-atoms that correspond to user-defined classes (opcodes) are to be found in the source code as preprocessor `!class` directives (file `usr.objects.replicode`).

3.2 Devices

Devices are external modules and as such, depend on the specification of the messaging infrastructure.

In r-code, devices are r-atoms whose structure is:

Descriptor	*Data*
`A2`	[*node-identifier*(8) | *class-identifier*(8) | *instance-identifier*(8)]

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, in accordance with the messaging infrastructure.

3.3 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*
`A3`	[*unused*(8) | *r-opcode*(8)]

Device functions shall be defined in the source code, using preprocessor `!def` directives (file `usr.device.functions.replicode`).