Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

The Lucky language Reference Manual

Erwan Jahier, Pascal Raymond

Verimag Research Report n^o TR-2004-6

Initial version: March 12, 2004
Last update: June 8, 2006

Reports are downloadable at the following address
http://www-verimag.imag.fr

The Lucky language Reference Manual

Erwan Jahier, Pascal Raymond

Initial version: March 12, 2004
Last update: June 8, 2006

Abstract

The main challenge to automate a testing or a simulation process is to be able to automate the generation of *realistic* input sequences to feed the program. In other words, we need an executable model of the program environment which inputs are the program outputs, and which outputs are the program inputs.

In the first Lurette prototype – an automated testing tool of reactive programs designed at Verimag –, the System Under Test (SUT) environment behaviour was described by Lustre observers which were specifying what realistic SUT inputs should be. In other words, the environment was modelled by a set of (linear) constraints over Boolean and numeric variables. The work of Lurette was to solve those constraints, and to draw a value among the solutions to produce one SUT input vector.

But, from a language expressing power point of view, Lustre observers happen to be too restrictive, in particular to express sequences of different testing scenarios, or to have some control on the probabilistic distribution of the drawn solution. It was precisely to overcome those limitations that a new language, Lucky, was designed.

A Lucky program is an interpreted automaton whose transitions define the reactions of the machine. More precisely, each transition is associated to (1) a set of constraints (a lustre-like formula) that defines the set of the possible outputs, and (2) a weight that defines the relative probability for each transitions to be taken, i.e., to be used to produce the output vector for the current step.

Keywords: Reactive systems, validation, automatic test case generation, lurette, lustre

Reviewers: Nicolas Halbwachs

Notes:

How to cite this report:

@techreport { ,
title = { The Lucky language Reference Manual},
authors = { Erwan Jahier, Pascal Raymond},
institution = { Verimag Research Report },
number = {TR-2004-6},
year = { },
note = { }
}

Contents

1 Introduction **2**

2 The Lucky language principles **2**
2.1 Overview 2
2.2 Definitions 4
2.3 Operational Semantics 5
2.4 A Lucky program Example 6

3 The Lucky concrete syntax **8**
3.1 Identifiers, Blanks and Comments 8
3.2 Boolean, Integer and Floating-point literals 8
3.3 Type Definitions and Type Expressions 8
3.4 Variable declarations 8
3.5 Dynamic libraries and external functions declarations 9
3.6 Expressions 9
3.7 Control state declarations 10
3.8 Transitions 10
3.9 Lucky files 10
3.10 Pragmas 10

4 Calling external code **11**
4.1 Current restrictions 11
4.2 External code with side-effects 11

5 Numeric solver issues **12**
5.1 Solving integer constraints in dimension $n \geq 2$ 12
5.2 Fairness versus efficiency 12
5.3 Fair mode and precision and the computations 13

6 The Lucky file interpreter **14**

A More Lucky program Examples **16**
A.1 The Lucky program of Figure 2 16
A.2 Illustration of the use of Polyhedra 18
A.3 Illustration of the use of infinite weights 19
A.4 Illustration of the use of dynamic weights 20
A.5 Illustration of the use of structured types 21
A.6 Illustration of the use of external function calls 23

B Automata Product **24**

1 Introduction

Motivations. Synchronous programs [BG92a, HCRP91, LBBG86] deterministically produce outputs from input values. To be able to compile, synchronous programs need to be fully deterministic. However, sometimes, we want to be able to describe synchronous systems in a non deterministic manner:

- if one wants to describe (and simulate) an intrinsically non-deterministic system. A typical example is when one want to describe the environment of a reactive program; it can be very useful for testing and simulation purposes.
- Another potential use of the animation of non-deterministic code is when one wants to simulate partially written reactive programs (some components are missing). The idea is then to take advantage of program signatures, pre/post conditions, or code chunks to simulate those programs the more realistically as possible, taking into account the available constraints, and drawing the non-deterministic parts. This can be very useful to simulate and test applications at every stage of the development process.

We call an *non-deterministic program* such pieces of code that produce their outputs non-deterministically. Lucky is a language to describe such non-deterministic programs. Basically, it is an interpreted automaton which transitions are labelled by Boolean expressions. As in Lustre, expressions are formulas over Boolean and numeric values[1] that define an atomic reaction (in the synchronous sense). The key difference with lustre is that those equations may admit several solutions, hence the first source of non-determinism. The second source of non-determinism is due to the fact that several expressions can be reached from the current state; the choice is done according to weights that also label transitions. Those weights let one assign probabilities to the different temporal scenarios defined by the automaton. Lucky can be seen as a language to program stochastic processes (Markov chains).

Plan. We first present the Lucky language principles in Section 2, and give its syntax rules in Section 3. Then, we briefly present the `lucky` command-line tool in Section 6. Finally, In Appendix, we provide several commented Lucky programs.

2 The Lucky language principles

A reactive system is an automated system that indefinitely responds to its environment. We are particularly interested here in control and embedded applications, where the environment is often the physical world. During the development of such systems, non-determinism is often useful, for describing a partially designed system and/or its environment.

2.1 Overview

We propose a model where a basic qualitative model describing a set of behaviours is extended with a probabilistic mechanism. The main features of this model are presented here.

Symbolic state/transition systems. The basic qualitative model consists in a very general state/transition system, characterised by:

- a memory: a finite set of variables with no special restrictions on their domains (to simplify, we will consider here just Boolean, integer and rational values);
- an interface: variables are declared as inputs, outputs, or locals;
- a finite control structure: an interpreted finite automaton, whose transitions are representing reactions of the machine.

[1] A restriction is that numeric constraint ought to be linear (e.g., $x + y > 3$, but not $x^2 + y^2 > 2$ nor $log\ s + sin\ r > e^s$).

A global state of the system is then a pair made of the current control point in the automaton (the *control-state*), and a current valuation of its memory (the *data-state*). Therefore the set of global states is potentially infinite.

Synchronous relations. We adopt the synchronous approach for the reactions: all values in the memory are changing simultaneously when a reaction is performed. The previous value of the memory corresponds to the source data-state, and the current value to the next data-state. The transitions are labelled with information denoting what are the possible values of the current memory depending on the current data-state. This information is quite general: it is a *relation* between the past and current values of the variables. In particular, no distinction is made between uncontrollable (inputs and past values) and controllable (locals and outputs) variables. Performing a reaction will consist in finding solutions to such a formula. This problem induces a restriction: we suppose that, once reduced according to the past and input values, the constraints are solvable by some actual procedure[2].

Weights instead of probabilistic distribution. Since we have to deal with uncontrollable variables, defining a sound notion of distribution must be done carefully: depending on those variables, a formula may be infeasible, and thus its actual probability is zero. In other terms, if we want to use probabilistic distributions, we would have to define a reaction as a map from the tuple ⟨source state, past values, input values⟩ to a distribution over the pairs ⟨controllable values, next state⟩. Expressing and exploiting this kind of model would be too complex. We prefer a pragmatic approach where probabilities are introduced in a more symbolic way.

The main idea is to keep the distinction between the probabilistic information and the constraint information. Since constraints are influencing probabilities (zero or non-zero), this information does not express the probability to be drawn, but pragmatically, the probability to be *tried*. In order to emphasise the difference, we do not use distributions (i.e., set of positive values the sum of which is 1) but *relative weights*. A relative weight is a positive rational value, not necessarily less than one, the meaning of which is only defined relatively to another weight: if two possible reactions (i.e., the corresponding constraints are both satisfiable) are labelled respectively with the weights w and w', then the probability to perform the former is w/w' times the probability to perform the latter.

Static weights versus dynamic weights. The simplest solution is to define weights as constants, but in this case, the expressive power can be too weak. With such static weights, the uncontrollable variables qualitatively influence the probabilities (zero or not, depending on the constraints) but not quantitatively: the idea is then to define *dynamic weights* as numerical functions of the inputs and the past-values. Taking numerical past-values into account can be particularly useful. A good example is when simulating an *alive process* where the system has a known average life expectancy before breaking down; at each reaction, the probability to work properly depends *numerically* on an internal counter of the process age.

Transient states. For the time being, we have only one notion of state: a state is a stable control point, and a transition between two states defines an atomic reaction. However, we think it may be convenient to introduce the notion of *transient state*, and, as a consequence a notion of micro-step: a complete reaction is then a sequence of transitions between two stable states, where all the intermediate states are transient. Transient states do not affect the synchronous interpretation of the variable changes: intuitively, if we abstract probabilities, a reaction $q \xrightarrow{f} t \xrightarrow{g} q'$, is qualitatively equivalent to $q \xrightarrow{f \wedge g}$. In contrast, transient states affect probabilities, and may be helpful to express complex conditional relative weights.

Global concurrency. Concurrency (i.e., parallel execution) is a central paradigm for reactive systems. The problem of merging sequential and parallel constructs has been largely studied: classical solutions are hierarchical automata "à la StateCharts" [Mar92, And96], or statement-based languages like Esterel [BG92b]. Our opinion is that deeply merging sequence and parallelism is a problem of high-level language, and that it is sufficient to have a notion of global parallelism: intuitively, local parallelism can always be made

[2] concretely, we have developed a constraint solver for mixed Boolean/linear constraints.

global by adding extra idle states. As a consequence, concurrency is a top level notion in our model: a complete system is a set of concurrent automata, each one producing its own constraints on the resulting global behaviour.

Weights and parallelism. In terms of control structures, parallelism corresponds to a kind of synchronous product of automata. Transient states make this "product" more complex than a simple Cartesian product, but do not involve big difficulties. For formulas, the product is simply the logical "*and*". Unfortunately, there is no obvious way for combining stochastic information: as they are defined, they are only local information and they may induce paradoxes when combined into a parallel composition.

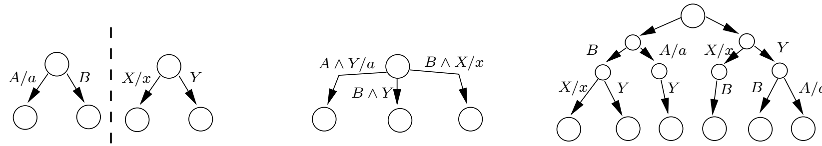


Figure 1: Weights and parallelism: the parallel composition (1a), and, assuming that $A \wedge X$ is infeasible, the product solution (1b) and the arbiter solution (1c).

A simple example is shown in Figure 1a: the first automaton (resp. the second) has the choice between the constraints A or B (resp. X or Y) both satisfiable. In the first automaton, the choice of A has a big weight $a >> 1$ compared to B (1 by default), and in the second one, X has a big weight $x >> 1$ comparing to Y. Suppose that the data-state makes it impossible to satisfy $A \wedge X$, it follows that it is impossible to satisfy the stochastic demand of both components. There are mainly two ways to solve the problem:

- Consider that weights are not only local information, but are also influencing the parallel composition: for instance, if a is much bigger than x, that means that the stochastic demand of the first component is much stronger than the one of the second. The simplest way to implement this notion is to combine weights with multiplication, as shown in Figure 1b.

- The problem is treated at the parallel composition level, where some indications are added to express priority for satisfying stochastic demands. Intuitively, the components of a parallel composition are treated sequentially: the first one is perfectly served, according to its own local weights, then the second is served according to what was decided by the first one, etc. The order of components is, in general non-deterministic, and stochastic information may be added to influence it. The Figure 1c shows a product where a first fair choice is made to decide which component will "play" first (note that all intermediate states are transient).

There is no obvious argument to prefer one solution to another: both are consistent, and none is clearly more natural than the other. As a consequence, we plan to implement both and let the user choose between them. For the time being, only the first one is implemented.

Those two product algorithms are described into more details in Appendix B.

2.2 Definitions

Variables. Lucky programs variables are either *input*, *output*, *local*, or *previous*: $V = V_i \uplus V_o \uplus V_l \uplus V_p$. The previous variables are meant to refer to previous values of the other variables in $V_{\backslash p} = V_i \uplus V_o \uplus V_l$. Each previous variable is denoted by $\bullet v$ where $v \in V_{\backslash p}$. Moreover, each variable in $V_{\backslash p}$ is defined with a default value: the value just before the first reaction. Local variables can be seen as output variables that are hidden from the outside.

Valuations. A *valuation* is a mapping from variables to values. A *data-context* is a pair (σ_i, σ_p), where σ_i (input valuation) associates a value with each input, and σ_p (previous valuation) associates a value with each previous variable. Previous valuations are also called *data-state*. In particular, the default values of variables are defining the *initial data-state*, denoted by σ_p^0.

Formula. A formula is any well-typed Boolean expression made of variables, constants, and classic logical and numerical operators ($\neg,\wedge,\vee,=,>,<,\geq,\leq,+,-,*,/$). We note $\mathcal{F}$ the set of well-typed formula.

Control structure. At the top level, the behaviour of a system is described by a non-empty set of concurrent Lucky programs sharing the same variables. Each Lucky programs is an interpreted automaton, where transitions are labelled by qualitative and stochastic constraints, as presented in the sequel.

control states. The set of *control-states*, is divided into *stable* states and *transient* states: $Q = Q_s \uplus Q_t$. The initial control state is a particular stable state $q^0 \in Q_s$.

Weights. Weights are positive numerical functions of the uncontrollable variables: $\mathcal{W} : \Sigma_i \times \Sigma_p \rightarrow \mathbb{N} \cup \{\infty\}$. More concretely, they are given as numerical expressions made of inputs, previous variables, classical operators or predefined computable functions. The ∞ value is introduced to express a sound notion of mandatory choice: a transition with the infinite weight has priority on any finite weighted transition. There is a single notion of infinite weight: two feasible transitions with infinite weight have the same probability. In order to express relative probabilities between mandatory choices, it is necessary to detail the control structure by introducing transient states.

Transitions. The set of transitions is a relation: $T \subseteq Q \times \mathcal{F} \times \mathcal{W} \times Q$, and we note $q \xrightarrow[w]{f} q' \in T$ a transition from q to q' labelled by the formula f and the weight w.

Note that 2 transitions can have the same origin and the same target. Note also that at most one outgoing transition can have an infinite weight – which not really a restriction since an equivalent automaton can be written using transient states, eg, $\{q_1 \xrightarrow[\infty]{f} q_2, q_1 \xrightarrow[\infty]{f} q_2, ...\} \equiv \{q_1 \xrightarrow[\infty]{f} q', q' \xrightarrow[w_1]{f} q_2, q' \xrightarrow[w_2]{f} q_2, ...\}$.

Transitional loops. We do not try to give sense to infinite loops of transient control states: models containing such combinational loops are statically rejected.

2.3 Operational Semantics

The Lucky programs are defined in such a way that their operational semantics is straightforward. In some sense, they are *executable* by definition. We give here the main lines of the simulation algorithm.

In the sequel, we suppose that we have a single automaton, obtained with one of the product operations defined above. Note that, in the concrete implementation, the global product is not statically built: local products are simply build *on the fly* to avoid space state explosion.

Execution. A *Lucky program execution*, according to a given input history $(\nu^n)_{n\geq 0}$ is a sequence of pairs made of a stable state and a valuation: $(s^n, \sigma^n)_{n\geq 0} \in \mathbb{N} \rightarrow Q_s \times \Sigma$, such that:

- s^0 is the initial control state and σ_p^0 is the default map of the variables,

and for each k:

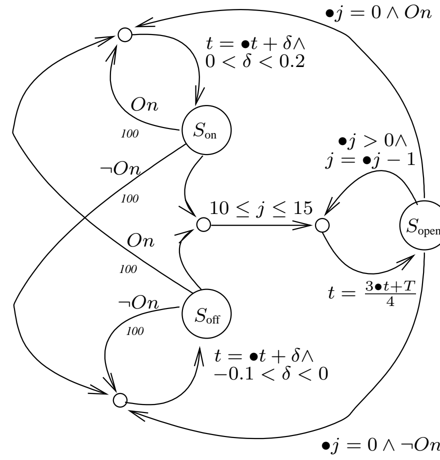
- $\sigma_i^k = \nu^k$ (the valuation history meets the input history)
- $\sigma_{\setminus p}^k = \sigma_p^{k+1}$ (the current-part of the valuation meets the previous-part of the next valuation)
- $(s^k, \sigma_p^k, \sigma_i^k) \xrightarrow{\sigma_o^k, \sigma_l^k} s^{k+1}$ is a feasible, fair reaction according to the control structure. We detail in the sequel the algorithm for finding such a reaction.

Reaction. Intuitively, a step in an execution is done by drawing, according to weight directives and current values of uncontrollable variables, a path in the automaton from the current (stable) state to a next stable state. More formally, let $s = q_0$ be the current control-state, σ_p the current data-state and σ_i the current inputs. For all k, we note $\Theta_k = \{q_k \xrightarrow[w]{f} q\}$ the set of transitions leaving q_k, and we use the notation w_τ to denote the weight attached to a transition τ. According to the current data context, the sum of weights $W_k = \sum_{\tau \in \Theta_k} w_\tau(\sigma_p, \sigma_i)$ is a numerical constant. If there exist some transition τ from q_k to q_{k+1}, the probability to complete a path $(q_0, q_1, ..., q_k)$ with q_{k+1} is then: $w_\tau(\sigma_p, \sigma_i)/W_k$. This process is repeated until a stable state $q_n = s'$ is reached: $s \xrightarrow[w_1]{f_1} q_1 \xrightarrow[w_2]{f_2} q_2 \cdots q_{n-1} \xrightarrow[w_n]{f_n} s'$, where all $q_1, \cdots q_{n-1}$ are transient.

The conjunction of all formulas labelling the drawn path is the elected formula: $f = \bigwedge_{k=1}^{n} f_k$. We substitute in f input and previous variables ($f_{|\sigma_i \sigma_p}$), and solve it. A valuation of output and local variables for the current step is obtained by performing a fair toss among the solutions of that formula. If $f_{|\sigma_i \sigma_p}$ is unsatisfiable, another path is drawn. If no satisfiable path can be drawn, the machine stops.

Note that infeasible paths are detected as soon as possible (by marking them) to avoid divergence.

2.4 A Lucky program Example



$V_i = \{On, T\}$, $V_o = \{t\}$, $V_l = \{\delta, j\}$, $V_p = \{\bullet t, \bullet j\}$, $Q_s = \{S_{on}, S_{off}, S_{open}\}$, $q_0 = S_{on}$.

Figure 2: A Lucky program that simulates the temperature in a room with a heater and a window.

The automaton of Figure 2 represents a Lucky program that models the temperature in a room containing a heater and a window which is opened from times to times. The version of this automaton given in Lucky concrete syntax is provided in Section A.1. Input variables are a Boolean On, which is true if the heater in on, and a real T, which indicates the temperature outside the room. The only output variable is a real t, indicating the temperature inside the room. Local variables are the real δ, which is used to compute the new temperature, and the integer j, which is used to count the number of steps the window remains open. Previous variables are $\bullet t$ and $\bullet j$. Stable states are denoted by s_{on}, s_{off}, and s_{open}. The other unnamed states are transient. The initial state is s_{on}.

If the heater is Initially on (resp. off), only 2 transitions are possible among the 3 output transitions of s_{on}, since the transition labelled by $\neg On$ (resp. On) is unsatisfiable. The first possible transition has weight 100, and the other one has weight 1.

- The first transition will therefore be drawn with a probability of $100/101$. It leads to a transient state which has only one output transition leading back to s_{on} (resp. s_{off}); the elected formula is therefore $0 < \delta < 0.2 \wedge t = \bullet t + \delta$. It states that the local variable δ will be uniformly drawn between 0 and 0.2 (resp. -0.1 and 0), and that δ is then used to increase the temperature. This is intended to model that, when the heater is on (resp. off), the temperature slightly increases (resp. decreases).

- The second transition will be drawn with a probability of $1/101$. After two transient states, the only stable state that is reachable is s_{open}, and the elected formula is therefore $true \wedge 10 \leq j \leq 15 \wedge t = \frac{3 \bullet t + T}{4}$. It states that the local integer variable j is drawn uniformly between 10 and 15, and defines how the new temperature is computed. This is intended to model that, whenever the window is open, the temperature becomes closer from the temperature outside.

  For the next step, from s_{open}, 3 transitions are possible, but they are labelled by formulas that can not be true at the same time (they form a partition): as long as j is greater than 0, the window will remain open; j is decremented at each step, and when it reaches 0, the control gets back to either s_{on} or s_{off}, depending on the variable On.

The concrete syntax version of this Lucky automata is given in Section A.1.

3 The Lucky concrete syntax

The syntax rules are given in an extended-BNF-like notation, where the meta-symbols '→ ', '⟨', '⟩','?','*', and '+' have the usual meaning. Non-terminals are in bold between brackets (**<like this>**) and terminals are in typewriter font (`like that`).

3.1 Identifiers, Blanks and Comments

```
<ident>  →  ⟨<letter> | _ ⟩ ⟨ <letter> | 0...9 | _ | ' ⟩*
<letter> →  A ... Z | a ... z
```

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage return, line feed and form feed. Blanks are ignored, but they separate adjacent identifiers, and literals. Single line comments are introduced by the two characters `"--"`. Multi-line comments are introduced by the two characters `"(*"`, and terminated by the characters `"*)"`. Comments cannot occur inside string or character literals, and can be nested.

3.2 Boolean, Integer and Floating-point literals

```
<bool>  →  true | false
<int>   →  ⟨-⟩? ⟨ 0...9 ⟩+
<float> →  ⟨-⟩? ⟨ 0...9 ⟩+ ⟨. ⟨0...9⟩* ⟩? ⟨⟨e|E⟩ ⟨+|-⟩? ⟨ 0...9 ⟩+⟩?
```

Lucky integers are coded on 31 bits, which means that they range from -2^{30} to $2^{30}-1$. Float are double-precision floating-point numbers conforming to the IEEE 754 standard, with 53 bits of mantissa and an exponent ranging from -1022 to 1023.

todo: *Add* `--int32` *and* `--int64` *options.*

3.3 Type Definitions and Type Expressions

```
<type_def>   →  <ident> = <type_exp>

<type_exp>   →  <ident> | <basic_type> | <array> | <struct> | <enum>
<basic_type> →  bool | int | float[3]
<array>      →  <type_exp> ^ <int>
<enum>       →  ( <ident> ⟨, <ident>⟩* )
<struct>     →  { <ident> : <type_exp> ⟨; <ident> : <type_exp> ⟩* }
```

The syntax for type expression is the same as the one of Scade and Lustre. Examples of type definitions are:

```
ex_array = bool ^ 3;                -- A Boolean Array of size 3
ex_struct = { a:int; b:ex_array }; -- A structure with 2 fields
ex_enum = ( red, white, blue );     -- An enum with 3 values
```

3.4 Variable declarations

```
<var_decl> →  <ident>: <type_exp> ⟨ <flag> <exp> ⟩*
<flag>     →  ~min | ~max | ~default | ~alias | ~init
```

[3]or alternatively, one can use `real` instead of `float`

Variable declaration <**flag**>s of course only make sense for output and local variables. The `min` flag (resp `max`) lets one define a global lower bound (resp upper) to a variable. The Default values for min and max are -10000 and 10000 respectively. The flag `default` lets one assign a default value to variables that have not been constrained. The flag `alias` states that the variable is an alias for an expression <**exp**>; alias variables are inlined. The flag `init` lets one assign an initial value to variables so that a `pre` on that variable can be done at the first instant – otherwise, an error is raised at runtime.

3.5 Dynamic libraries and external functions declarations

Lucky expressions can use external functions defined in dynamic librairies. To do that, one first need to declare the name and types of external functions to be used, as well as the name of the dynamic libraries into which those functions are defined. The dynamic library should be build according to certain conventions that are described in Section 4.

<**library_decl**> → <**ident**> `=` <**string**>

A library declaration associates a library identifier to a library file name.

<**function_decl**> → <**ident**> `:` <**func_type**> `:` <**ident**>
<**func_type**> → <**type**> ⟨ `*` <**type**> ⟩* `->` <**type**>

A function declaration is made of a function identifier, a function type, and a (declared!) library identifier.

3.6 Expressions

<**exp**> → `(` <**exp**> `)`
| <**int**> | <**float**> | <**bool**> | ⟨ `pre` ⟩* <**ident**>
| <**op1**> <**exp**> | <**exp**> <**op2**> <**exp**>
| `if` <**exp**> `then` <**exp**> `else` <**exp**>
| <**array_access**> | <**struct_access**>
| <**array_exp**> | <**struct_exp**>
| <**ext_func_expr**>

<**ext_func_expr**> → <**ident**>`(`<**exp**> ⟨`,` <**exp**>⟩*`)`

<**op1**> → `not` | `-`
<**op2**> → `and` | `or` | `nor` | `xor` | `=` | `=>` | `<>` | `<` | `>` | `<=` | `>=`
| `+` | `-` | `*` | `/` | `mod` | `div` | `abs`

<**array_access**> → <**exp**>`[`<**int**>`]`
<**struct_access**> → <**exp**>`.`<**ident**>
<**array_exp**> → `[` <**exp**> ⟨`;` <**exp**> ⟩* `]` | <**exp**> `^` <**int**>
<**struct_exp**> → `{` <**ident**> `=` <**exp**> ⟨`;` <**ident**> `=` <**exp**> ⟩* `}`

The syntax for lucky expresssions is basically the same as the one of lustre expressions. One difference is that (for the time being) `pre` are only allowed over identifiers. The typing rules for lucky expressions are also the Lustre ones.

Once inputs and memories are replaced by theirs values, all constraints should be linear, and all external function expression arguments should be bound. Note that constraint `x mod y` is not linear if `y` is unbound (i.e., if it is a controllable variable). The same holds for the `/` and `div` operators.

3.7 Control state declarations

```
<states_decl> → <states> ⟨, <states> ⟩* : ⟨ <state_mode> ⟩
<state_mode> → transient | stable | final
<states> → <ident>
```

`final` states are a special kind of `stable` states: if the execution stops on a final state, the interpreter exits normally; otherwise it returns an error code and an error message.

3.8 Transitions

```
<transition> → <states> -> <states> ⟨<transition_flag>⟩*
<transition_flag> → ~weigth <weight> | ~cond <exp>
<weight> → <int> | infinity | ⟨ pre ⟩* <ident>
```

<weight> can either be integers or variable identifiers that ought to be bound to some integer value. The default weigth is 1. The default constraint is `true`.

Note that transition weigth can be infinite to express mandatory choice (namely, if the transition is possible, take it). But two transitions outgoing from the same state can not have an infinite weigth. One need to use an intermediary transient state to achieve the same effect. The advantage of enforcing such a configuration is that it enforces one to precisely specify the relative probability between the two mandatory choices.

3.9 Lucky files

```
<lucky_file> →
        ⟨ libraries { ⟨ <library_decl> ; ⟩* <library_decl> ⟨;⟩? } ⟩?
        ⟨ functions { ⟨ <function_decl> ; ⟩* <function_decl> ⟨;⟩?} ⟩?
        ⟨ typedef { ⟨ <type_def> ; ⟩* <type_def> ⟨;⟩?} ⟩?

          inputs  { ⟨ <var_decl> ; ⟩* <var_decl> ⟨;⟩? }
          outputs { ⟨ <var_decl> ; ⟩* <var_decl> ⟨;⟩? }
          locals  { ⟨ <var_decl> ; ⟩* <var_decl> ⟨;⟩? }

          states      { ⟨ <states_decl> ; ⟩* <states_decl> ⟨;⟩? }
          start_state { ⟨ <states> ; ⟩* <states> ⟨;⟩? }
          transitions { ⟨ <transition> ; ⟩* <transition> ⟨;⟩? }
```

Library declarations, function declarations, and type definitions are optional. The semi-colons (`;`) may be written or not at the end of declaration lists to ease the use of copy/paste.

3.10 Pragmas

```
<pragma_list> → % <pragma> ⟨ ; <pragma> ⟩* %
<pragma> → "<ident>":"<ident>"
```

Pragma can occur after any ident or operator. A pragma is a pair of string list that begins and ends with the terminal `%`. They are intended to be used for source recovering.

4 Calling external code

In order to use external code from Lucky, we provide a mechanism based on dynamic libraries. Such dynamic libraries should be built and used according to certain conventions that we describe in this section. Moreover, the type of imported functions should be declared in the Lucky file (cf Section 3.5). And of course, the declared types should match their definitions in the library.

BEWARE: if the types you declare in the Lucky file does not match their definitions, it might run silently returning wrong values!

4.1 Current restrictions

- Imported code can only be functions over integers and doubles.
- At least one and at most five arguments are required.
- There is an issue with integers as Lucky manipulates integers coded on 31 bits, not 32 bits. Therefore make sure you use only integers between -1073741824 and 1073741823. This may be changed in future versions.

  todo: *Add* `--int32` *and* `--int64` *options.*

A example is given in the appendix A.6. This exemple (plus the Makefile) is also provided in the distribution in the `demo-lucky/external_code` directory.

bug: `x = f(y) and z=g(x)` is not equivalent to `z = g(f(y))`, because in the former case Lucky raises an error as it believes that `x` is not bound.

4.2 External code with side-effects

Currently, external function call does not work very well if such calls are doing side-effects. For instance, the following code:

```
if 5 > random_up_to(10) then ... else
if 5 > random_up_to(10) then ... else ...
```

is not equivalent to:

```
if   5   > random_up_to(10) then ... else
if (4+1) > random_up_to(10) then ... else ...
```

because in the former case, the call to the function `rand_up_to` is done once, while it is done twice in the latter case. This (buggy) behaviour migth be fixed in future versions.

5 Numeric solver issues

Since we target the test of real-time software, we put the emphasis on the efficiency of the solver.

In order to solve numeric linear constraints, we use the library of convex polyhedron polka [Jea02] which is reasonably efficient, at least for small dimension of manipulated polyhedra – the complexity of the algorithms are exponential in the dimension of polyhedron. Polyhedron of dimension bigger that 15 generally leads to unreasonable response time.

Note however that independent variables – namely, variables that do not appear in the same constraint – are handled in different polyhedra. This means the limitation of 15 dimensions does not lead to a limitation of 15 variables.

5.1 Solving integer constraints in dimension $n \geq 2$

When the dimension is greater than 2, for the sake of efficiency, we do not use classical methods such as linear logic for solving integer constraints: we solve those constraints in the domain of rational numbers and then we truncate. The problem is of course that the result may not be a solution of the constraints.

In such a case, we chose to pretend that the constraint is unsatisfiable (after a few more tries according to various heuristics), which can be wrong, but which is safe in some sense. The rigth solution there would be to call an integer solver, which is very expensive, and yet to be done.

todo: *implement an –int-solver option that make Lucky use an integer solver when the polyhedron-based approach fails*

5.2 Fairness versus efficiency

Lucky (and Lurette) can be run in two different modes; one that emphasises the fairness of the draw; the other one that emphasises the efficiency. Indeed, suppose we want to solve the following constraint:

$$((b \wedge \alpha_1) \vee (\bar{b} \wedge \alpha_2)) \wedge \alpha_3 \wedge (\alpha_4 \vee \alpha_5)$$

where b is a Boolean, and where α_i are atomic numeric constraints of the form: $\sum_i a_i x_i < cst$. The first step is to find solution from the Boolean point of view. This leads to the four solutions:

$$b\alpha_1\overline{\alpha_2}\alpha_3\overline{\alpha_4}\alpha_5, \quad b\alpha_1\overline{\alpha_2}\alpha_3\alpha_4\overline{\alpha_5}, \quad \bar{b}\overline{\alpha_1}\alpha_2\alpha_3\overline{\alpha_4}\alpha_5, \quad \bar{b}\overline{\alpha_1}\alpha_2\alpha_3\alpha_4\overline{\alpha_5}$$

Now, suppose that:

$$\alpha_1 = 100 > x, \;\; \alpha_2 = 200 > x, \;\; \alpha_3 = x > 0, \;\; \alpha_4 = x > x, \;\; \alpha_5 = x > 1$$

where x an integer variable that has to be generated by Lucky. We use the convex polyhedron library to solve the numeric constraints, which lead respectively to the following sets of solutions:

$$S1 = b \wedge x \in [2;100]; \;\; S2 = b \wedge x = 0; \;\; S3 = b \wedge \overline{x} \in [2;200]; \;\; S4 = b \wedge \overline{x} = 0$$

In order to perform a fair draw among the set of all solutions, we need to compute the number of solutions in each of the set Si. But this computation is very very expensive for polyhedron of big dimension. Moreover, as we use Binary Decision Diagrams [Som98] to solve the Boolean part, associating a volume to each numeric part results in a lost of sharing in BBDs.

Therefore, we have adopted a pragmatic approach:

- implement an efficient mode that is fair with respect to the Boolean part only;
- implement a fair mode that performs an approximation of the polyhedron volume.

The polyhedron volume is approximated by the smallest hypercube containing the polyhedron. Note that this leads to no approximation for polyhedron of dimension 1 (intervals), and reasonable approximation in dimension 2. But the error made increases exponentionally in the dimension. Therefore, for polyhedron

of big dimension, it is better to use the efficient mode, and to rely only the probability defined by transition weights.

Note that when there are only Boolean variables as output or local variables, the two modes are completely equivalent.

5.3 Fair mode and precision and the computations

In the fair mode, we compute an approximation of polyhedron volume. But how to mix set of solutions that involves both integers and floats (which are necessarily computed by distinct polyhedra)?

The solution we have adopted is the following: relate both domain via the precision of the computations, which is a parameter of Lucky and Lurette. For example, with a precision of 2 digit after the dot, we condiser that the set $x \in [0;3]$ contains 300 solutions.

6 The Lucky file interpreter

The Lucky drawing engine is the main component of the Lurette testing tool [RWNH98, JR04, Jah04]. However, we, at Verimag, also provide an unpluggled version of this engine under the form of a command-line Lucky file interpreter. Its work on PC/Linux, Sun/Solaris, and PC/Windows-cygwin. Its usage is summarised in Figure 3.

```
usage: lucky [options]* (<file.luc>)+
   where `<file.luc> contains a Lucky program. Automata
   that share output variables are executed as if they were
   multiplied.

   options:

  --boot, -boot, -b
The Lucky machine starts generating values.
  --with-seed, --seed, -seed
Set the value of the seed the random engine is initialized with.
  --step-number, -l
Set a bound on the number of steps to perform.
  --precision, -precision, -p
Set the precision used for numerical values (number of digits).
  --step-inside
Draw inside the convex hull of solutions.
  --step-edges
Draw inside the convex hull of solutions, but a little bit more
        at edges and vertices.
  --step-vertices
Draw among the vertices of the convex hull of solutions.
  --show-aut, -s
Run lucky showing the lucky automata.
  --locals, -locals, -loc
Shows local variables.
  --oracle, -oracle
Launch an oracle (replace blank by '+', e.g., --oracle ecexe+f.ec).
  --fair, --compute-poly-volume
Compute the polyhedra volume before drawing: more fair, but more expensive.
  --verbose, -verbose, -v
Set on a verbose mode.
  --pre-processor, -pp
Pipe lucky file(s) through a preprocessor (e.g., cpp).
```

Figure 3: The lucky file command-line interpreter usage

References

[And96] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, jul 1996. 2.1

[BG92a] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 1

[BG92b] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 2.1

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 1

[Jah04] E. Jahier. The Lurette V2 User guide. Technical Report TR-2004-5, Verimag, 2004. www-verimag.imag.fr/∼synchron/tools.html. 6

[Jea02] B. Jeannet. *The Polka Convex Polyhedra library Edition 2.0*, May 2002. www.irisa.fr/prive/bjeannet/newpolka.html. 5

[JR04] E. Jahier and P. Raymond. The Lucky Language Reference Manual. Technical Report TR-2004-6, Verimag, 2004. www-verimag.imag.fr/∼synchron/tools.html. 6

[LBBG86] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. Signal , a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986. 1

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, August 1992. LNCS 630, Springer Verlag. 2.1

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. 6

[Som98] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0*, 1998. 5.2

A More Lucky program Examples

A.1 The Lucky program of Figure 2

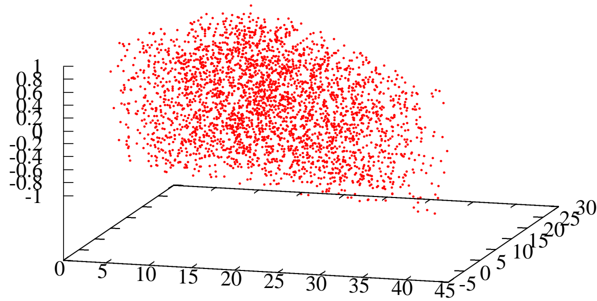
```
  --
--   This program simulates the temperature in a room that contains a
--   heater and a window which is opened from times to times.

--   Input variables are a Boolean On, which is true if the
--   heater in on and false otherwise; and a rational T, which indicates
--   the temperature outside the room (behind the window).
inputs {
     T: real;
     On: bool;
}

--   The only output variable is a rational t, which simulates the
--   temperature inside the room.
outputs {
    t: real
}

--   Local variables are the real delta, which is used to compute
--   the new temperature, and the integer cpt, which is used to count
--   the number of steps the window remains open (before rebooting).
locals {
    delta: real ~min -10.0 ~max 10.0;
    j: int ~init 0 ~min 0 ~max 20 ~default 0
}


--   Stable nodes are s_on, s_off, and s_open. Note that node labels are
--   just identifiers; the names we give them here serve only as
--   documentation.
states {
  s_init :stable; s_on   : stable; s_off  : stable; s_open : stable;
  init : transient;
  t1 : transient; t2 : transient; t3 : transient; t4 : transient
}
start_state { init }

transitions {

     init -> s_init ~cond t = 17.0;
--   By convention, no weight label stands for a weight of 1.

     s_init -> t1 ~cond On;
     s_init -> t2 ~cond not On;
     s_on -> t1 ~weight 100 ~cond On;
     s_on -> t2 ~weight 100 ~cond not On;
     s_on -> t3;

     s_off -> t1 ~weight 100 ~cond On;
     s_off -> t2 ~weight 100 ~cond not On;
```

```
    s_off -> t3;

    t1 -> s_on ~cond
              0.0 < delta and delta < 0.5 and
              t = pre t + delta;

    t2 -> s_off ~cond
              - 0.5 < delta and delta < 0.0 and
              t = pre t + delta;

    t3 -> t4 ~cond (10 <= j) and (j <= 15) ;

    t4 -> s_open ~cond t = (3.0 * (pre t) + T) / 4.0 ;

    s_open -> t1 ~cond pre j = 0 and On ;
    s_open -> t2 ~cond pre j = 0 and not On;
    s_open -> t4 ~cond pre j > 0 and j = (pre j) - 1;
}
```

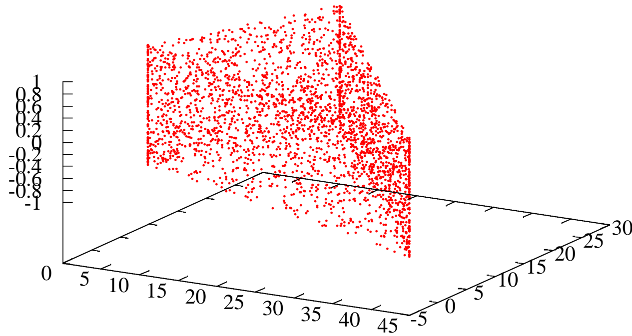
A.2 Illustration of the use of Polyhedra

```
outputs { a:real; b:real; c:real }

states { 1 : stable }
start_state { 1 }

transitions {
    1 ->  1 ~cond abs c < 1.0
                  and (a <= 40.0 - b) and (2.0 * a >=  b - 10.0)
                  and (a >= -3.0 * b + 30.0)
}
```

Generated by `lucky --draw-inside -l 3000` and displayed with Gnuplot



Generated by `lucky --draw-edges -l 3000` and displayed with Gnuplot



A.3 Illustration of the use of infinite weights

```
-- A lucky example illustrating the use of infinite weigths
-- as well as the ~default option

inputs  { }
outputs {
    cpt : int
      ~init 0
        -- initial value of pre cpt is 0
      ~default (pre cpt + 1)
        -- if cpt does not appear in a transition,
        -- it is incremented at each step
}
locals  { }

states { 0 : stable ; 1 : stable ; 2 : stable ; 3 : final}
start_state  { 0 }

transitions {
     0 -> 0 ;
     0 -> 1 ~weight infinity ~cond pre cpt = 3 and cpt = pre cpt;
        -- This infinitely weighted transition will be always taken
        -- as soon as it is possible (ie, when pre cpt = 3)
     1 -> 1 ;
     1 -> 2 ~weight infinity ~cond pre cpt = 6  and cpt = pre cpt;
     2 -> 2;
     2 -> 3 ~weight infinity ~cond pre cpt = 10 and cpt = pre cpt
}

-- generates the sequence of integers:
--    1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 10, 10

-- The integer 3 is generated twice:
--    * once at step 3 when cpt is bound to 3
--    * once at step 4 because the transition "0 -> 1" is satisfiable
--      iff the previous value of cpt is 3.

-- Ditto for 6 and 10
```

A.4 Illustration of the use of dynamic weights

```
-- A lucky example illustrating the use of dynamic weigths

inputs   { }
outputs  {
  x : real
}
locals   {
  cpt : int ~init 0
}
states { In_use : stable ; Breakdown : stable }
start_state { In_use }

transitions {
     In_use -> In_use
             ~weight 100
             ~cond    cpt = pre cpt + 1 and 5.0 < x and x < 10.0;
     In_use -> Breakdown
             ~weight pre cpt
                   -- Breaking down becomes more and more likely to happen
             ~cond  x = 0.0;
     Breakdown -> Breakdown ~cond x = -1.0
}

-- Generates sequences such as:
-- 5.33 5.46 7.92 8.91 9.45 7.58 8.63 5.13 9.51 5.26 8.13 8.74
--              7.67 9.04 7.46 5.64 0.00 -1.00 -1.00 -1.00 ...
```

A.5 Illustration of the use of structured types

```
-- A lucky example illustrating the use of strutured types,
-- or, an obfuscated way to generate sequences of integers
-- and floats betwenn 0 and 10.

typedef {
  enum = (zero, one, two) ;
  array = int ^ 4 ;
  array_strange = array ^ 2 ^ 5 ;
  struct = { f1:bool; f2:enum; f3:array_strange }
}
inputs  { }
outputs { s : struct ~default {f1 = false ; f2=two ; f3= 0^4^2^5 } }
locals  { t : array_strange ~min 0^4^2^5 ~max 1000^4^2^5 }

states { start : stable ; 1 : stable ; 2 : stable }
start_state  { start }

transitions {
      start -> 1 ;
      1 -> 1 ~cond s.f3 = t and s.f2 = zero ;
      1 -> 1 ~cond s.f3[1] = [1,3,5,7]^2 and s.f2 = one ;
      1 -> 1 ~cond s = { f1 = true; f2 = two ; f3 = t }
}

-- generates sequences such as:
-- # step 1
-- f 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--  0 0 0 0 0 0 0 0 0 0 0
-- # step 2
-- t 2 885 852 15 688 195 748 211 330 571 544 916 734 982 86 422
--  168 519 533 609 981 819 671 839 414 821 123 581 93 504 150
--  739 702 273 108 375 591 216 649 21 354
-- # step 3
-- t 2 22 593 338 695 230 234 742 568 716 761 791 728 378 599 135
--  452 579 140 564 959 313 705 220 24 865 625 146 10 978 903 560
--  862 148 491 922 685 718 252 564 207
-- # step 4
-- f 1 0 0 0 0 0 0 0 0 1 3 5 7 1 3 5 7 0 0 0 0 0 0 0 0 0 0 0 0 0
--  0 0 0 0 0 0 0 0 0 0
-- # step 5
-- t 2 136 612 115 790 190 364 79 160 883 916 550 810 706 729 0 369
--   362 58 726 606 640 538 484 739 380 416 745 127 856 718 443 596
--   272 219 31 174 104 302 833 206
-- # step 6
-- t 2 48 622 432 676 379 6 96 414 703 779 726 700 748 426 827 955
--  426 253 295 559 897 835 559 889 206 370 436 247 545 953 391 356
--  921 88 195 789 317 145 830 308
-- # step 7
-- f 1 0 0 0 0 0 0 0 0 1 3 5 7 1 3 5 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--   0 0 0 0 0 0 0 0 0
-- # step 8
-- f 0 377 457 252 923 747 577 756 306 817 558 31 134 713 762 47 578
```

```
--   501 701 508 946 440 390 347 554 517 551 705 52 203 287 1 46 134
--  272 117 289 639 625 286 553
-- # step 9
-- f 0 998 821 27 986 203 666 325 995 726 536 991 574 132 714 825 161
--   517 546 589 454 182 323 235 691 854 659 751 282 527 58 724 852
--  830 96 268 8 313 106 656 508
-- # step 10
-- f 1 0 0 0 0 0 0 0 0 1 3 5 7 1 3 5 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--  0 0 0 0 0 0 0 0 0
```

A.6 Illustration of the use of external function calls

```
-- A simple Lucky program calling a function "rand_up_to" defined in "foo.so"

libraries {
   foo_lib  = "./i386-linux-gcc3/foo.so";
   math_lib = "libm.so"
      -- the standard C math lib (which path ougth to be in LD_LIBRARY_PATH)
}
functions {
   sqrt : float -> float : math_lib;
   sin : float -> float : math_lib;
   rand_up_to : int -> int : foo_lib;
}
outputs {
   f1 : float ~init 1.0;
   f2 : float;
   i : int;
}
nodes {0: stable }
start_node { 0 }
transitions {
   0 -> 0 ~cond
          0.0 < f1 and f1 < 100.0
      and f2 = sin(sqrt(pre f1))
      and i = rand_up_to(10)
}
-- "lucky -l 10 call_external_c_code.luc" generates outputs such as:
--
-- #The random engine was initialized with the seed 3064352
-- #inputs
-- #outputs "f1":real "f2":real "i":int
-- #step 1
-- #outs 51.62 0.84 9
-- #step 2
-- #outs 64.93 0.78 4
-- #step 3
-- #outs 27.27 0.98 8


-----------------------------------------------------------------------

// The "foo.c" program that is used to generate de "foo.so" library
#include <stdio.h>

int rand_up_to(int max){
// uniformly draws an integer between 0 and max.
// Not the most useful function for Lucky :-)
  double r = ((double) random ());
  int res = ((int) ((r * (((double) (max+1)) / ((double) RAND_MAX)))));
  return res;
}
```

B Automata Product

First of all, the behaviour which is in general expressed as a a set of concurrent automata, is semantically equivalent to the one of a single *product automaton*. Two different products are defined, depending on the semantics chosen for weights composition (Figure 1). The simplest one is almost a classical *synchronous product*:

- the global state space is the Cartesian product of the component state spaces; the global initial state is the tuple of initial states; a global state is stable if it is composed of stable states only.
- the definition of global transitions is almost a classical composition where constraints are combined with the $\wedge$ operator, and weights with the $*$ operator. The only problem is to enforce the synchronisation on stable states; we note s_1, s_2 stable states, t_1, t_2 transient states and q_1, q_2 any states, so:
  - $(s_1, s_2)\xrightarrow[w_1 * w_2]{f_1 \wedge f_2}(q_1, q_2)$ iff $s_1\xrightarrow[w_1]{f_1}(q_1)$ and $s_2\xrightarrow[w_2]{f_2}(q_2)$
  - $(t_1, t_2)\xrightarrow[w_1 * w_2]{f_1 \wedge f_2}(q_1, q_2)$ iff $t_1\xrightarrow[w_1]{f_1}(q_1)$ and $t_2\xrightarrow[w_2]{f_2}(q_2)$
  - $(s_1, t_2)\xrightarrow[w_2]{f_2}(s_1, q_2)$ iff $t_2\xrightarrow[w_2]{f_2}(q_2)$, and symmetrically for (t_1, s_2).

The second kind of "product" (as shown in Figure 1c) is a little bit tricky: the idea is to introduce, for each component, an additional *starting* transient state $\hat{s}$ and an additional *waiting* transient state $\breve{s}$ for each stable state s. The global state space is then defined as the Cartesian product over the extended component state spaces. The definition of initial and stable global states does not change. The transitions are defined in such a way that each component performs its reaction in turn. We only give the rules where the first component starts, the other case is similar:

- from a global stable state, the first component may start while the other waits: $(s_1, s_2)\xrightarrow{}(\hat{s}_1, \breve{s}_2)$,
- the starting component performs its first transition:
  $(\hat{s}_1, \breve{s}_2)\xrightarrow[w_1]{f_1}(q_1, \breve{s}_2)$ iff $s_1\xrightarrow[w_1]{f_1}q_1$,
- the starting component is not yet in a stable state, and it performs another transition: $(t_1, \breve{s}_2)\xrightarrow[w_1]{f_1}(q_1, \breve{s}_2)$ iff $t_1\xrightarrow[w_1]{f_1}q_1$,
- the starting component has reached a stable state, and the waiting one starts its reaction: $(s_1, \breve{s}_2)\xrightarrow[w_2]{f_2}(s_1, q_2)$ iff $s_2\xrightarrow[w_2]{f_2}q_2$,
- the second component performs transitions until it reaches a stable state: $(s_1, t_2)\xrightarrow[w_2]{f_2}(s_1, q_2)$ iff $t_2\xrightarrow[w_2]{f_2}q_2$.