Pattern-based Definition and Generation of Components for a Synchronous Reactive Actor-oriented Language

Ivan Jeukens

[1]Department of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
Campus Universitario, S/N, Lagoa Nova, Natal, Brazil

`ijeukens@dimap.ufrn.br`

Abstract. *This paper describes a method for specifying a component and automatically generating its code under a synchronous reactive language. The main advantage of the proposed method is that the generated code is guaranteed to be correct by construction with respect to the language's semantics. In this work, besides from capturing syntactic information, the interface of a component is also used to model the behavior of the component through patterns of required data. Necessary and sufficient conditions for ensuring correct components are proven and automatic code generation algorithms are described. A case study demonstrates the usefulness of the method.*

1. Introduction

The field of embedded software development requires its own languages, due to its specific needs. For instance, concurrency and a notion of time should be first class citizens in such languages []. One popular approach, being applied within academia and industry, is the combination of a coordination language and a programming language under a component-based model. The main element of the coordination language is its semantics, called Model of Computation (MoC), and is responsible for defining the interaction semantics between components. When this interaction is based on data exchange (receive input events, compute, produce output events), the component is called an actor []. The behavior of each actor is captured by predefined services, using some programming language. Examples of actor-oriented frameworks includes [][][][].

The synchronous reactive (SR) MoC is an example of successful semantics for the programming of control-oriented embedded systems. Languages such as Esterel [] and Lustre [] have been used to develop and deploy code for industrial scale projects. This MoC was also applied within an actor-oriented framework, as described in []. It is based on the perfect synchrony hypothesis [**?**]: the system is able to produce a response instantaneously fast to any set of input events. Time is divided into a discrete set of instants, where each reaction to input events corresponds to an instant. Central to the SR MoC is the ability to compute the fixed-point for the system at any instant. This guarantees that the system's implementation is deterministic. In [], the necessary theoretical constraints for fixed-point computation are established and an execution semantics is described.

Unfortunately, there is a gap between the theoretical requirements for ensuring determinancy and actually obtaining it from a specification composed of actors written in a programming language. There are two places where errors can be introduced: 1) within the component's code; 2) at the composition of components. The first error might occur when components are allowed to produce output events without knowing some input events. Those components are named **non-strict**. The second one happens when cyclic dependencies among signals exists, therefore not allowing the production of output events (a condition called deadlock). In order to remove cycles, non-strict components must be employed.

Currently, the designer is responsible for ensuring that a component respects the SR semantics. The contribution of this paper is an approach that enhances the interface of a component with patterns of input events. Those patterns capture the possible conditions where output events can be generated. Therefore, the designer has to determine what are the needed scenarios (input signals and internal state) for computation. From an actor interface with patterns, I have developed algorithms for generating its code, ensuring that it is consistent with the SR semantics.

The remainder of this paper is organized as follows: section 2 contains a brief explanation of some syntactic elements of and actor and the semantics of the SR MoC; in section 3, the concept of a pattern and pattern-based interface are presented; section 4 gives the necessary and sufficient conditions for obtaining a consistent SR actor from a pattern-based interface; in section 5, an expression language for capturing patterns is show; section 6 presents the code generation algorithms; section 7

2. Background

2.1. Language syntax

The main syntactic elements of an actor are:

- input port: used to receive data from the outside;
- output port: used to send data to the outside;
- attribute: associate a variable to an element;
- method: contains the implementation of a service provided by the actor;

Actors exchange data through connections, which are relations with one output port and at least one input port. A topology is a set of actors and connections involving ports of those actors. When a MoC is associated with a topology, it can be simulated or executed. A topology may itself be viewed as an actor. In this case, it is called a hierarchical actor, and its behavior is given by the contained composition of actors. The ports of a hierarchic actor are ports of the contained actors designated to be viewed by the outside. A model for a (sub)system can be realized as a topology.

Depending of the MoC, an actor must provide certain services. For an actor governed by the SR MoC, it has the obligation to provide one service: output computation. Output computation is where new data may be produced, and sent through an output port, as the result of new data (events) on the actor's input ports. Three optional services may also be provided:

- initialization: actions that should be performed before execution of the system are associated with this service;
- state update: if the actor has state, this service is where it should be updated (based on the current state and the instant's events on the actor's input ports);
- termination: allows the execution of actions while shutting down the system.

2.2. Synchronous reactive semantics

A central point of the SR semantics is to provide determinancy under concurrency, since there are no restrictions to how actors can be composed, or on the processing order of input events. The necessary and sufficient conditions for obtaining determinism are []:

1. The possible values associated with any connection should be drawn from a complete partial order (CPO)[1];
2. Each component must implement a monotonic function.

The CPO adopted is the lifting of the data type associated with the respective connection, adding the absent state (denoted by ϵ) as a defined state, that is, the absent event is incomparable with any other possible value of the type. Such a CPO is also known as flat (bottom $\oplus$ anti-chain). Figure 1 (b) contains the Hasse diagram for the CPO of a connection.

The reason for imposing these two conditions is a fixed-point theorem that states: *let P be a CPO, F: P $\rightarrow$ P and continuous, then the least fixed point (lfp) of F equals to the least upper bound on the chain* $\{\perp, F(\perp), F^2(\perp), ...\}$.

The height of a CPO is one less than the length of its longest chain. If P is a finite height CPO, then a monotonic self-function on P is also continuous. Therefore, the theorem guarantees the existence of the *lfp*, and provides a way to compute it. Since all the CPOs of a topology are of finite height, the *lfp* computation is finite. Obtaining a *lfp* at any instant guarantees determinancy.

From the designer's view point, the condition that each connection is associated with a flat CPO implies that an input port may have an undefined state ($\perp$) or a defined state (with or without data). The flat CPO of an output port, together with the monotonicity condition, force that once the state of an output port is defined, it can no longer be changed in the same instant. If a datum was sent, no other can be sent in the same instant. As a result, software tools and libraries that implement the SR MoC have methods associated with ports that can test the state of a port and produce error messages if one tries to change a defined state of an output port.

A simple way to ensure that a source code implements a monotonic function would be to allow the definition of the actor's outputs only when all input ports have a defined state. Such type of actor is called strict. This is restrictive, since only acyclic topologies can be implemented. Therefore, non-strict actors are needed. Ensuring monotonicity of non-strict components is not trivial, and it is the problem addressed in this paper.

3. Definition of SR Components

3.1. Patterns

An elementary way of specifying a function is to give its graph, that is, a set of pairs of elements where the first element is a point of the domain, and the second a point from the codomain. Obviously, this is used only for functions with very small graphs. An improvement is to use a pattern, that is, the addition of symbols to represent several

[1]Readers unfamiliar with order theory, please refer to [**?**] or other reference.

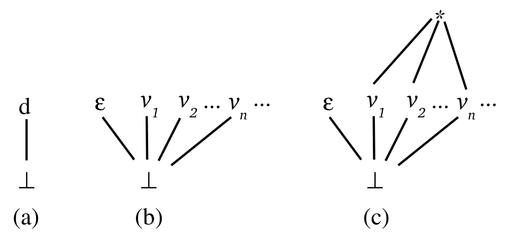

Figure 1. Three versions of a CPO

points of the domain and codomain. A classical example would be boolean functions, where each dimension of a cube may be associated with the boolean values or the don't care symbol.

The pattern that I have employed in this work borrows the notion of a don't care, by adding only one symbol to a flat CPO: the wildcard symbol (*). This symbol represents any possible defined value from the data type associated with the respective port. Such a CPO will be called extended CPO, denoted by **CPO***. Let **I** $= (CPO_1 \times ... \times CPO_n)$ denote a vector of CPOs associated with the *n* input ports of an actor. Likewise, **O** $= (CPO_1 \times ... \times CPO_m)$ denote a vector of CPOs associated with the *m* output ports. The sets **I*** and **O*** denote the sets **I** and **O** respectively, when the cartesian product is taken with respect to **CPO***. The order relation $\leqslant$ associated with a CPO is extended in CPO^*, making $\perp \leqslant *$, $v \leqslant *$ and $\epsilon \nleqslant *$. Figure 1 (c) shows the Hasse diagram for a **CPO***.

A pattern[2] is an element of the set **I***. The set **I*** has an order relation $\preccurlyeq$ given by applying $\leqslant$ componentwise. It is easy to see that the set **I*** together with $\preccurlyeq$ forms a CPO. The state of an actor is omitted from a pattern since it is always available to the computation service, independently of any input event.

For example, consider the fragment of code from Figure **??**(a), with the association of an input port to a position in a pattern given by: (*preferred*, *alternate*). Figure **??**(b) shows the patterns that capture the behavior of the code from **??**(a)

Patterns are classified according to its constituent symbols:

- strict: does not contain the symbol $\perp$ in any position;
- non-strict: does contain the symbol $\perp$ in at least one position;
- concrete: does not contain the symbol * in any position;
- abstract: does contain the symbol * in at least one position.

3.2. Pattern-based actor interface

An actor with *n* input ports, and *m* output ports, is viewed as a set of functions: $actor : (\mathbf{I}, \mathbf{E}) \rightarrow$ **O**. The set **E** is the state of the actor represented as a vector of variables, each associated with a data type. For the sake of simplicity, it will be assumed henceforth that each actor contains only one output port.

The idea behind a pattern is to specify a group of input events necessary for the computation of the output value. If one of those groups occur during an instant, it $\underline{\text{may}}$ be

[2]It could be called input pattern instead, since only input ports are considered. Currently, the set **O*** has no role in my strategy.

```
computationService(Inputs [] inputs) {
        for(int i = 0;i < patterns.size();i++) {
            if(patterns[i].Match(inputs) && patterns[i].Def(inputs)) {
                send patterns[i].F(inputs);
            }
        }
}
```

Figure 2. Pseudocode for an pattern-based actor.

possible to compute the value. An interface populated with patterns gives all the possible conditions of output computation. Therefore, given a concrete pattern for an actor, one first needs to know if it matches one of the patterns, and if so, compute the value based on the available data. This leads to a decomposition of the actor computation service in auxiliary functions, that are systematically called. For matching a pattern we have:

1. Ins : **I*** $\times$ **I** $\rightarrow$ **I** : for $ip \in \{1..n\}$, $p \in$ **I*** and $cp \in$ **I**, *Ins* is a partial function that is defined only when: 1)for every position *ip* of *p* that has a defined state, then the position *ip* of *cp* must have the same defined state; 2) and for every position *ip* of *p* that has a wildcard (*), then the position *ip* of *cp* must have a defined and not absent state, i.e, a value. If defined for a certain *p* and *cp*, the *Ins* function returns a concrete pattern with the same defined states of *p* and with every wildcard of *p* replaced by the corresponding defined state of *cp*;
2. $Match$: **I*** $\times$ **I** $\rightarrow$ $Bool$: given $p \in$ **I*** and $cp \in$ **I**, returns **false** whenever $Ins(p, cp)$ is not defined. When $Ins(p, cp)$ is defined, returns **true**. In this case, the pattern *p* is said to be **matched**;
3. Def : **I** $\times$ **I** $\times$ **E** $\rightarrow$ $Bool$: given $p \in$ **I***, $cp \in$ **I**, $e \in$ **E**, and $Match(p, cp) =$ **true**, returns **true** if the output can be defined. This function is given by the designer. A pattern *p* that has $Def(p, cp, e) =$ **true** is said to be **satisfied**;

The purpose of the *Ins* and *Match* function is to check if there is enough information at the actor's input ports in order to satisfy a given pattern. The *Def* function is only called if the respective *Match* function returns **true**.

Once a pattern is satisfied, the value for the respective output port is computed by the **F** function:

1. F_i : **I** $\times$ **E** $\rightarrow$ CPO_o: given $cp \in$ **I** and $e \in$ **E**, if pattern *i* is satisfied, computes the value of the output. This function is given by the designer.

Under this decomposition of the component's behavior in auxiliary functions, for each distinct pattern, there are distinct *Match*, *Def* and *F* functions associated. Figure 2 shows a pseudocode for the computation service of an actor, generated from a pattern-based interface.

Although correct, the code from Figure 2 does not guarantee an implementation that respects the SR semantics. In order to capture monotonic functions using patterns, it is imperative to establish how the *F* function is computed when more than one pattern is satisfied.

4. Ensuring Monotonicity

4.1. Necessary Conditions

Before proving the necessary and sufficient conditions for obtaining monotonicity from patterns, some definitions are needed:

Definition 1 The set of all patterns given by a designer in order to specify a component is denoted by the set **A**.

Definition 2 Let $cp \in$ **I** be strict, $e \in$ **E**. The set of all patterns *p*, such that $p \in$ **A** and $Def(p, cp, e) =$ **true** is denoted by the set **B**.

The set **B** is obtained with respect to a concrete and strict pattern. Therefore, for each possible pair (*cp*, *e*), and a set **A**, there is one and only one associated set **B**.

Lemma (L1) $\forall b_1, b_2 \in$ **B**, $\exists j \in I^* | j = b_1 \vee b_2$[3].

Proof By the definition of the *Def* function, there must be $cp \in$ **I**, such that $Match(b_1, cp) = Match(b_2, cp) =$ **true**. Let $j = Ins(b_1, cp) \vee Ins(b_2, cp)$.If *j* is inexistent, then there is at least one position that has defined and different states in $Ins(b_1, cp)$ and $Ins(b_2, cp)$. This implies that *cp* is inexistent, hence a contradiction. □

The following proposition formally establish the necessary and sufficient condition for a given set **A** to specify a monotonic function.

Proposition 1: A set **A** describes a monotonic function ⇔ $\forall cp \in$ **I** and $e \in$ **E**, the respective set **B** $= \{b_1, b_2, ..., b_n\}$ is such that:

$$\forall b_i, b_j \in \mathbf{B}, F_{b_i}(Ins(b_i, cp), e) = F_{b_j}(Ins(b_j, cp), e).$$

Proof: (⇒) Since the output CPO is flat, and the function is monotonic, then all defined values must be equal. (⇐) Let $cp_1, cp_2 \in$ **I**, where $cp_1 \preccurlyeq cp_2$. Either $cp_1 \prec cp_2$ or $cp_1 = cp_2$. In the first case, for some inputs, cp_1 is undefined whether cp_2 has a defined state. Let **B1** be the set **B** with respect to cp_1, *e* and **A**, and **B2** be the set **B** with respect to cp_2, *e* and **B**. Clearly, **B1** $\subseteq$ **B2**. Therefore the output value computed from cp_1 and cp_2 is the same. The second case is trivial.□

Proposition 2: The function *F* is monotonic ⇔ $\forall cp \in$ **I**, $e \in$ **E** and respective set **B**:

$$\forall b \in \mathbf{B}, F(Ins(b, cp), e) = F(Ins(\bigwedge \mathbf{B}^{4}, cp), e).$$

Proof: (⇐) By proposition 1.

(⇒)

□

It is possible to reduce the set **B** to a minimum amount of patterns, by removing comparable patterns:

[3] $\vee$ and $\wedge$ denote the join and meet operations respectively. For patterns, they are applied point-wise. If in any position the operation is not defined, then the resulting pattern is also not defined.

[4] The meet of all patterns of **B**.

Definition 3 The set **C** denotes the greatest subset of **B**, such that any two patterns belonging to **C** are incomparable (**C** is an anti-chain).

Lemma (L2) $\forall cp \in$ **I**, $e \in$ **E** and respective sets **B** and **C**, $\bigwedge \mathbf{B} = \bigwedge \mathbf{C}$.

So far, proposition 2 has established an important fact: when more than one pattern is satisfied in an instant, the output value has to be computed based only on the common information to all satisfied patterns. It remains to be determined the conditions where a set **C** has been obtained.

4.2. When can an output be defined?

Definition 4 A **Pattern Group** for a pattern p_1 (PG(p_1)) is a subset of set **A**, such that if $p_2 \in A$ and $p_1 \vee p_2 \in$ **I***, then PG(p_2) = PG(p_1).

A pattern belongs to one and only one PG. It is possible that two different patterns p_i and p_j in the same PG do not have a join. In this case, they belong to the same PG because there must exist a pattern $p_k \in$ PG such that $p_i \vee p_k$ and $p_j \vee p_k$ exists. Therefore, forming all pattern groups of a set **A** partitions it. For example, each pattern of Figure **??**(b) defines a different PG.

Lemma (L3) $\forall p_1, p_2 \in$ **B**, $cp \in$ **I** and $e \in$ **E**, if $Def(p_1, cp, e) =$ **true** and $Def(p_2, cp, e) =$ **true** $\Rightarrow$ PG(p_1) = PG(p_2).
Proof from lemma L1.

Definition 5 Let p_1 and $p_2 \in A$, such that PG(p_1) = PG(p_2). They belong to the same **Definition Constraint Group** (DCG) when they are incomparable and $\exists p_3 \in$ **I*** such that $p_1 \vee p_2 \preccurlyeq p_3$.

Definition 6 The pattern formed by the join of all patterns of a DCG is called the **characteristic pattern** (c-pattern) of the DCG.

A DCG is said to be **complete**[5] when there are no more patterns belonging to the respective PG that can be included in the DCG, and **unitary** when it is composed of only one pattern. A PG composed of only one pattern is also called unitary. When all patterns of a DCG are satisfied, the DCG is said to be **satisfied**. The set of DCGs of a PG can be equipped with an order relation ($\precsim$):

Definition 7 Given two DCGs D1 and D2, D1 $\precsim$ D2 $\Leftrightarrow \forall t_1 \in$ D1, $\exists t_2 \in$ D2 such that $t_1 \preccurlyeq t_2$.

The following proposition constrains when the output value can be computed:

Proposition 3: For a set **A**, $\forall cp \in$ **I** and $e \in$ **E**, the set **C** has been obtained when:

C1 *cp* is a strict pattern;
C2 at least one DCG is satisfied;
C3 for every pattern $p_i \in$ **A** that has $Match(p_i, cp) =$ **false**, $\exists p_j \in$ **B** such that $p_j \prec p_i$.

[5]Henceforth, all DCGs are complete unless otherwise stated.

```
expression =
     'independent'
   | inputExpression

inputExpression =
     portExpression
   | inputExpression 'OR' portExpression

portExpression = implicitExpr | explicitExpr

explicitExpr =
     {port}* '.' state
   | explicitExpr 'AND' explicitExpr
   | explictiExpr 'OR' explicitExpr

implicitExpr =
     'EP.' state
   | 'AP.' state
   | 'SP'
   | 'CP.' number '.' state

state = '*' | 'absent' | 'value'
```

Figure 3. The syntax for the pattern expression language. The *value* terminal means a value from the respective type associated with the port.

Proof: C1 and C3 are clearly true. By the definition of a satisfied DCG, all of its patterns are incomparable and satisfied, as in the set **C**. Let D1 be the satisfied DCG. Let $p \in$ **B** and p not in D1. Two facts are possible for *p* to not be in D1: (1) it does not have a join with the patterns of D1;(2) it is comparable with some pattern of D1. The first case is not possible, because *p* could not be in **B** in this case. Therefore, either $d \prec p$ or $p \prec d$, for some pattern *d* $\in$ D1. If $p \prec d$, then *d* will not belong to the set **C**, being replaced by *p*. If $d \prec p$, then *p* does not belong to **C**. Hence, whenever a DCG is satisfied, all patterns of the set **C** are known.□

4.3. Output definition methods

Although proposition 3 states when an output can be defined, it is interesting to consider the definition of the output when a pattern *p* is satisfied. Is this case, a conservative assumption has to be taken: all patterns belonging to all DCGs that contain *p* must be considered to be satisfied. It is not necessary to take into account a pattern $d \prec p$, because *p* is tested only when *d* is not satisfied. This type of output definition method is called **pattern-based**, where the one that follows proposition 3 is called **DCG-based**.

It is easy to see that only monotonic functions will be captured with pattern-based definition. However, there are monotonic functions that cannot be captured by this approach. For example, consider the four-input patterns: p1 = (true, *, $\perp$, $\perp$), p2 = (false, $\perp$, *, $\perp$), p3 = ($\perp$, $\perp$, $\perp$, *). With pattern-based definition, the output function would never have an input available. While with DCG-based, if p3 is not satisfied, then p1 or p2 could use its inputs.

5. Pattern Expression Language

The patterns of an actor interface are specified using an expression language. An advantage of this approach is more flexibility for specifying patterns that range over all input ports. Figure 3 presents the syntax of the first version of the language.

For each actor output port, an expression is specified. The *independent* reserved word identifies an output that does not depend on any input port. No pattern will be associated with such an output.

The semantic meaning of an expression for an output that depends of inputs is a list of patterns. An expression is formed by explicit and/or implicit subexpressions over input ports.

In an explicit subexpression, the state of an input port is directly associated with its name. For example, an actor with four input ports (i1..i4), the subexpression *i1,i2,i4.10* generates the pattern *(10,10,*$\perp$*,10)*. The *AND* and *OR* operators are used to combine explicit subexpressions. The *AND* of two patterns generates a new pattern, combining the defined states of both patterns. If the two patterns have a defined state for the same input port, then they must be equal in order to not reject the expression and abort the translation procedure. The *OR* of two patterns generates a list containing both patterns. The *AND* and *OR* of two subexpressions produces a list of patterns by applying the respective operator to every pair of patterns, where the first pattern comes from one subexpression, and the second from the other subexpression.

Implicit subexpressions generates a list of patterns according to an operator. Four operators are available for implicit subexpressions: *AP*, *EP*, *SP* and *CP*. The first generates a pattern with all positions equal to the specified state. The second generates a list of patterns, one for every possible input port, where each pattern has the position of the respective input port with the specified state. The *CP* operator generates a list of patterns containing every possible pattern with *n* (an parameter to the operator) defined inputs. Thus, for an actor with *n* inputs, *AP.state* and *EP.state* are equivalent to *CP.n.state* and *CP.1.state* respectively. The *SP* operator is used for capturing a strict function, that is, it specifies all possible strict patterns.

Implicit and explicit subexpressions may be combined only via the **OR** operator. During the process of translation of an expression into patterns, some checks are performed in order to avoid conflicting patterns. For example, it is not possible to have a strict pattern specified together with an *SP* subexpression.

6. Code Generation

The process of code generation produces the skeleton of code for the computation service. During an instant, every time an actor evaluates its computation service[6], it must be determined whether the conditions of proposition 3 are satisfied, by inspecting the returned value *Def* function of each matched pattern. Then, the *F* function can be called, respecting the conditions of proposition 1 and 2. Before describing how the skeleton of code is produced for DCG-based and pattern-based methods, some special cases are described, together with an intermediate data structure that the process is based on.

6.1. Special Cases

There are four special cases for code generation:

1. $|\mathbf{A}| = 0$: the function F is independent of any input event;
2. $|\mathbf{A}| = 1$: the function F is associated with the only pattern and has as parameters all inputs specified by the pattern;
3. all PG are unitary: by L3, a distinct function F is associated for each pattern and has as parameters all inputs specified by the pattern;
4. all DCGs of a PG are unitary: in this case, the set of patterns of the PG forms a chain under $\preccurlyeq$. Therefore, the set $\mathbf{C}$ is allways composed of one pattern. A distinct function F is associated with each pattern, and has as parameters all inputs specified by the pattern.

[6]If a component is implemented as a strict function, it will have its computation service evaluated only once during an instant.

In the general case, patterns are partitioned into PGs. Once a pattern is matched, only patterns from the same PG can be matched. For patterns belonging to the same PG, three items must be defined in order to produce executable code:

1. An order for calling the *Def* functions must be established;
2. The implementation of the *F* function;
3. What are the parameters for the *F* function when a DCG/pattern is satisfied;

6.2. Pattern Graph

Central to the code generation processes is the classification of patterns into PGs and DCGs. For PGs, it is necessary to know if two patterns have a join. For DCGs, it is also necessary to know if two patterns are incomparable. A complication to the process of pattern classification is the presence of implicit subexpressions. One solution would generate all possible patterns for a given implicit operator. However, this is not necessary, since the meaning of any implicit operator is: *whenever n inputs have a certain state, the output can be defined*. The same *Def* function can be used for all patterns generated from an implicit operator, since knowing the exact inputs that have the state specified by the operator is irrelevant, only the number of such inputs is important.

Definition 8 The **level** of a pattern is the number of inputs with a defined state. The **anti-level** of a pattern is the number of inputs with undefined state.

A pattern associated with an implicit subexpression is represented by a 3-tuple: (Operator, Level, State). Such pattern is called an **implicit** pattern[7].

The code generation algorithms represent patterns and their relations via a graph, called the *pattern graph*. To each node is associated a pattern of the set **A**. There are three types of edges, representing the respective relations, between any two different nodes: *join*, *incomparable* and *comparable*. The first two are undirected edges, while last one is directed. The pattern graph is constructed by analyzing every pair of patterns, and determining which relation holds between them. For any two patterns p1 and p2, the relations between them are computed using the following procedures:

1. *Join* Edges

Case 1 (Both patterns are explicit): by directly comparing each position and searching for incompatibilities (defined but different states).

Case 2 (p1 is implicit and p2 is explicit): in this case, a *join* edge means that there is at least one pattern generated by the implicit pattern, such that there is a join with p2. Such situation happens when: $alevel(p1) \geq level(p2) - es$, where *alevel* returns the anti-level of a pattern, *level* returns the level of a pattern and *es* is the number of inputs that have the same state in both pattern.

Case 3 (Both patterns are implicit): in this case, the equation of Case 2 must also hold in order to every pattern generated from an implicit pattern to have a match with the other implicit pattern.

2. *Incomparable* Edges

[7]Whenever necessary, a pattern will be called explicit in order to distinct it from an implicit pattern.

```
if(PG == -1) {
  if(Match(minimalPatterns[0], cp)) {
     PG = 0;
  }
  else
  if(Match(minimalPatterns[1], cp)) {
     PG = 1;
  }
  else
  ....
  if(Match(minimalPatterns[n], cp)) {
     PG = n;
  }
}
```

(a)

```
if(PG == n) {
   if(satisfiedAll(DCGs[n][0], cp) {
      F( computeArgs(cp) );
   }
   else
   ...
   if(satisfiedAll(DCGs[n][n], cp) {
      F( computeArgs(cp) );
   }
   else
   if( isStrict() || noMoreSatisfiable() ) {
      F( computeArgs(cp) );
   }
}
```

(b)

Figure 4. (a) detects with PG has matched patterns. (b) Code for DCG-based output definition. The matrix DCGs contains the ordered list of DCGs.

Case 1: p_1 must have a join with p_2. By directly comparing each input and searching for a pair of distinct positions *i* and *j*, such that $p_1(i) < p_2(i)$ and $p_2(j) < p_1(j)$.

Case 2: an *incomparable* edge between an implicit pattern and an explicit pattern means that there is at least one pattern generated from p1, such that it is incompatible with p2. Two situations are possible:

- if there is a *join* edge between p1 and p2 and there is at least one defined state in p2 such that it is different from the state of p1;
- if there is a *join* edge between p1 and p2 and the anti-level of both p1 and p2 is greater or equal to one.

Case 3: the same two conditions of Case 2 are applied.

3. *Comparable* Edges

All cases: there is a join edge between the nodes and no incomparable edge.

The classification of patterns in PGs is done by finding the connected components of the pattern graph, considering only the *join* edges. Each connected component is exactly a PG. Figure 4 (a) shows a pseudocode that detects which PG has to be considered during an instant. The array *minimalPatterns* contains the set of minimal patterns of each PG, found by looking at the pattern graph for nodes that are not target of an comparable edge.

6.3. DCG-based definition

Once the PGs are obtained from the pattern graph, it is necessary to compute the DCGs of each PG. It should be clear that any implicit pattern generates a DCG, and its c-pattern is strict. The following proposition presents an useful result related to DCG computation:

Proposition 4: Let *pi* be an implicit pattern and *pe* be an explicit pattern, such that *pi* and *pe* forms a DCG. Let $j \in$ **I***$|j = pi \vee pe$, then *j* is strict.
Proof (by contradiction): Let *j* be non-strict. Then, in at least one position of all joinable patterns[8], the state is $\perp$. Let *i* denote this particular position. Therefore, every pattern generated by *pi* that has the state *s* in position *i* must not have a join with *pe*. However,

[8]*pe* plus the patterns of *pi* that have a join with *pe*.

```
boolean satisfiedAll(patterns []p, pattern input) {
  boolean isSatisfied = true;
  for(i = 0;i < p.size;i++) {
    if(satisfied[p[i]]) continue;
    if(Match(p[i], input)) {
      if(Def(p[i], input)) {
        satisfied[p[i]] = true;
      }
      else {
        satisfied[p[i]] = false;
        isSatisfied = false;
      }
    }
    else {
      isSatisfied = false;
    }
  }
  return isSatisfied;
}
```

(a)

```
pattern computeArgs(pattern input) {
  int n = level(input);
  pattern result = input;
  for(i = 0;;i++) {
    if(level(pattern[i])) > n; break;
    if(satisfied[pattern[i]]) {
      result = meet(result, pattern[i]);
    }
    else
    if(Match([pattern[i]], input)) {
      if(Def(pattern[i], input)) {
        result = meet(result, pattern[i]);
      }
    }
    if(result == bottom) return result;
  }
  return result;
}
```

(b)

Figure 5. (a) Determines if a DCG is satisfied. (b) Computes the arguments to the *F* function.

this set of patterns have their anti-level equal to all other contained pattern of *pi*, thus must have a join with *pe*.□

Corollary: Every DCG that has a non-strict c-pattern must be composed only of explicit patterns.

From the above corollary, the computation of a DCG is simplified, because there is no need to consider a DCG with a strict c-pattern. This happens since in order to satisfy such a DCG, all inputs must be at a defined state, and this situation already satisfies condition C1 of proposition 3. As a consequence, implicit patterns are not useful to represent non-strict actors under this definition method, therefore not allowed.

From the definition of a DCG, every clique of the pattern graph is a DCG, when only the *incomparable* edges are considered. For each PG, computing all cliques of the respective component of the pattern graph gives the list of DCGs. The obtained list of DCGs is ordered according to the relation of definition 7.

Every pattern is associated with the same *F* function. This is necessary in order to satisfy propositions 1 and 2. The parameters of the *F* function are computed when the conditions of proposition 3 are met. Figure 4 (b) shows a pseudocode for DCG-based output definition. The matrix *DGCs* contains for each PG, the ordered list of DCGs. Function *satisfied* (Figure 5 (a)) detects if the respective DCG is satisfied. The *computeArgs* (Figure 5 (b)) function is responsible for finding the common inputs to all satisfied patterns. These inputs are used as arguments to the designer supplied *F* function. The *noMoreSatisfiable()* function detects condition C3 of proposition 3.

6.4. Pattern-based definition

For this type of output definition method there is no need to compute DCGs. For a pattern *p*, all nodes of the pattern graph linked to *p* by *incomparable* edges will be contained in some DCG with *p*. Therefore, all of them must share the same *F* function with *p* (proposition 1). The fact that there can be only one *F* function per pattern establish a transitive closure on the pattern graph: if a pattern p_1 is incomparable to p_2 and p_2 is incomparable to p_3, then p_1 and p_3 must have the same *F* function.

```
if(PG == n) {
   if(satisfiedAny(clousure[0], cp) {
      F[0](meet[0]);
   }
   else
   ...
   else
   if(satisfiedAny(clousure[n], cp) {
      F[n](meet[n]);
   }
}
```

(a)

```
boolean satisfiedAny(patterns []p, pattern input) {
   for(i = 0;i < p.size;i++) {
      if(Match(p[i], input)) {
         if(Def(p[i], input)) {
            return true;
         }
      }
   }
   return false;
}
```

(b)

Figure 6. The matrix *closure* and arrays *F* and *meet* represents a transitive closure group of patterns, the associated *F* function and the computed meet, respectively.

The computation of the transitive closures is done by finding the connected components of the subgraph of the pattern graph associated with the respective PG, considering only *incomparable* edges. The same *F* function is assigned to all patterns of a component, and the parameters are computed based on the meet of all such patterns. The obtained transitive closures are ordered based on the *comparable* edges between the contained patterns (there can be no *incomparable* edges between closures). This order determines the sequence for testing the satisfaction of a pattern: if any pattern of a closure is satisfied, output can be defined.

The skeleton of code produced for pattern-based definition consists of: 1) as for the DCG-based skeleton, find what PG contains matched patterns; 2) respecting the transitive closure order, test patterns for satisfaction; 3) once a pattern is satisfied, execute the associated *F* function with the respective arguments. Figure 6 shows a pseudocode for pattern-based definition.

7. Case Study and Results

The expression language and code generation algorithms were implemented and successfully tested against a case study of an embedded control for a crane system []. The model for this system is composed of five actors and pattern-based definition was adequate for the implementation. The table below summarizes some facts about the model.

Actor	Inputs	Outputs	Patterns	Code-generated lines	User-supplied lines
Actor1	2	1	2	323	21
Actor2	4	2	2	355	250
Actor3	2	1	1	304	20
Actor4	2	1	1	304	38
Actor5	3	3	7	375	23

The columns give, respectively: actor name; number of input ports; number of output ports; number of patterns of the interface; number of lines of the final implementation that were automatically generated; number of lines of the final implementation that were user-supplied. As can be observed, the majority of the code was generated. The exception is *Actor2*, because it is where the control algorithm is implemented.

8. Final Remarks

In this work, I have developed a technique that enhances the applicability of the Synchronous Reactive model of computation to an actor-based language. Its major advantage is guaranteed generation of consistent actors with respect to the SR semantics. Therefore, it allows the designer to concentrate on capturing the functionality of the system under design, instead of being worried about respecting semantic restrictions of the model. It should also be noted that the method is independent of the language that implements each actor.

Future works includes the development and implementation of the causality analysis algorithms, study of scheduling algorithms from actors specified by patterns and definition of semantics for the interaction of other MoCs with the SR MoC.