The Journal of Supercomputing, 36, 51–81, 2006

Nautilus, a Concurrent Diagrammatic Specification and Programming Language

CLAUDIO NAOTO FUZITAKI fuzitaki@inf.ufrgs.br
PAULO BLAUTH MENEZES blauth@inf.ufrgs.br
JÚLIO PEREIRA MACHADO jhapm@inf.ufrgs.br
FERNANDO D'ANDREA dandrea@inf.ufrgs.br
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil

Abstract. Nautilus is a high-level specification and programming language having abstraction mechanisms not commonly found in other programming languages inspired by its semantic domain (a categorial model named Nonsequential Automata). It constitutes an elegant solution for concurrency and non-determinism as well as for synchronization of concurrent systems. The role as specification language highlights the diagrammatic syntax (it was originally text based).The diagrammatic syntax for Nautilus allows complete programs to be written using symbols and graphical diagrams. The graphical notation was elaborated in order to be able to express all the structures in the language, yet trying to improve the visualization of written programs. A brief comparison with UML is included. To support Nautilus as a programming language, a mapping to Java is constructed, setting the basis for an execution environment of Nautilus specifications.

Keywords: object language, concurrent systems, Nautilus, graphical notation, nonsequential automata

1. Introduction

Nautilus was originally introduced in [9, 14] and was inspired by the GNOME language [16], which in its turn was a simplified and revised version of the object oriented language OBLOG [18, 19]. Nautilus is a high-level object based and oriented language with abstraction facilities inspired by its semantic domain (a categorial model named Nonsequential Automata [10,14,15]). Among the main features of the language we highlight the following: concurrency and non-determinism—the language is naturally concurrent and non-deterministic; refinement—abstraction mechanism which implements the concept of transaction, in the sense that we can define high-level (finite and atomic) actions as the composition of other sequential and/or concurrent actions; aggregation—it allows the definition of combined behavior of a set of objects. An interesting feature is anticipation, that is emphasized in the example used to introduce the language.

The language naturally deals with concurrency and synchronization mechanisms and, as presented in [2], it could be used for teaching students how to deal with specifications of concurrent systems and was suggested as an academic language for teaching how to program such systems.

Another important feature is a diagrammatic syntax for Nautilus that was developed and presented in [3]. It provides a more intuitive, clear and organized way of building programs. The motivation was the advantages that a diagrammatic notation could bring to the language. A diagram is usually more intuitive and easy to learn than its textual

counterpart, specially when presenting composition operations like parallelism, aggregation and refinement. The notation was inspired by the graphical version of OBLOG and by the diagrams in Category Theory [1]. The choice for diagrams in Category Theory was influenced by the domain which gives semantics to the language. A first sketch for a integrated diagrammatic Nautilus environment were presented in [17].

To show how Nautilus can be used as a programming language, we present a mapping of Nautilus to Java. Different from a previous presented mapping [7], that aimed at constructing a translator for educational purposes, the present mapping explores more the concept of concurrency.

This paper is organized as follow: in Section 2 we briefly present the ideas behind Nautilus and its semantic domain; in Section 3 we develop our working example; Section 4 brings a discussion on diagrammatic Nautilus and UML diagrams (illustrating its use as specification language); Section 5, a mapping for the textual version of Nautilus to Java (showing its use as programming language); finally, we present some conclusions.

2. Nautilus and nonsequential automata

The focus of this article is the Nautilus language, not its semantic domain. But since its features were inspired by its semantic domain a brief introduction is made. For more details see [15]. A semantics for Nautilus is given by Nonsequential Automata—a domain with full concurrency where a class of morphisms stands for refinement. A refinement implements an automaton over sequential or concurrent computations of another, i.e., maps transitions into transactions reflecting the implementation of an automaton on top of another. Therefore, a refinement mapping is viewed as a special automaton morphism (a kind of implementation morphism [10]) where the target object is closed under computation, i.e., the target (more concrete) automaton is enriched with all the conceivable sequential and nonsequential computations that can be split into permutations of original transitions.

A nonsequential automaton is a special kind of automaton in which states and transitions posses a commutative monoidal structure. A structured transition specifies an independence or concurrency relationship between the component transitions. A structured state can be viewed as a ”bag” of local states where each local state can be regarded as a resource to be consumed or produced, like a token in Petri nets. This domain is based on labeled transition systems and inspired by Meseguer and Montanari’s work [13]. Adjunctions between models for concurrency were provided in [8–10] extending the approach of Winskel and Nielsen [20]. The steps of abstraction involved in moving between models have shown that nonsequential automata are more concrete than Petri nets. Moreover, categories of Petri Nets are indeed isomorphic to subcategories of nonsequential automata.

3. Nautilus as specification language

Since most languages for system specifications in the field of software engineering are graphical, the focus in this section will be the diagrammatic notation, but

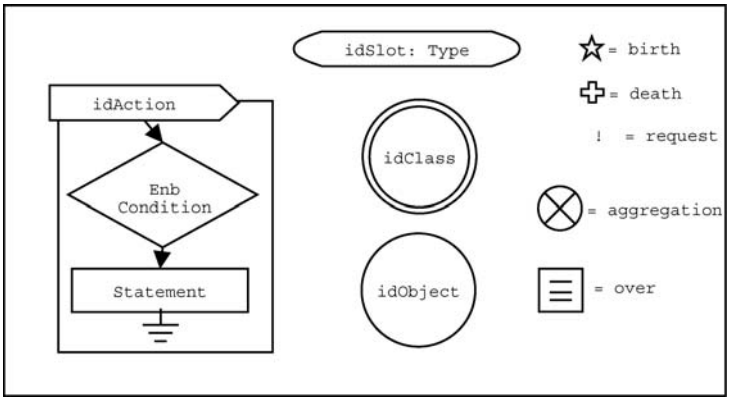

Figure 1. Basic notations for graphical specifications.

the textual counterpart will also be shown to facilitate the understanding of the next sections.

In this discussion of Nautilus we introduce some key words in order to help the understanding of the examples below. The specification of an object in Nautilus depends on if it is a simple object (whose construction does not depend on other objects) or a structured object resultant of an encapsulation, aggregation, refinement or parallel composition (which are constructions for building new objects over another objects). In any case, a specification has two main parts: interface and body. The interface declares the category (`category`) of some actions (`birth`, `death`, `request`). The body (`body`) declares the attributes (`slot`) and the methods of all actions (`act`). A birth or death action may occur at most one time (and determines the birth or death of the object), much like constructors and destructors in object-oriented languages. An action may have enabling (`enb`) conditions and also alternative execution bodies (`alt`). An action may be a sequential (`seq...end seq`) or multiple (`cps...end cps`) composition of clauses. A multiple composition is a special composition of concurrent clauses based on Dijkstra's guarded commands where the valuation (`val`) clauses are evaluated before the results are assigned to the corresponding slots. Several objects may be specified inside a unity (`spec...end spec`). Figure 1 shows some diagrammatic versions of the keywords (others can be infered in the example).

3.1. Graphical specifications

In this section we present an example depicting the features of Nautilus language. The example is presented in textual and in visual diagrammatic formats.

The example is called "The Carrot Principle" was used to introduce the anticipation features of the Nautilus language (mainly derived from the refinement construction `over`). In this example, borrowed from [4] and [11]:

> You like to go for a donkey ride. You sit on the donkey which does not want to go. So you present a carrot before it and then it goes to capture the carrot, but at the

same time the carrot goes also before it. The carrot is an 'anticipatory attractor' of the movement. To go to the left or to the right, you position the carrot to the left or to the right respectively: this defines a selection of one particular trajectory amongst all the potential possible subtrajectories.

This example depicts two kinds of anticipation: the attractor which is the motor of the action, and several selections of local anticipatory trajectories aiming at obtaining a global anticipatory trajectory or the final goal. We have specified in Nautilus other examples of objects for systems with anticipation behavior. We can mention among others which are not presented here due to limitation of space: two players competing against each other in a game, a simulation of a simple case of free-will, a system which tries to avoid unwanted future states, etc.

The Nautilus code for the "Carrot Principle" is shown in listings 1, 2, 3, 4 and the diagrammatic versions in Figures 2–5. As the purpose of this example is to explore the building blocks of the language, it is not necessarily the best example of specification one could built using Nautilus. It has two simple objects named *Donkey* and *Carrot*, which specifies the behavior of the basic objects that participate in our system. An aggregation of these two simple object is defined as the *Guided_Donkey* object, which behavior corresponds to the synchronization of *Donkey* and *Carrot* (it describes the "anticipatory attractor" of the donkey movement). Finally we define a possible anticipation between all the possible paths *Guided_Donkey* may fallow, thus building *Trained_Donkey* by the means if the refinement operator.

Listing 1: Textual specification of simple object *Donkey*

```
object Donkey
export
   New
   Left
   Right
   Walk
   Stop
category
   birth request New
body
   slot Direction: (N,S,E,W)
   slot Action: 0..1
   act New
      alt New1
          seq
          val Direction << N
          val Action << 0
          end seq
      alt New2
```

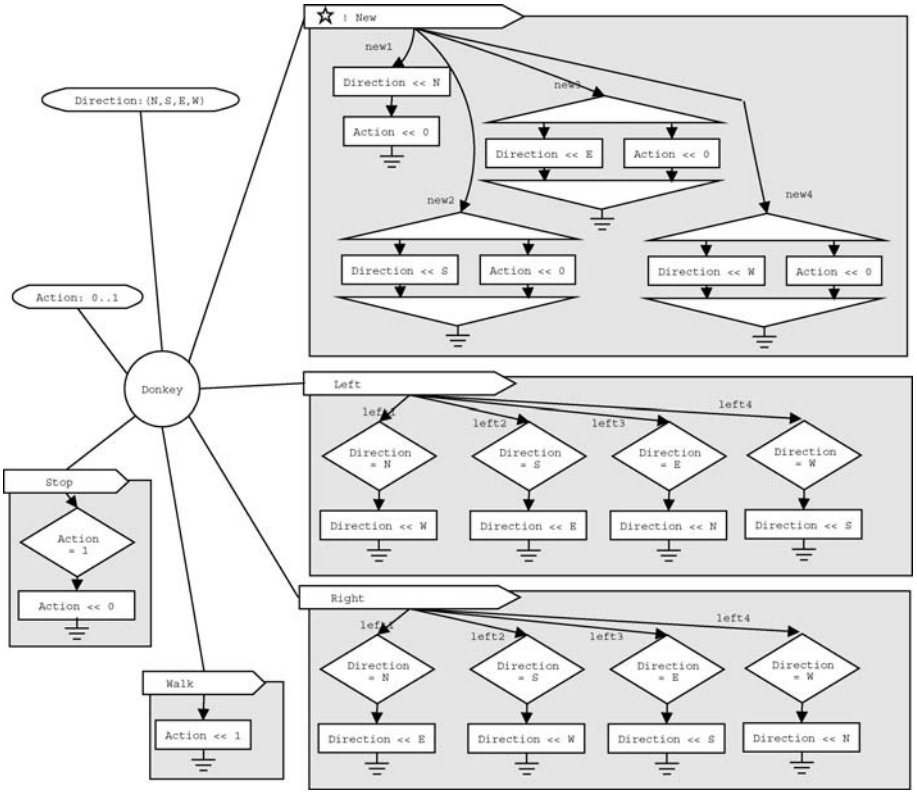

Figure 2. Diagrammatic specification of simple object *Donkey*.

```
        cps
        val Direction << S
        val Action << 0
        end cps
    alt New3
        cps
        val Direction << E
        val Action << 0
        end cps
    alt New4
        cps
        val Direction << W
        val Action << 0
        end cps
    act Left
        alt Left1
        enb Direction = N
```

```
            val Direction << W
            alt Left2
                enb Direction = S
                val Direction << E
            alt Left3
                enb Direction = E
            val Direction << N
            alt Left4
                enb Direction = W
                val Direction << S
        act Right
            alt Right1
                enb Direction = N
                val Direction << E
            alt Right2
                enb Direction = S
                val Direction << W
            alt Right3
                enb Direction = E
                val Direction << S
            alt Right4
                enb Direction = W
                val Direction << N
        act Walk
                val Action << 1
        act Stop
                enb Action = 1
                val Action << 0
end Donkey
```

Listing 2: Textual specification of simple object *Carrot*

```
object Carrot
export
    New
    Left
    Right
    Forward
    Hide
category
    birth request New
body
    slot Pos: (L,R,F,H)
    act New
```

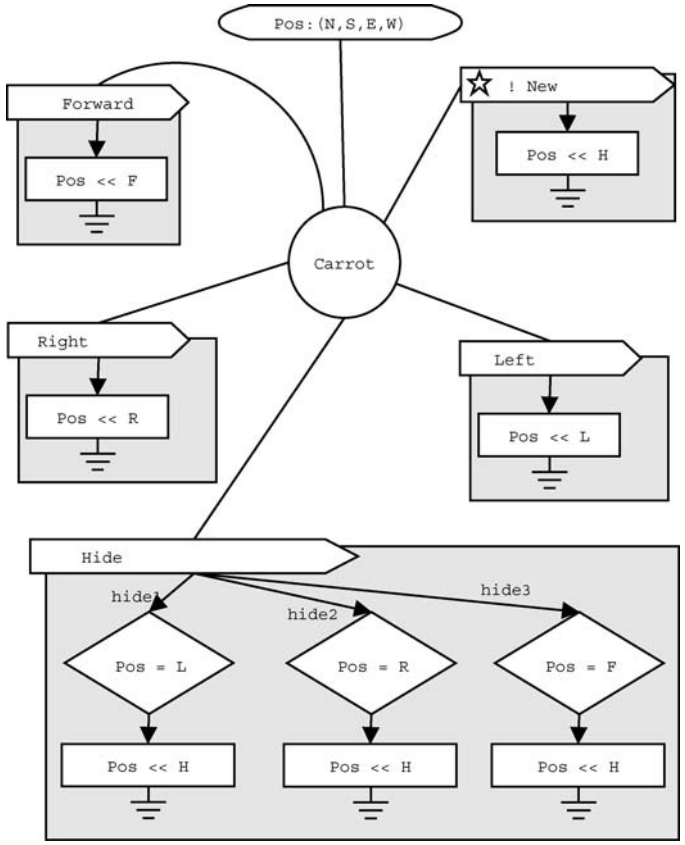

Figure 3. Diagrammatic specification of simple object *Carrot*.

```
    val Pos << H
act Left
    val Pos << L
act Right
    val Pos << R
act Forward
    val Pos << F
act Hide
      alt Hide1
         enb Pos = L
         val Pos << H
      alt Hide2
         enb Pos = R
         val Pos << H
      alt Hide3
```

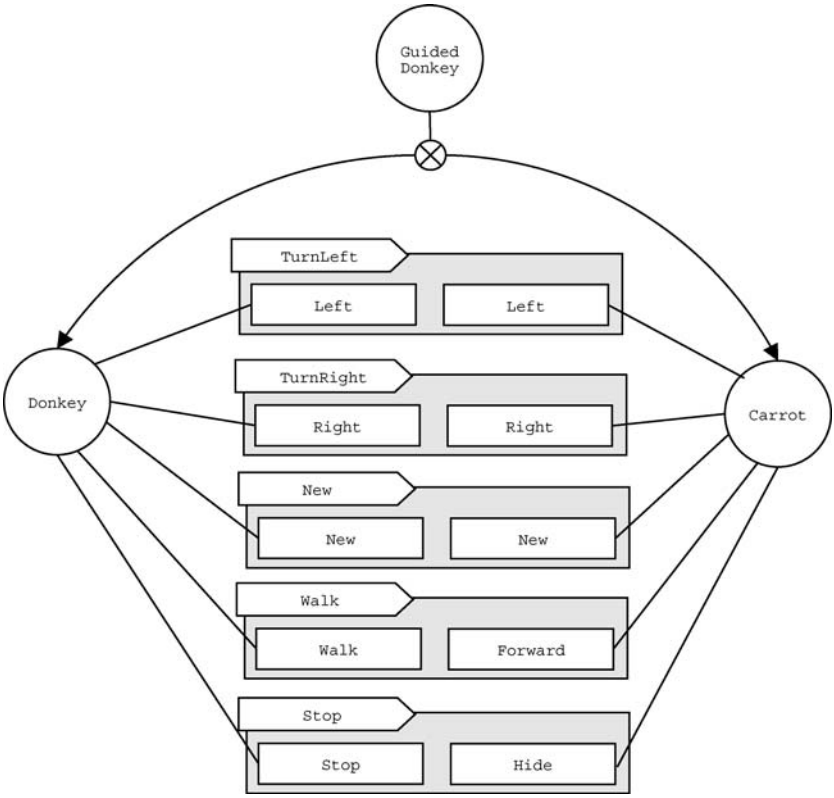

Figure 4. Diagrammatic specification of simple object *Guided_Donkey*.

```
            enb Pos = F
            val Pos << H
end Carrot
```

Listing 3: Textual specification of aggregation object *Guided_Donkey*

```
object Guided_Donkey
   aggregation of
         Donkey
         Carrot
export
   New
   TurnLeft
   TurnRight
   Walk
```

```
    Stop
category
    birth request New
body
    act New composed by
          New of Donkey
          New of Carrot
    act TurnLeft composed by
          Left of Donkey
          Left of Carrot
    act TurnRight composed by
          Right of Donkey
          Right of Carrot
    act Walk composed by
          Walk of Donkey
          Forward of Carrot
    act Stop composed by
          Stop of Donkey
          Hide of Carrot
end Guided_Donkey
```

Listing 4: Textual specification of anticipation object *Trained_Donkey*

```
object Trained_Donkey
over Guided_Donkey
export
    Born
    Go
    Return
category
    birth request Born
    request Go
    request Return
body
    act Born
            New
    act Go
            alt Go1
               seq
                 Walk
                 TurnLeft
                 Walk
                 Turn Right
```

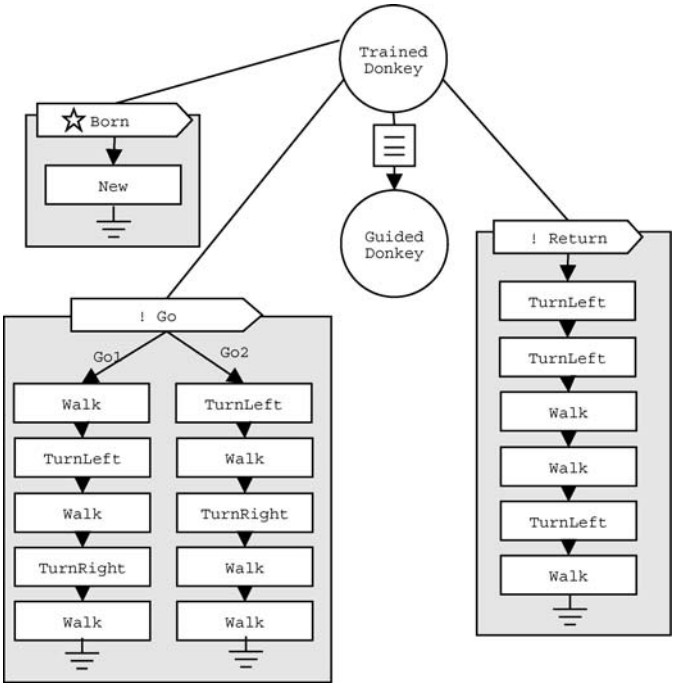

Figure 5. Diagrammatic specification of anticipation object *Trained_Donkey*.

```
                Walk
             end seq
          alt Go2
             seq
                TurnLeft
                Walk
                TurnRight
                Walk
                Walk
             end seq
   act Return
          seq
             TurnLeft
             TurnLeft
             Walk
             Walk
             TurnLeft
             Walk
          end seq
end Trained_Donkey
```

The *Donkey* is composed of five possible actions: *new* (causes the object to be born, as pointed by the `birth` clause), *left* (the donkey turns left), *right* (the donkey turns right), *walk* (the donkey goes forward), *stop* (the donkey stops moving). Those actions alter the internal state of the object, i.e., they change the values of the object `slots` which possible values are the cardinal points (N, S, E, W) for the *direction* slot, and *0* (donkey is not moving) or *1* (donkey is moving) for the *action* slot. The set of actions listed in the sentence `export` are the only actions that can be referenced by other objects. It is important to explain how actions are executed. Active actions are the ones for which every condition regarding to its requirements of usage are satisfied and so that every call can be executed. The selection of the activated action that will arise is an internal nondeterminism. Note that the birth action *new* has four alternatives. Both alternatives are always enabled, since they do not have enabling conditions. However, since it is a birth action, it occurs only once. For the purpose of presenting different clauses in Nautilus we have used the sequential (`seq`) composition of clauses in the alternative *new1* and the multiple (`cps`) composition in the other alternatives. This choice imposes no special meaning to the problem being modeled.

Aggregation consists in synchronizing two objects, therefore in a deeper level, synchronizing two automata. In the example, the object *Guided_Donkey* is implemented as the aggregation (`aggregation of`) of two other objects *Donkey* and *Carrot* respectively. The initially separated parts are now faced together forming an aggregate object that will specify the component objects behaviors. The semantics of an aggregation is the result of the synchronization of anticipations of nonsequential automata.

The structured object *Trained_Donkey* is constructed using the anticipation clause (`over`) in Nautilus. The anticipation constructor in Nautilus aims to set an object in function of the possible transactions from another object, which is called the anticipation base. Thus, the anticipation of an object is specified over an existing object (the *Trained_Donkey* is specified over the *Guided_Donkey*). An action may be anticipated into a complex action (a sequential or multiple composition of clauses) of the target object. In the example, the actions *Go* and *Return* are sequential compositions (`seq/end seq`) and the action *Born* is anticipated into a single action. Also, an action may be anticipated according to several alternatives, that is, an anticipation may be state dependent. In this sense, an action of the source object may have more then one implementation which may be explicit, i.e. alternatives are explicit in the source object (the action *Go* has two explicit alternatives) or implicit, i.e. actions in the target object used in an anticipation have alternatives (the action *New* has four alternatives as defined in the *Donkey* object). Note that anticipations are compositional and therefore, the target object of an anticipation may be the source of another anticipation.

3.2. Nautilus and UML

This section presents a brief comparison of the diagrammatic version of the language Nautilus and UML. The objective of this comparative study is to show the capacity of Nautilus of expressing systems that can be described in other modeling languages.

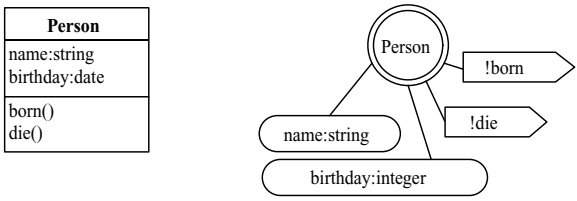

Figure 6. Class in UML and Nautilus.

UML was chosen because it is a widely accepted language for modeling systems. It is important to remember that the development method associated to UML is not in the scope of this comparison, only a fragment of the modeling language (mainly related to classes, objects and behavior) will be discussed. The comparative is organized by showing a construction in UML, and an equivalent or very similar one in Nautilus. Then, some comments will be done.

Most of the examples presented here are class diagrams of UML, as shown in [5]. In fact, UML uses many different and complementary kinds of diagrams. Some of the examples are from [5], and the considerations about UML are based on the same source.

3.2.1. Class diagrams. Class diagrams are one of the most basic kind of diagrams in UML, and they describe entities in the system like object types and static relationships between them [5]. In UML, classes (object types) are presented like having attributes and operations. Attributes are characteristics from the class. Operations from a class are linked to public methods in a object oriented language, and represent actions from an object of that type. For example, in Figure 6 all objects from class *Person* have attributes *name* and *birthday* and operations *born* and *die*.

These constructions are reproduced in Nautilus through classes, slots (attributes) and exported actions (operations). It is necessary to have in mind that, in Nautilus, a class or object does not allow others objects or classes to access directly the contents of its slots. An example is given in Figure 6: at left, we have a class in UML, and an equivalent Nautilus class at right.

Class diagrams in UML also express, in a very general form, association between classes. This information is usually given with a cardinality, like one or nothing (0..1), only one (1), zero or more (*) and one or more (1..*). Nautilus does not have facilities to show the cardinality of the relationship between classes. But, as a programming language, it can specify the relationship in a clear way and, if necessarily, in a more elaborated way. In Figure 7, we can observe the aggregation of classes *Person* and *License*. Similar effect could be obtained with a vision (a Nautilus construction used to encapsulate actions) of the interaction between the objects.

Another interesting UML construction that deserves attention is generalization. The idea of generalization appears when two classes with differences have also many similarities and these similarities can be united in a class that is a superclass of the formers. In order to use generalization, it is necessary not only similar attributes, but also that it can be said that any object of the descendant class is also an instance of the generalized class. An example are classes *Teacher* and *Student* that can be generalized in

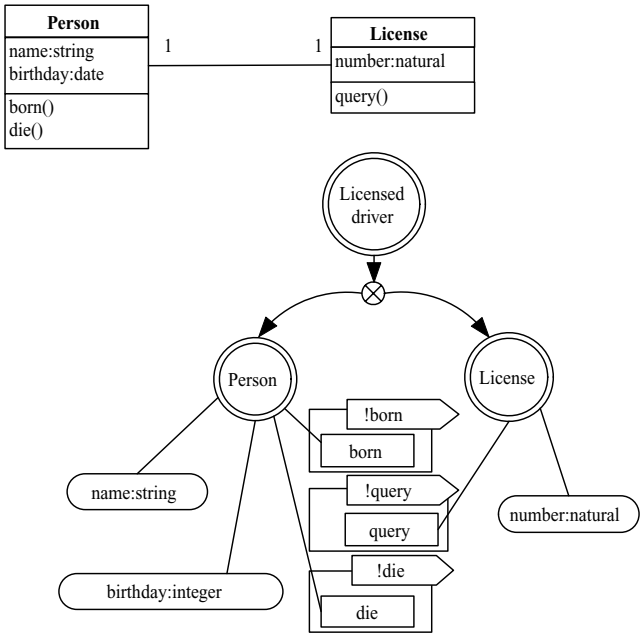

Figure 7. Class association in UML vs. a Nautilus construction.

the class *Person*: both have characteristics from the class *Person*, and, in fact, both are people.

In Figure 8 we can find the generalization described in UML and a similar construction in Nautilus. The idea in Nautilus is to create a general class, with the characteristics of both derived classes. Then, classes T and S are created, with the differences between them. So, we can extract the visions of the respective interactions. In this case, we have natural visions (in which all actions remain, just the object name is altered), so we will have classes with the complete interface of the children classes. We could use these new objects in any aggregation or reification or any other operation of Nautilus in the same way that we used the object *Person*, because they will have the same interface of person, but, with additions. These can be readily ignored in case of application of other operation, like a new vision (not natural), for example.

In UML there is the possibility of defining an association class. But, [5] shows that it is possible to transform an association class in a normal class between the two classes from the relationship. So, this facility can be used to construct the same specification in Nautilus.

UML class diagrams also allow the specification of aggregations and compositions. The notion of composition is a stringer form of aggregation, implying that component classes are deleted in cascade, if the composition is deleted. This is not needed in an aggregation. The UML concept of aggregation is very similar of aggregation in Nautilus. But here we find a problem from the fact of UML being a semi-formal notation. Martin Fowler says in ([5], p. 80) “Peter Coad gave an example of aggregation as the

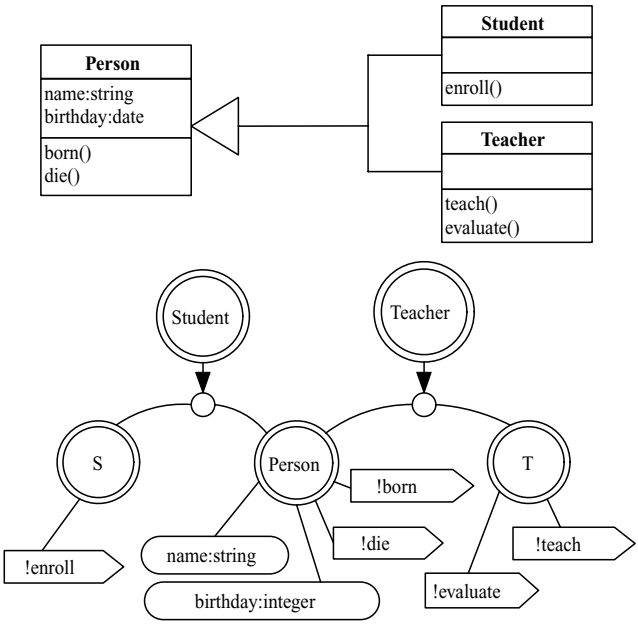

Figure 8. Class generalization in UML and a similar Nautilus construction.

relationship between an organization and its clerks; Jim Rumbaugh stated that a company is not an aggregation of its employees. When the gurus can't agree, what are we to do?" In that time, the concept of aggregation in UML was relatively elastic. Nautilus has a formal and not ambiguous semantics of aggregation that works like one of the interpretations of aggregation in UML. In fact, either in UML or in (a fragment of) Nautilus, we can express a car like an aggregation of motor, wheels and chassis, as in Figure 9.

3.2.2. Activity diagrams. Activity diagrams are one of the means for describing behavior of systems within the UML language and has been chosen here for its capability in describing flows of activities of a desired system with a notable similarity in the form of expressing activities with Nautilus.

Observe the example from ([5], page 130), in Figure 10. Horizontal black bars represent synchronization points, while rounded boxes represent activities states. Transitions between states are by means of arcs connecting the source and destination states.

A construction very similar in aspect and function can be constructed in Nautilus through a refinement. If we have an object in which all the activities that appears in Figure 10 exist as exported actions, we can construct a new object over that, and create an action that represents the activity. Figure 11 shows the actions of this new object.

While in UML, transitions may be adorned with a guard condition which is a boolean expression that is evaluated when a transition is triggered, Nautilus does not offer an

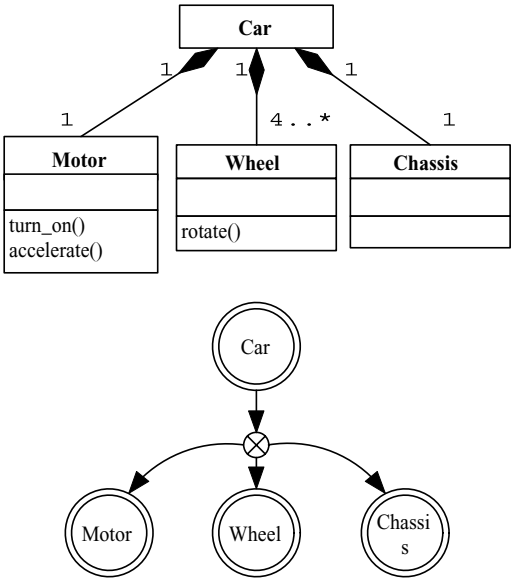

Figure 9. Aggregation in UML and Nautilus.

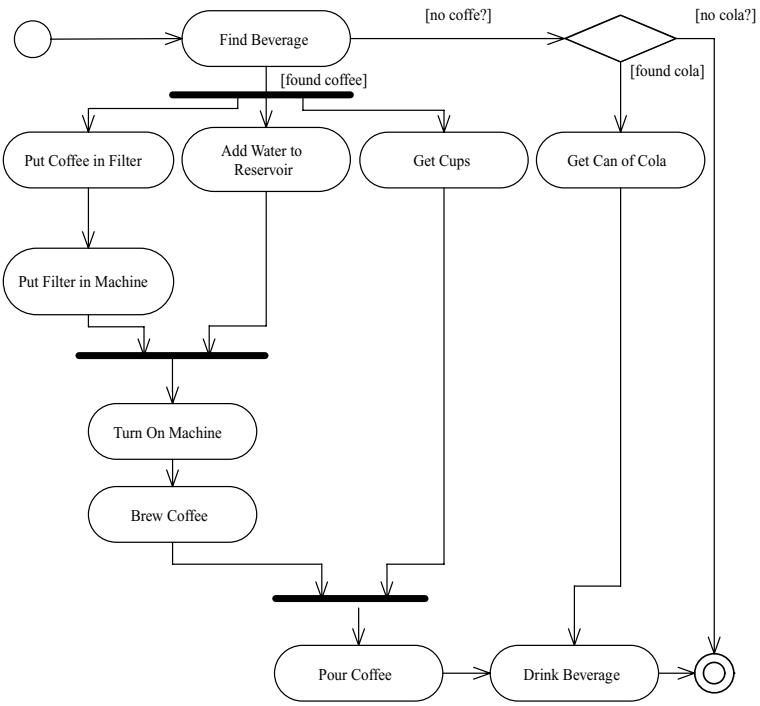

Figure 10. Activity diagram showing process concurrency.

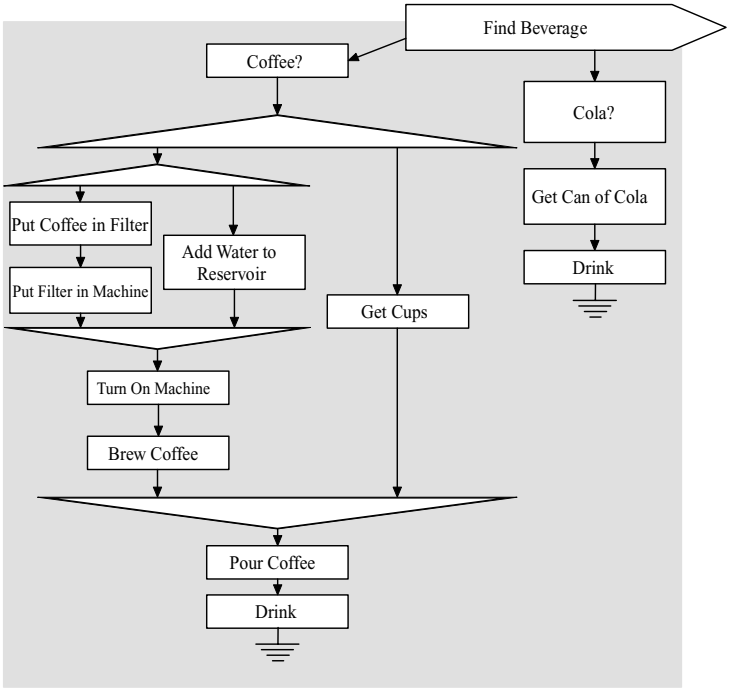

Figure 11. A refinement action with similar behavior to activity diagram.

option for setting attributes to activation conditions of a refined action, like in basic actions. It can be solved through an action in the base object of the refinement that does not have transactions inside it but only the desired activation condition, in our example actions `Cola?` and `Coffee?`. When these actions are used in the refinement, all refined alternatives will only be active if the component actions are active. The difference of this proposed refined action from the UML activity diagram in our example is that, for the case in which the coffee and coke are available, the person will always prefer coffee in case of the UML diagram. It is easy to program this behavior in a refined action. But, the action behavior was altered to show one of the characteristics of Nautilus, that is external non-determinism: if both coffee and coke are available, it is not known which one the person will choose. Such nondeterminism is not present in UML.

4. Nautilus as programming language

To show how Nautilus can be used as a programming language, we present a mapping of Nautilus to Java and one translation of the example from previous section. This section will use the textual notation, since the translation schemas are simpler in this

way. A first Nautilus-Java translation was presented in [7] and aimed at educational use. The translation presented here is aimed at parallelism and uses a greater number of threads.

4.1. Actions

In Nautilus the executable units are not objects but actions which are going to be mapped to methods in Java. Objects are only a way to organize actions that are self-executable by default, i.e. actions which are not tagged with the `request` clause. But the actions that will be executed are only the ones which are active, i.e., that do not have any kind of restriction (are not component of other actions, whose enabling condition is true, at least one alternative is active). So, if an action satisfies these conditions, besides the translation of its body, it is necessary to simulate the self-execution of the action.

Actions in Nautilus are naturally transactional and in case of fail should rollback to the previous state, just like in Database Management Systems. A scheme to translate Nautilus actions into Java methods is shown in Figure 12.

The translation is long because of the self execution nature of the action and the transactional structure of the actions (requiring locks, commits, rollback and bodies) and the need to support the construction of composite actions.

When the context makes the action no self executable `pA1` would not be used and composite actions would use `a1Body()` or `pA1Body` (an `a1body()` with an associated thread, see below) instead.

```
public class pA1Body extends Thread {
    public pA1(ThreadGroup tg, String s) {
        super(tg, s);
    }
    private VarInt s2;

    public Exception ep = null;

    public RightVars(VarInt _s2) {
        s2 = _s2;
    }
    public void run() {
        try {
            a1Body(s2);
        } catch (Exception e) {
            ep = e;
        }
    }
}
```

```
object o1
 ..
body
  slot s1, s2, .., s5:integer
  act a1
    enb ⟨cond⟩
    seq
      s1 << 1
      s2 << s2+1
    end seq
  ..
end o1
```

```
public class TO1 extends Thread {
  VarInt s1 = new VarInt();
  VarInt s2 = new VarInt();
  ..
  VarInt s5 = new VarInt();
  Boolean End = true;
  void a1Body(VarInt _s2) {
    s1.set(1);
    s2.set(_s2.get() + 1);
  }
  void a1() {
    try {
      // a1 Lock()
      s1.lock();
      s2.lock();
      if (!⟨cond⟩) throw
         new Exception("a1 enb false");
      a1Body(s1);
      // a1 Commit();
      s1.commit();
      s2.commit();
    } catch (Exception e) {
      // a1 Rollback()
      s1.rollback();
      s2.rollback();
    }
  }
  public class pA1 extends Thread {
    Boolean Autoexec;
    public pA1( Boolean b,
                ThreadGroup tg,
                String s) {
      super(tg, s);
      Autoexec = b;
    }
    public void run() {
      while (true) {
        if (End) break;
        a1();
        if (!Autoexec) break;
      }
   }
  }
  ..
}
```

Figure 12. A simple Nautilus action translated to Java.

The parameter in method `a1Body()` is necessary to represent a variable in the right side of assignment in the case of `cps`, which are explained in more details in section Refinement.

The auxiliary class `VarInt`, as defined below, is basically used to create a critical region where the internal value can be manipulated and then changes can be committed or restored (it is also used to handle `Null` cases). Each type (string, array, etc.) would be translated in a similar class (VarString, VarArray, etc.).

```
class VarInt {
      int old_value = 0, value = 0;
      boolean free = true;
      boolean old_isNull = true, isNull = true;

      synchronized void lock()
            throws InterruptedException
      {
         while (!free) { wait();}
         free = false;
      }

  void set(int value) {
      if (free) throw
         new Error("Trying to set without lock");
      this.value = value;
      this.isNull = false;
  }

  int get() {
      if (free) throw
         new Error("Trying to get without lock");
      if (isNull) throw
         new Error("Null value");
      return (value);
  }

  synchronized void commit() {
      old_value = value;
      old_isNull = isNull;
      free = true;
      notifyAll();
  }

  synchronized void rollback() {
      value = old_value;
      isNull = old_isNull;
      free = true;
      notifyAll();
  }
}
```

A `birth` action would be called in the start of the object's `run()` method. It would have a `Started = true` in the commit phase. While a `death` action would be similar to normal actions but having a `End = true` in the commit phase.

```
public void run() {
boolean Started = false;
while (!Started)
birth();

ThreadGroup myThreadGroup =
new ThreadGroup("My Group of Threads");

Thread a1 = new pA1(true,myThreadGroup,"a1");
a1.start();
..
Thread an = new pAn(true,myThreadGroup,"an");
an.start();
}
```

For the next constructions part of the code necessary to deal with exceptions and auxiliary functions may be omitted for simplicity.

4.2. Intra-action concurrency

Concurrency inside an action is expressed by keyword `cps` which are used to make concurrent assigns, like in Figure 13. It could be easily translated to Java using temporary variables and sequential execution.

It is also possible to create a thread to evaluate each right expression, but to optimize this behaviour it would be necessary to verify how complex an expression need to be for compensate the extra burden of creating threads.

```
slot
  S1, S2:⟨type⟩
act A1
  cps
    val S1 << ⟨expression1⟩
    val S2 << ⟨expression2⟩
  end cps
```

```
⟨type⟩ S1, S1', S2, S2', Temp;

void A1_body() {
    Temp = ⟨expression1⟩;
    S2 = ⟨expression2⟩;
    S1 = Temp;
}
```

Figure 13. Intra-action concurrency translation.

4.3. Nondeterminism

One very important characteristic of Nautilus is nondeterminism [12]. In Nautilus, an action may have alternative execution bodies (introduced by the keyword `alt`). To translate such nondeterminism, initially it was thought of using a choice operator that could be the built-in `random()` function. But it was perceived that this would not work in case of actions that are restricted by `request` or are used as a refinement component—in this cases the formal semantics demands that if there are alternatives that fail and others that succeed, the chosen one should be one that succeeds—so it was necessary to use another pattern where alternatives are tried until one of them succeeds. For example, an action `A` with alternatives `A1`,. . . ,`An` should be translated like shown in Figure 14.

4.4. Action with parameters

In case of actions with parameters (like in an aggregation) there are more complexities to consider. Parameters impose restrictions in the execution order of actions. The initial

```
act A
  alt A1
    ⟨body action A1⟩
  alt A2
    ⟨body action A2⟩
    .
    .
  alt AN
    ⟨body action AN⟩
```

```
void A_body(){
  Integer[] array = new Integer[N];
  int i = 0;
  for(i = 0; i < N; i++)
    array[i] = new Integer(i);
  //Shuffle the elements in the array
  Collections.shuffle(
    Arrays.asList(array));
  boolean success = false;
  i = 0;
  while(success == false && i < N){
    try{
      switch(array[i].intValue()){
      case 0:
        ⟨body action A1⟩
          break;
        ..
      case N−1 :
        ⟨body action AN⟩
          break;
      }
      success = true;
    }catch(Exception e){
      A_Rollback();
      i++;
      success = false;
    }
  }
}
```

Figure 14. Nondeterminism translation.

```
object Obj1
export
  A1
    in i:⟨type⟩
    out o:⟨type⟩
body
  ...
  act A1
    ⟨body action A1⟩
end Obj1

object Obj2
export
  A2
    in i:⟨type⟩
    out o:⟨type⟩
body
  ...
  act A2
    ⟨body action A2⟩
end Obj2

object ObjAg
aggregation of
  Obj1
  Obj2
...
body
  act AR composed by
    A1 of Obj1
    A2 of Obj2
   match
     A1.i of Obj1
     A2.o of Obj2
   match
     A2.i of Obj2
     A1.o of Obj1
end ObjAg
```

Figure 15. Parameter example.

and final parts of actions then become synchronization points where the actions receive and send actual parameters.

Every action that receives or returns parameters is implicitly of the type `request`, since it cannot self-execute alone. Parameters references are by identifiers, not by position and are possible only to exported actions (actions which are public to other objects).

An illustrative example is the program in Figure 15. In this example, objects `Obj1` and `Obj2` have actions with parameters, and a third object `ObjAg` is an aggregation acting as synchronization of inputs/outputs of the first objects.

Depending of the body of `A1` and `A2`, there are the following possibilities:

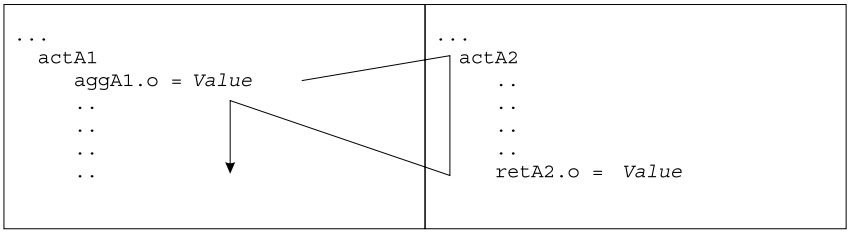

Figure 16. Sequential execution.

- Sequential Execution
  As shown in Figure 16, one of the actions supply the output parameter in the first instruction through the keyword `agg`, that is the input parameter of the second action, that now can execute and returns an output parameter that is used by the first action.
- Parallel execution
  When both actions supply parameters in the beginning, both can be executed. See Figure 17.
- No execution
  The no execution case means that actions are mutually dependent and do not occur (Figure 18). The semantic is empty, for all effects is like they have `enb false` clauses.

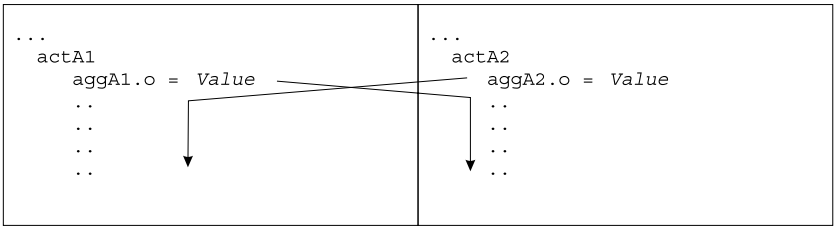

Figure 17. Parallel execution.

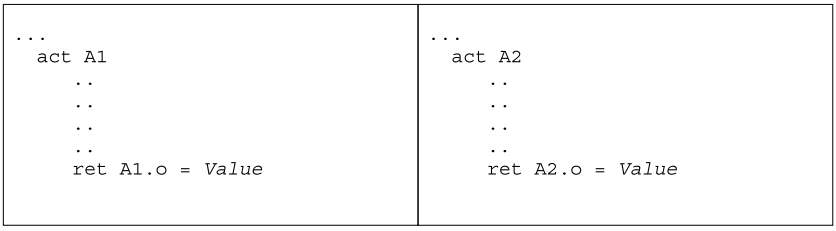

Figure 18. No execution.

Based on these cases it is necessary to make an analysis: which parameters are provided by `agg` clauses? If there is no action that could start just with these parameters, then it is a no execution case, otherwise choose any action that can be executed.

In the mapping presented in [7] the Java translation of Nautilus parameters was suggested as using "pseudo-slots". In this way each parameter would be a slot like "ActionId_Param_Id':TypeId", with the restriction that it cannot be used outside the correspondent action "ActionId" and that attribution is done only in `agg` and `ret` sections. That suggestion was adequate for that mapping where there was only a maximum of one thread per object. For the new mapping presented here, more is necessary to guarantee the semantics is respected. It is necessary to block a consumer thread until a producer send the parameter, and this can be attained by the following code.

```
class ParamInt {
private boolean IsNull = true;
private int Value = 0;

synchronized int getInt() throws InterruptedException {
while (IsNull) {
wait();
}
int r = Value;
IsNull = true;
notifyAll();
return r;
}

synchronized void setInt(int i) throws InterruptedException {
while (!IsNull) {
wait();
}
IsNull = false;
Value = i;
notifyAll();
}

}
```

The idea is the consumer thread waits until the producer thread supply a value. But there are more details to be considered. In the case of no-execution showed deadlock is not a valid interpretation of the semantics since could also lock slot variables making them unavailable. One way to implement no-execution is introducing a control thread that periodically verifies that the actions in the aggregation are not all locked, if they are the control thread should restore the values and restart the threads. But since the action threads involved in the no-execution case of aggregation would always deadlock,

```
object ObjBase
category
  birth Start
body
  ...
  act Start
    ⟨body action Start⟩
  act Action1
    ⟨body action Action1⟩
  act Action2
    ⟨body action Action2⟩
  act Action3
    ⟨body action Action3⟩
  ...
end ObjBase

object ObjRef
  over ObjBase
category
  birth Start
body
  act Start
     Start
  act ActionComp1
     seq
       Action1
       Action2
     end seq
end ObjRef
```

```
class TObjBase extends Thread{
  ...
  TObjBase(){
    super();
    Start();
  }
  void Start(){...}
  void Action1(⟨RightSideSlots⟩){...}
  void Action2(⟨RightSideSlots⟩){...}
  void Action3(⟨RightSideSlots⟩){...}
  ...
}
class TObjRef extends Thread {
  TObjBase ObjBase;
  TObjRef(){
    super();
    Start();
  }
  void Start(){
    ObjBase = new TObjBase();
  }
  void ActionComp1() {
    try {
      ⟨sequence of
         Action1 locks ∪ Action2 locks⟩
      ActionComp1_body();
      ⟨sequence of
         Action1 commits ∪ Action2 commits⟩
    } catch(Exception e) {
      ⟨sequence of
         Action1 rollbacks ∪ Action2 rollbacks⟩
    }
  }
  void ActionComp1_body(){
    ObjBase.Action1_body(⟨RightSideSlots⟩);
    ObjBase.Action2_body(⟨RightSideSlots⟩);
  }
}
```

Figure 19. Refinement.

we proposed a simple protocol checker for the Nautilus parameters in [6] that would eliminate this pathological case in static analysis.

4.5. Refinement

Refinement is an implementation of an object over another. An action of a refinement object is constructed over the actions of a base object. Its the only construction that imposes an order, respecting sequentiality, concurrency and nondeterminism. It can be translated into Java as method calls with a transaction restriction (Figure 19).

Now the necessity of $\langle RightSideSlots \rangle$ will be explained.

Supose that

```
act Action1
  s1 << s2

act Action2
  s2 << s1
```

with the translation

```
void Action1_body(VarInt _s2){
  s1.set(_s2.get());
}

void Action2_body(VarInt _s1){
  s2.set(_s1.get());
}
```

then the translation of

```
act ActionComp1
  seq
     Action1
     Action2
  end seq
```

is

```
void ActionComp_Body(){
  Action1_body(ObjBase.s2);
  Action2_body(ObjBase.s1);
}
```

But if Action Comp1 was a compose (cps) clause

```
act ActionComp1
    cps
        Action1
        Action2
    end cps
```

then the translation would be

```
void ActionComp_Body(){
  VarInt s1 = ObjBase.s1.clone();
  VarInt s2 = ObjBase.s2.clone();
  Action1_body(s2);
  Action2_body(s1);
}
```

And Action1_body, Action2_body could also been executed in parallel.

```
void ActionComp_Body() {
  ThreadGroup tgAc =
      new ThreadGroup("Ac");
  VarInt s1 = ObjBase.s1.clone();
  VarInt s2 = ObjBase.s2.clone();
  ObjBase.pActionComp_Body a1 =
      new ObjBase.pActionComp_Body(tgAc,"WriteTic");
  a1.RightVars(s2);
  a1.start();
  ObjBase.pActionComp_Body a2 =
      new ObjBase.pActionComp_Body(tgAc,"WriteTac");
  a2.RightVars(s1);
  a2.start();
  a1.join();
  a2.join();

  if ((a1.ep != null) || (a2.ep != null)) {
      if ((a1.ep == null) { throw a2.ep; }
          else { throw a1.ep; }
      }
}
```

The notation $\langle sequence\ of\ Action1\ rollbacks \cup Action2\ rollbacks \rangle$ means that if $\langle Action1\ rollbacks \rangle$ is {s1.rollback(), s2.rollback()} and $\langle Action2\ rollbacks \rangle$ is {s2.rollback(), s3.rollback()}, then $\langle Action1\ rollbacks \cup Action2\ rollbacks \rangle$ is

{s1.rollback(), s2.rollback(), s3.rollback()}. And *sequence of* organizes the elements of the set in a canonical sequence.

4.6. Aggregation

Aggregation is one of the main composition mechanisms of Nautilus. Each object can be viewed as a system and the aggregation establishes some synchronization points among the aggregated objects. A simple example is shown in Figure 20.

In this case aggregation takes off the possibility of an action to self execute, but if, for example `Tac` have an action called `WriteTac2` and no reference to this action in the object `TicTac`, this action could self execute, but it would be inaccessible (the aggregation would hide it). And the `switch` of the `run()` method would have a `case 1: Tac.WriteTac2()`.

In this simple example there is no imposed sequentiality of how the component actions would have to execute, but in case of aggregation using actions with parameters there are more complexities as shown in Section 4.4.

5. Conclusion/future works

In this article, we presented an overview of Nautilus language as specification language; and to show its possibilities as a programming language, we presented a new Nautilus-Java mapping.

Since Nautilus has abstraction mechanisms not commonly found in other programming languages inspired by its semantic domain (a categorial model named Nonsequential Automata), we made a brief introduction of its characteristics.

The original definition of Nautilus dates back to the sequence of papers [9, 14]. Since then, most discussion have concentrated on the semantics of concurrency and refinement and little has been explored on the software environment for supporting the language. The paper presented here aimed at filling this gap—the diagrammatic notation and mapping to Java are the basis for the development of an adequate software environment (like translator and interpreter) for Nautilus.

A problem with its use as programming language is that the originally defined computational core is very limited. For example, basic actions don't present any loop command or recursion (you must explore auto execution to obtain a loop effect). So, while a Turing complete language, in its current state is much like a coordination language defining possible interactions between components. For usability sake we are studying forms to add constructions in a way that respects the constraints from the semantic domain (transactional actions).

About distributed computing, currently the language doesn't have location primitives, but in principle a future implementation could easily incorporate annotations for starting top-level objects in different machines, obtaining a limited form of distributed computing. Internal objects would present some difficulties, while components of an aggregation could easily be distributed, components of a refinement could not. An aspect that will have to be considered in a distributed environment is the computational cost of transactions.

```
object Tic
category
  birth Start
  export WriteTic
body
  ...
  act Start
    ⟨body action Start⟩
  act WriteTic
    ⟨body action WriteTic⟩
end Tic

object Tac
category
  birth Start
  export WriteTac
body
  ...
  act Start
    ⟨body action Start⟩
  act WriteTac
    ⟨body action WriteTac⟩
end Tac

object TicTac
aggregation of
  Tic
  Tac
category
  birth Start
body
  act Start composed by
     Start of Tic
     Start of Tac
  act WriteTicTac composed by
     WriteTic of Tic
     WriteTac of Tac
end TicTac
```

```
class TTic{ ... }
class TTac{ ... }
class TTicTac extends Thread {
  TTic tic;
  TTac tac;
  TTicTac() {
    super();
    Start();
  }
  void Start() {
    tic = new TTic();
    tac = new TTac();
  }
  void WriteTicTac_Body() {
    ThreadGroup tgWTT =
      new ThreadGroup(
            "WriteTicTac");
    Tic.pWriteTic t1 =
      new Tic.pWriteTic(
            tgWTT,"WriteTic");
    t1.start();
    Tac.pWriteTac t2 =
      new Tac.pWriteTac(
            tgWTT,"WriteTac");
    t2.start();
    t1.join();
    t2.join();
    if ((t1.ep != null)
         ||(t2.ep != null)) {
      if ((t1.ep == null) {
        throw t2.ep; }
      else {
        throw t1.ep; }
    }
  }
  void WriteTicTac() {
    try {
      ⟨sequence of
       Tic locks ∪ Tac locks⟩
      WriteTicTac_Body();
      ⟨sequence of
       Tic commits ∪ Tac commits⟩
    } catch (Exception e) {
      ⟨sequence of
       Tic rollbacks ∪ Tac rollbackss⟩
     }
  }
  ..
}
```

Figure 20. Aggregation.

The role as specification language emphasized the diagrammatic syntax and was presented using a example showing some anticipatory properties of the language. Also a brief comparison with UML was included. The presentation of this Nautilus-Java mapping is novel. Different from previous presented mapping that aimed at constructing a translator for educational use, this mapping explores more concurrency.

Acknowledgments

This work was partially supported by CNPq (Projects HoVer-CAM, GRAPHIT, E-Automaton) and FINEP/CNPq(Project Hyper-Seed) in Brazil.

References

1. M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.
2. C. Carneiro, T. Veit, P. B. Menezes, and F. D'Andrea. Nautilus: its concurrent and distributed characteristics as an academic language. In H.R. Arabnia, ed., *Proc. of International Conf. on Parallel and Distributed Processing Techniques and Application*. CSREA, Vol. 4, pp. 1919–1925, 1999.
3. F. D'Andrea, P. B. Menezes, C. N. Fuzitaki, J. P. Machado, and S. A. da Costa. Nautilus, a diagrammatic specification and programming language. In S.G. Akl, T. Gonzalez, eds., *Proc. of 14th International Conference on Parallel and Distributed Computing and Systems*, ACTA Press pp. 386–391, 2002.
4. D. M. Dubois. Review of incursive, hyperincursive and anticipatory systems. In D.M. Dubois, ed., *Proc. of 3th International Conference on Computing Anticipatory Systems*. Volume 517 of AIP Conference Proceedings., Melville, American Institute of Physics, pp. 3–30, 2000.
5. M. Fowler and S. Kendall. *UML Distilled*. Addison-Wesley, 1997.
6. C. N. Fuzitaki, P. B. Menezes, and J. P. Machado. A protocol checker for nautilus language. In H.R. Arabnia, ed., *Proc. of International Conf. on Parallel and Distributed Processing Techniques and Application*. CSREA, Vol. 3, pp. 1336–1341, 2004.
7. C. N. Fuzitaki, P. B. Menezes, J. P. Machado, and S. A. da Costa, Mapping nautilus language into java: Towards a specification and programming environment for distributed systems. In R.M. Diaz, F. Pichler, eds., *Proc. of 9th International Conference on Computer Aided Systems Theory and Technology*. Springer-Verlag, Volume 2809 of Lecture Notes in Computer Science., pp. 243–252, 2003.
8. P. B. Menezes. Compositional reification of concurrent, interacting systems. In H.R. Arabnia, ed., In *Proc. of International Conf. on Parallel and Distributed Processing Techniques and Application*. CSREA, Vol. 4, pp. 1754–1761, 1998.
9. P. B. Menezes and J. F. Costa. Compositional reification of concurrent systems. *Journal of the Brazilian Computer Society*, 2:50–67, 1995.
10. P. B. Menezes, J. F. Costa, and A. Sernadas. Refinement mapping for general (discrete event) system theory. In F. Pichler, R.M. Diaz, R. Albrecht, eds., *Proc. of 5th International Conference on Computer Aided Systems Theory and Technology*. Volume 1030 of Lecture Notes in Computer Science., Springer-Verlag, pp. 103–116, 1996.
11. P. B. Menezes, S. A. da Costa, J. P. Machado, and J. Ramos. Nautilus: a concurrent anticipatory programming language. In D.M. Dubois, ed., *Proc. of 5th International Conference on Computing Anticipatory Systems*. Volume 627 of AIP Conference Proceedings., Melville, American Institute of Physics, pp. 553–564, 2002.
12. P. B. Menezes, J. P. Machado, and S. A. da Costa. Explicit and implicit nondeterministic refinement for concurrent, interacting systems. In H.R. Arabnia, ed., *Proc. of International Conf. on Parallel and Distributed Processing Techniques and Application*, CSREA, 2002.
13. J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation* 88; pp. 105–155, 1990.
14. P. B. Menezes, A. Sernadas, and J. F. Costa. Refinement in a concurrent object-based language. In R. da Silva Bigonha, ed., *Proc. of I Brazilian Symposium on Programming Language*. SBC, Vol. 1., pp. 237–250, 1996.

15. P. B. Menezes, J. F. Costa, and A. Sernadas, Nonsequential automata semantics for concurrent, object-based language. In R. Cleaveland, M.W. Mislove, P.S. Mulry, eds., *Proc. of 2nd US-Brazil Joint Workshops on the Formal Foundations of Software Systems*. Volume 14 of Electronic Notes in Theoretical Computer Science., Elsevier, 1998.
16. J. Ramos and A. Sernadas. A brief introduction to gnome. Technical Report, Instituto Superior Técnico, Lisboa, 1995.
17. E. F. Seganfredo, R. Gatto, C. N. Fuzitaki, P. B. Menezes, and D. J. Nunes. An outline to a diagrammatic nautilus environment. In H.R. Arabnia, ed., in *Proc. of International Conf. on Parallel and Distributed Processing Techniques and Application*. CSREA, Vol. 4., pp. 1726–1731, 2003.
18. C. Sernadas, P. Gouveia, and A. Sernadas. Oblog: Object-oriented, logic-based conceptual modeling. Technical Report. Instituto Superior Técnico, Lisboa, 1992.
19. C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large object-oriented design of information systems. In F. V. Assche, B. Moulin, C. Rolland, eds., *The Object-Oriented Approach in Information Systems*. North-Holland, pp. 209–232, 1991.
20. G. Winskel and M. Nielsen. Models for Concurrency. In: Handbook of Logic in Computer Science. Oxford University Press, Vol. 4, pp. 1–148, 1995.