Anemone: a Workbench for the Multi-Bach Coordination Language Extended version

Jean-Marie Jacquet[a,*], Manel Barkallah[a]

[a]*Nadi Research Institute, Faculty of Computer Science, University of Namur*
Rue Grandgagnage 21, Namur, Belgium

Abstract

Although many research efforts have been spent on the theory and implementation of data-based coordination languages, not much effort has been devoted to constructing programming environments to analyze and reason over programs written in these languages. This paper proposes a workbench for describing concurrent systems using a Linda-like language, for animating them and for reasoning over them using a reachability logic.

Keywords: coordination languages, Bach, animation, verification

1. Introduction

In order to ease the development of interactive and distributed systems, Gelernter and Carriero have advocated in [1] to clearly separate the interactional and the computational aspects of software components. Their claim has been supported by the design of a model, Linda [2], originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

A number of other models, now referred to as coordination models, have been proposed afterwards (see e.g. [3, 4, 5, 6, 7, 8, 9, 10]). However, although

*Corresponding author

Email addresses: jean-marie.jacquet@unamur.be (Jean-Marie Jacquet), manel.barkallah@unamur.be (Manel Barkallah)

many pieces of work have been devoted to the proposal of new languages, semantics and implementations, just a few articles (e.g. [11, 12, 13, 14]) have addressed the concerns of constructing programs in coordination languages in practice, in particular in checking that what is described by programs actually corresponds to what has to be modeled. This article aims at addressing this need by introducing a workbench, named Anemone, to reason over Linda-like languages. More specifically, our goal is threefold:

- to allow the user to understand the meaning of instructions written in Linda-like languages, by showing how they can be executed step by step and how the contents of the shared space, central to coordination languages, can be modified so as to release suspended processes;
- to allow the user to better grasp the modeling of real-life systems, by connecting agents to animations representing the evolution of the modeled system;
- to allow the user to check properties by model checking reachability formulae and by producing traces that can be replayed as evidences of the establishment of the formulae.

In building the workbench, we also aim at two main properties:

- the tool should be simple to deploy and to use. As a result, we have built it as a standalone executable file. We also propose a process algebra, named Multi-Bach[1], that allows the user to concentrate on the key coordination and animation features and consequently avoid the burden of handling other features such as those induced by the integration of the code in a host language;
- the tool should maintain a direct relation between what is written by the user and its internal representation. This property allows the user to better grasp what is actually computed as well as to produce meaningful traces.

[1]This name comes from the fact that the language includes *multiple* versions of the tell, ask, nask and get primitives of the Bach coordination language

Figure 1: Rush Hour Problem. On the left part, the game as illustrated at https://www.michaelfogleman.com/rush. On the right part, the game modeled as a grid of 6×6, with cars and trucks depicted as rectangles of different colors.

To make the paper more concrete, we use the Anemone workbench on a running example, the rush hour puzzle. This game, illustrated in Figure 1, consists in moving cars and trucks on a 6×6 grid, according to their direction, such that the red car can exit. It can be formulated as a coordination problem by considering cars and trucks as autonomous agents which have to coordinate on the basis of free places.

This work builds upon previous work by the authors on coordination languages, in particular on [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]. In these pieces of work, a Linda-like dialect has been proposed under the name of Bach. It is enriched in this paper by primitives necessary to model and animate systems.

This article is also an extension of a preliminary work [26] presented at the Coordination'19 conference. Although the research goals are the same, several features have been revised and new ones have been included. At the process algebra level, we have introduced variables, generalized choices, primitives to handle processes as active data, primitives to manipulate blackboard rules and have revised the description of scenes, allowing now for multiple scenes and using widgets as main elements of scenes. We have also refined the workbench to account for these new features as well as to provide new facilities. The interactive blackboard now incorporates a filter button to enable the user to select elements of the shared space. As parallels to the interactive and autonomous agent windows, two new windows are proposed to tackle the creation of new processes induced by their view as active data. Moreover a new kind of window is proposed to handle rules. The resulting

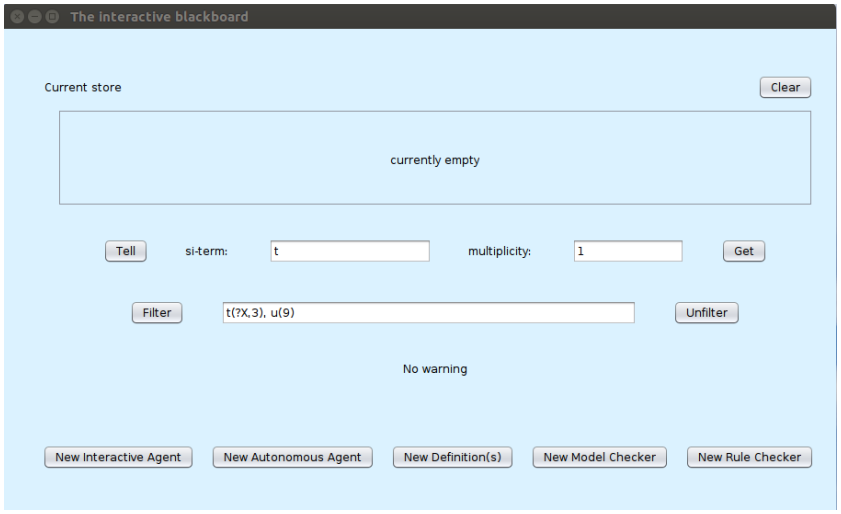

Figure 2: The interactive blackboard window

workbench is available at `https://github.com/UNamurCSFaculty/anemone` together with examples, videos, tutorial and documentation.

The rest of this paper is organized as follows. Section 2 describes the main functionalities of Anemone and, in doing so, provides an overview of the workbench. Section 3 specifies the coordination language and temporal logic to be used in the workbench. In this light, Section 4 revisits the workbench and shows how features not covered in the overview of the workbench are actually handled. Section 5 sketches how Anemone is implemented. Section 6 compares our work with related work. Finally, Section 7 draws our conclusion and suggests future work.

2. The Anemone workbench in a snapshot

Following Linda, the Multi-Bach language relies on a shared space to coordinate processes. It is this space that provides the decoupling of time and space of processes which is central to so-called data-based coordination languages [27]. As a natural consequence, following the blackboard metaphor [28], according to which a group of specialists iteratively updates knowledge on a blackboard starting from a problem specification, Anemone is articulated around a so-called interactive blackboard. As depicted in Figure 2, it starts by displaying the current contents of the shared space and allows to inter-

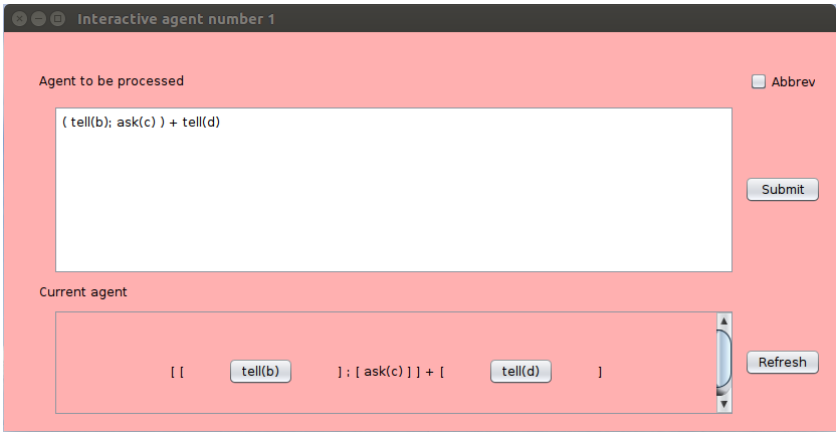

Figure 3: Interacting with the blackboard

act directly through the `tell`, `get`, `filter`, `unfilter` and `clear` buttons. Moreover, it offers to create five types of processes.

The first two processes, named respectively *Interactive Agent* and *Autonomous Agent*, allow the user to enter instructions in Multi-Bach and to execute them. As depicted in Figure 3, windows of the first kind perform computations step-by-step by letting the user choose which primitives to execute. In contrast, windows of the second type execute computations in one run or in a step-by-step manner but in both cases with the Anemone workbench deciding the primitives to be executed. It is worth noting that the execution in the windows are made in a parallel fashion, hence the name *agent* to indicate entities capable of concurrent activities. In both cases, the `abbrev` option provides a convenience to abbreviate complex instructions by hiding the non-executable part.

The facilities offered by the interactive and autonomous agents are well-suited to debug, at a low level, concurrent executions executed over the shared space, possibly deadlocking while waiting for unavailable data. However, they do not provide much insights on whether what is described in Multi-Bach really reflects what the programmer intends to model. Moreover, they provide too many details on the main execution steps leading to a solution of the problem under consideration. To that end, Anemone provides the possibility of specifying various features through a third kind of window launched by the `New Definition(s)` button (see Figure 2). Such definitions

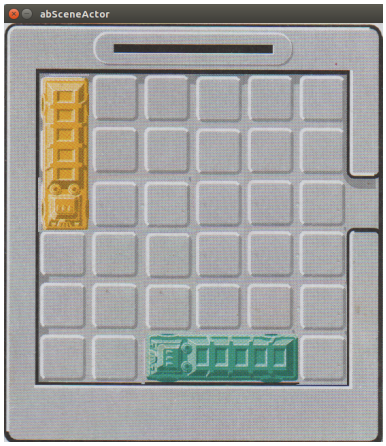

Figure 4: Animation

of features include the definitions of sets, equations, procedures, rules, scenes and widgets, which constitute several aspects of Multi-Bach, as described in the subsections of Section 3. In particular, widgets are manipulated during the execution of the program by means of primitives for inserting them on the scene at specific places, for making them visible or invisible, and for making them move to specific places. In doing so, these primitives allow to draw and animate, at a high-level, pictures such as the one of Figure 4. Note that, as these primitives can be inserted inside instructions of autonomous agents, the concurrent execution of these agents provides dynamic simulations of the problem under consideration.

Simulating graphically systems does not however necessarily provide a solution to the problem under consideration. The rush hour problem is a clear example. To that end, the Anemone workbench offers a fourth type of window, launched by the `New Model Checker` button of Figure 2. Such a window allows to verify formulae written in a reachability logic, to determine traces of execution that establish the formulae and to replay these traces, including the primitives that generate animations.

Finally, a fifth kind of windows is provided through the `New Rule Checher` button to allow the user to reason on rules (as will be specified later).

Although designed and implemented as a simple piece of software, we believe that the Anemone workbench meets the threefold goal expressed in the introduction:

- by providing a view on the contents of the shared space and by means of the interactive and autonomous agents, the user can better understand the execution of programs written in Multi-Bach;
- the animation facilities provide a high-level view on what is actually computed as well as an intuitive perception of the modeling of the problem under consideration;
- the model checker facilities allow to check properties and, by using animation facilities, to replay executions graphically as a form of visual proofs.

As side effects, thanks to the animation facilities, the Anemone workbench can be used to ease the communication between business stakeholders and developpers. It can also be used for educational purposes to let students or even developpers learn how agent systems can be encoded with coordination languages.

It is worth stressing that the Anemone workbench is currently more in the state of a proof-of-concept prototype than a well-polished commercial tool. In particular, it relies on a home-made model checker which can certainly be improved. It also assumes that the user enters syntactically correct programs and it does not provide auto-correction or auto-completion facilities offered by state-of-the-art IDE's. Still Anemone can be used in simple situations and opens ways of future research to tackle more complex cases.

3. The Multi-Bach language and its temporal logic

Let us now turn to the process algebra supporting Anemone.

3.1. Definition of data

Following Linda, the Bach language [20, 29] uses four primitives for manipulating pieces of information: *tell* to put a piece of information on a shared space, *ask* to check its presence, *nask* to check its absence and *get* to check its presence and remove one occurrence. In its simplest version, named BachT, pieces of information consist of atomic tokens and the shared space, called the store, amounts to a multiset of tokens. Although in principle sufficient to code many applications, this is however too elementary in practice to code them easily. To that end, we introduce more structured pieces of information which may employ sets defined by the user. Concretely, such

sets are defined by associating an identifier with an enumeration of elements, such as in

```
eset RCInt = { 1, 2, 3, 4, 5, 6 }.
```

Formulated in the context of the rush-hour problem, this set allows to identify an element of the grid by using the row and column coordinates. We shall subsequently take the convention that the upper leftmost element of the grid is on the first row and on the first column.

It is worth noting that, for verification purposes, we deliberately restrict set definitions to finite sets. This limitation is of little concern in practice as most problems can be formulated by using finite sets. Moreover, we take profit of the enumeration of elements to induce an order between elements, thus implicitly defining the equality and inequalities relations. Strictly speaking our set declarations thus define more than sets but also orders. Hence the `eset` tag to denote enumerated sets.

In addition to sets, maps can be defined as functions between sets that take zero or more arguments. In practice, Anemone uses mapping equations as rewriting rules, from left to right in the aim of progressively reducing a complex map expression into a set element.

As an example, in the rush hour example, assuming that trucks take three cells and are identified by the upper and left-most cell they occupy, the operation `down_truck` determines the cell to be taken by a truck moving down:

```
map down_truck: RCInt -> RCInt.
eqn down_truck(1) = 4. down_truck(2) = 5. down_truck(3) = 6.
```

Note from this example that mappings may be partially defined, with the responsibility put on the programmer to use them only when defined.

Structured pieces of information to be placed on the store consist of flat tokens as well as expressions of the form $f(a_1, \cdots, a_n)$ where f is a functor and a_1, ..., a_n are set elements. Structured pieces of information are subsequently referred to as *si-terms* as well. As an example, in the rush hour example, free places of the game are represented by the si-terms `free(i,j)` with i a row and j a column.

It is convenient to introduce partially defined fields in si-terms. This is achieved by assuming a set of variables, say $\mathcal{V}$, and by inserting variables as arguments a_i of si-terms of the form $f(a_1, \cdots, a_n)$. For convenience, variables are sometimes syntactically denoted by prefixing them with an interrogation

mark, as in Figure 2. We also subsequently note them by identifiers ranging in sets in the generalized choices and rules introduced in the next subsection.

Note that, to force communication to occur by means of the store and not indirectly through shared variables, we take as an invariant the property that only closed si-terms, namely si-terms without variables, are put on the store.

In summary of this subsection, we may assume, in the remainder of this article, to be defined a series of sets, a series of mappings, a set of variables and a set of si-terms, say $\mathcal{I}$. Thanks to the mapping definitions, we additionally assume a rewriting relation $\rightsquigarrow$ that rewrites any mapping expression into a set element. For instance, thanks to it, the primitive `tell(down_truck(2))` is subsequently rewritten as `tell(5)`. With this defined, we can proceed with the definition of agents in Multi-Bach.

3.2. Agents

The primitives of Multi-Bach comprise the `tell`, `ask`, `nask` and `get` primitives already mentioned for Bach, which take as arguments elements of $\mathcal{I}$. Primitives can be composed to form more complex agents by using traditional composition operators from concurrency theory: sequential composition, parallel composition and non-deterministic choice. We add two other mechanisms: conditional statements and generalized choice.

Conditional statements take the form $c \rightarrow s_1 \diamond s_2$. Their execution first consists in evaluating condition c and then in computing s_1 if c is evaluated to true or s_2 otherwise. Conditions of type c are obtained from elementary ones, thanks to the classical and, or and negation operators, denoted respectively by &, | and !. Elementary conditions are obtained by relating set elements or mappings on them by equalities (denoted =) or inequalities (denoted ! =, <, <=, >, >=).

Generalized choices are constructs of the form $\Sigma_{e \in S} A_e$ where A_e is an agent parameterized by variable e ranging in set S. Typically, this is obtained by introducing a si-term involving e. The execution is obtained as a generalization of a classical choice. All the instances of A_e obtained by letting e take all the values in S are offered as possible choices. An alternative which can perform a computation step is selected to perform the step of the generalized choice.

Procedures offer a very convenient manner to abstract from series of instructions. They are defined in Multi-Bach similarly to mappings through the `proc` keyword by associating an agent with a procedure name. As in classical

$$(\mathbf{T}) \quad \frac{t \rightsquigarrow u, u \text{ closed}}{\langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle}$$

$$(\mathbf{A}) \quad \langle\, ask(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle$$

$$(\mathbf{G}) \quad \langle\, get(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{N}) \quad \frac{t \rightsquigarrow u, u \notin \sigma, u \text{ closed}}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 5: Transition rules for the primitives

concurrency theory, we assume that the defining agents are guarded, in the sense that any call to a procedure is preceded by the execution of a primitive or can be rewritten in such a form.

As an example, the behavior of a vertical truck can be described as follows:

```
proc VerticalTruck(r: RCInt, c: RCInt) =
  ( (r>1 & r<5) -> ( get(free(pred(r),c));
                     tell(free(succ2(r),c);
                     VerticalTruck(pred(r),c) )
  +
  ( (r>=1 & r<4) -> ( get(free(down_truck(r),c));
                      tell(free(r,c));
                      VerticalTruck(succ(r),c)) ).
```

where `pred`, `succ` and `succ2` are predecessor and successor functions for rows and columns, defined as one may guess:

```
pred(2)  = 1.   pred(3)  = 2.   ...
succ(1)  = 2.   succ(2)  = 3.   ...
succ2(1) = 3.   succ2(2) = 4.  ...
```

In summary, the statements of the Multi-Bach language, also called agents by abuse of language, consist of the statements A generated by the following grammar:

$$A \;::=\; Prim \mid Proc \mid A; A \mid A||A \mid A + A \mid C \rightarrow A \diamond A \mid \Sigma_{e \in S} A_e$$

where $Prim$ represents a primitive, $Proc$ a procedure call, C a condition, e a variable and S a set.

$$(\mathbf{S}) \qquad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle A\ ;\ B \mid \sigma\rangle \longrightarrow \langle A'\ ;\ B \mid \sigma'\rangle}$$

$$(\mathbf{P}) \qquad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \mid\mid B \mid \sigma\rangle \longrightarrow \langle A' \mid\mid B \mid \sigma'\rangle \\ \langle B \mid\mid A \mid \sigma\rangle \longrightarrow \langle B \mid\mid A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{C}) \qquad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A\ +\ B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle B\ +\ A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{Gc}) \qquad \frac{\langle A_f \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle,\ \text{for some } f \in S}{\langle \Sigma_{e\in S} A_e \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

$$(\mathbf{Co}) \qquad \frac{\models C,\ \langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle C \rightarrow A \diamond B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle !C \rightarrow B \diamond A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{Pc}) \qquad \frac{P(\overline{x}) = A,\ \langle A[\overline{x}/\overline{u} \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle P(\overline{u}) \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

Figure 6: Transition rules for the operators

The operational semantics of primitives and complex agents are respectively defined through the transition rules of Figures 5 and 6. Configurations consist of agents (summarizing the current state of the agents) and a multiset of si-terms (denoting the current state of the store). In order to express the termination of the computation of an agent, the set of agents is extended by a special terminating symbol E that can be seen as a completely computed agent. For uniformity purposes, we abuse the language by qualifying E as an agent. To meet the intuition, we always rewrite agents of the form $(E; A)$, $(E \mid\mid A)$ and $(A \mid\mid E)$ as A.

Rules of Figure 5 follow the intuitive description of the primitives. Note that before being processed, the si-term t is rewritten as u by means of the rewriting relation $\rightsquigarrow$. Rules of Figure 6 are quite classical. Rules (S), (P) and (C) provide the usual semantics for sequential, parallel and choice compositions. Rule (Gc) generalizes rule (C) for multiple alternatives obtained

by variable e ranging over set S. As expected, rule (Co) specifies that the conditional instruction $C \rightarrow A \diamond B$ behaves as A if condition C can be evaluated to true and as B otherwise. Note that the notation $\models C$ is used to denote the fact that C evaluates to true. Rule (Pc) makes procedure call $P(\overline{u})$ behave as the agent A defining procedure P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$.

3.3. Processes as active data

An interesting feature of Linda concerns the perception of computation threads as active data. Such a data is put on the shared tuple space as an object to be evaluated. In our setting, procedure definitions provide us with a mechanism to reach such a goal. Indeed, telling a procedure call may be interpreted as creating a new thread of execution, parallel to the thread under consideration. As a procedure call is a string, we may use it to refer to the thread. Moreover, as a procedure call also resembles a si-term, we may consider it as a new form of data. This leads us to extend the blackboard to contain a multiset of procedure calls in addition to the multiset of si-terms.

Consequently, telling several times a same procedure call creates several threads with the same name, in a similar way that the same si-term being told several times induces several occurrences on the store. Following the analogy, asking or getting a procedure call amounts to checking the presence of a thread corresponding to that call, and, in the case of get, of removing one of the threads associated with the call. Moreover, performing a nask primitive on a procedure name amounts to checking the absence of a thread corresponding to the procedure call. Note that terminated processes are considered in a special stopped state but are not removed from the multiset of procedure calls. This allows us to consider windows created by the `New Autonomous Agent` and `new Interactive Agent` buttons as special cases of processes being told, with `Agent1`, `Agent2`, ... as process call identifiers.

More formally, we thus introduce four new primitives: `tellp(Proc)`, `askp(Proc)`, `naskp(Proc)` and `getp(Proc)`. The description of their computations is obtained by means of a new transition relation denoted by the symbol $\hookrightarrow$. It takes as configurations constructions of the form $\prec \mathcal{P} \mid \sigma \succ$ where, on the one hand, $\mathcal{P}$ is a multiset of pairs of the form $pc : A$, with pc a procedure call and A an agent, and, on the other hand, σ a multiset of si-terms. The intuition behind this construct is that $pc : A$ is to be read as procedure pc is in computational state A and that $\mathcal{P}$ gathers together the threads running in parallel, each of which being described by a construct of

the form $pc : A$. The store σ is shared by all the threads. It is directly accessed by them both for reading and writing, so that any change to it is directly seen by all the threads.

The transitions are defined for the primitive agents by the rules (O) to (N_p) of Figure 7. Rule (O) relates the rules (T) to (N) of Figure 5, defining the transitions of the tell, ask, get and nask, with the new transition relation. It states that any computation step of $\langle A \mid \sigma \rangle$ can be lifted to the new configuration $\prec \mathcal{P} \cup \{pc : A\} \mid \sigma \succ$. Rule (T_p) states that $tellp(P)$ generates a new thread, referenced to by P with as current computation state, the agent P to be executed. Moreover, the process pc on which $tellp(P)$ is computed moves to the ending state E. It is here worth noting that union is to be understood in a multiset fashion. Rules (A^o_p) and (A^s_p) proceed in the same way but by checking that the asked process P belongs to the multiset of processes. Rule (A^O_p) treats the case of a process different from the process computing the $askp$ primitive while rule (A^s_p) allows the process to check itself. Rules (G^o_p) and (G^s_p) are similar with the process checked for presence being removed. Finally rule (N_p) specifies that $naskp(P)$ succeeds only when the multiset of procedure calls does not contain a process referenced to by P.

To completely describe the transitions, one should also lift the rules of Figure 6 for the operators to the new configurations. This can be done in a similar way as rules of Figure 5 were lifted in rule (O). For instance, rule (S) can be rewritten as follows:

$$\frac{\prec \mathcal{P} \cup \{pc : A\} \mid \sigma \succ \longhookrightarrow \prec \mathcal{P} \cup \{pc : A'\} \mid \sigma' \succ}{\prec \mathcal{P} \cup \{pc : A; B\} \mid \sigma \succ \longhookrightarrow \prec \mathcal{P} \cup \{pc : A'; B\} \mid \sigma' \succ}$$

Due to the space limit, the other resulting rules are not included in this article.

3.4. Rules

Illustrating computations forces us not only to capture events directly produced by the computations but also to cope with events being caused indirectly by these computations. For instance, a proper modeling of cars moving from a road to another one may lead us to not only illustrate these movements in a canvas depicting the roads but also to update a separate control board providing a global view on the traffic. To that end, following [20, 30] and assuming t, u, v, w to range over data terms and processes, we

$$(\mathbf{O}) \qquad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\prec \mathcal{P} \cup \{pc : A\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \cup \{pc : A'\} \mid \sigma' \succ}$$

$$(\mathbf{T_p}) \qquad \prec \mathcal{P} \cup \{pc : tellp(P)\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \cup \{pc : E\} \cup \{P : P\} \mid \sigma \succ$$

$$(\mathbf{A_p^o}) \quad \prec \mathcal{P} \cup \{pc : askp(P)\} \cup \{P : A\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \cup \{pc : E\} \cup \{P : A\} \mid \sigma \succ$$

$$(\mathbf{A_p^s}) \qquad \prec \mathcal{P} \cup \{pc : askp(pc)\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \cup \{pc : E\} \mid \sigma \succ$$

$$(\mathbf{G_p^o}) \qquad \prec \mathcal{P} \cup \{pc : getp(P)\} \cup \{P : A\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \cup \{pc : E\} \mid \sigma \succ$$

$$(\mathbf{G_p^s}) \qquad \prec \mathcal{P} \cup \{pc : getp(pc)\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \mid \sigma \succ$$

$$(\mathbf{N_p}) \qquad \frac{P : A \notin \mathcal{P} \cup \{pc : naskp(P)\} \text{ for any } A}{\prec \mathcal{P} \cup \{pc : naskp(P)\} \mid \sigma \succ \hookrightarrow \prec \mathcal{P} \cup \{pc : E\} \mid \sigma \succ}$$

Figure 7: Transition rules for active data

introduce rules of the form

$$+t, -u \longrightarrow +v, -w$$

asserting that the presence of t and the absence of u implies the presence of v and the absence of w. More generally such rules may involve several positively and negatively marked objects both in the left and right parts of the rules. Operationnally, they are to be interpreted as the fact that any combination of objects from the left-hand side leads to the addition of positively marked objects in the right-hand side of the rule and the removal of all the occurrences of objects negatively marked in this right-hand side. For instance, with respect to the above rule, a new occurrence of t or a removal of u leads to the addition of v and the removal of (one occurrence of) w.

Rules are applied as long as possible after each change on the blackboard. It is worth noting that, following the analogy between passive si-terms and active processes, objects of rules are to be understood as either si-terms or processes.

The careful reader will have certainly observed that rules may be mimicked by concurrent processes. For instance, the above rule can be mimicked

by the following process:

$$P = ask(t); nask(u); tell(v); get(w); P$$

However, this is only partially true. Indeed, as rules operate in an atomic manner and as long as possible, such processes should receive higher priorities over classical processes and should use an atomic construct, not provided by the language.

Concretely, Multi-Bach handles rules in a twofold manner. First, rules are declared by associating a name to an implication of the form mentioned above. Second, continuing along the analogy between passive and active data, rules are activated by telling them through the `tellr` primitive taking as argument the considered rule name. Rules are then checked for presence or absence by means of the primitives `askr` and `naskr` respectively. The primitive `getr(rn)` is finally used to remove an occurrence of rule named `rn`. This is illustrated by the following rule declaration which asserts that for any si-term `moveCar(p,r,c)` on the store, for some color `p`, some row number `r` and some column number `c`, the procedure call `MoveCar(p,r,c)` should be triggered:

```
rule move_car = for p in Colors, r in Rows, c in Cols:
  +moveCar(p,r,c)  -->  +MoveCar(p,r,c), -moveCar(p,r,c).
```

Note that to avoid that the rule is applied several times, the `moveCar(r,c,p)` si-term is also removed. Besides its declaration, this rule is activated by executing `tellr(move_car)`, for instance in the beginning of the whole computation.

To formally grasp rules, one has to extend once more blackboards to include multisets of rule names. We are thus lead to new configurations of the form $\lhd \mathcal{P} \mid \sigma \mid R \rhd$ with $\mathcal{P}$ a multiset of procedure calls associated with agents, σ a multiset of si-terms and R a multiset of rule names. It is easy to extend rules of Figure 7 to these new configurations. Moreover the execution of the primitives *tellr*, *askr*, *getr* and *naskr* may be described as one may expect by the corresponding updates of the R component. Because of space limit, these rules are not presented in this article. Let us however denote by $\twoheadrightarrow$ the resulting transition relation. In contrast, the execution of rules requires some attention. It is denoted by an auxiliary relation $\mapsto$. A few definitions are first needed.

Definition 1. *Define the set SRP of rule primitives as the set of elements*

generated by the following grammar:

$$RP ::= +s| - s| + P| - P$$

where s denotes a si-term, possibly containing variables, and P denotes a procedure call, possibly containing variables. Define the set SeqRP as the set of non empty sequences of rule primitives. Given such a sequence Seq, we respectively denote by Pterm(Seq) and Nterm(Seq) the multiset of positively marked si-terms $+s$ of Seq and the multiset of negatively marked si-terms $-s$ of Seq. Similarly, Pproc(Seq) and Nproc(Seq) are respectively the multiset of positively marked procedure calls $+P$ of Seq and the multiset of negatively marked procedure calls $-P$ of Seq.

Define the set Srule as the set of constructions of the form

$$for\ v_1 \in S_1, \cdots, v_n \in S_n : A \rightarrow B$$

where $A, B \in SeqRP$, where S_1, ..., S_n are user-defined sets, and where v_1, ..., v_n are all the variables occurring in A and B. The sequence A is called the precondition of the rule and the sequence B is called its postcondition.

Define the set Snrule of named rules as the set of constructs of the form $n = r$, where n is an identifier and r is a rule of Srule. We assume in the following that the identifier n of a named rule univocally identifies the rule. Consequently, given an identifier n, we denote by Inst(n) the set of all the instances $Ai \rightarrow Bi$ obtained by replacing the variables v_1, ..., v_n in the above rule by all the values of S_1, ..., S_n.

Several auxiliary notations are needed to formulate the transition rules.

Definition 2. *Given a multiset $\mathcal{P}$ of pairs pc:A of procedure calls and agents, we denote by $PC(\mathcal{P})$ the multiset of procedure calls pc appearing in $\mathcal{P}$. Moreover, for a multiset MPC of procedure calls, we denote by $\mathcal{P} \diagdown MPC$ the multiset obtained from $\mathcal{P}$ by removing a pair pc:A for any pc occurring in MPC. Dually, we denote by $\mathcal{P} \sqcup MPC$ the multiset obtained from $\mathcal{P}$ by adding a pair of the form pc:pc for each occurrence of pc in MPC.*

Definition 3. *Given two multisets σ and τ of closed si-terms, we denote by $\sigma \ominus \tau$ the multiset obtained from σ by removing an occurrence for each si-term of τ.*

$$(\mathbf{R}) \quad \frac{\begin{array}{c} n \in R, (A \to B) \in Inst(n), Pterm(A) \subseteq \sigma, Pproc(A) \subseteq PC(\mathcal{P}), \\ Nterm(A) \cap \sigma = \emptyset, Nproc(A) \cap PC(\mathcal{P}) = \emptyset \end{array}}{\begin{array}{rl} \lhd \mathcal{P} \mid \sigma \mid R \rhd & \longmapsto \quad \lhd (\mathcal{P} \sqcup Pproc(B)) \diagdown Nproc(B) \mid \\ & \qquad (\sigma \cup Pterm(B)) \ominus Nterm(B) \mid R \rhd \end{array}}$$

Figure 8: Transition for the application of a rule

We are now in a position to explain the transition induced by the execution of a rule. It is formalized by rule (R) of Figure 8. Given a rule of R, identified by n, we consider an instance of that rule, say $(A \to B)$, which can be applied. This requires that positively marked si-terms and processes of the precondition A are present in the current configuration and that negatively marked si-terms and processes are absent. More formally, as regards si-terms, this amounts to check that $Pterm(A) \subseteq \sigma$ for the positively marked si-terms and that $Nterm(A) \cap \sigma = \emptyset$ for the negatively marked si-terms. Procedure calls are handled similarly by using $PC(\mathcal{P})$ instead of σ.

Now the application of such an applicable instance $A \to B$ updates the current store σ by adding the positively marked si-terms of the postcondition B and by removing an occurrence of any negatively marked si-terms. Similarly, the current multiset of procedure calls is updated by adding the positively marked procedure calls of B and removing an occurrence of any negatively marked procedure calls of B. This is formalized in rule (R) by adding $Pterm(B)$ to σ and by taking the $\ominus$-difference with $Nterm(B)$. Similarly, $\mathcal{P}$ is updated by adding new procedure calls for any element of $Pproc(B)$ and removing the procedure calls of $Nproc(B)$.

To conclude the formalization of rules, we need to relate the $\twoheadrightarrow$ and $\mapsto$ transition relations. This is achieved by rule (W_1) and (W_2) of Figure 9 and by introducing a new transition relation $\rightarrowtail$, which is the final one. Rule (W_1) expresses that a $\rightarrowtail$ transition step can occur whenever a $\mapsto$ transition step can occur. In contrast, rule (W_2) states that a $\twoheadrightarrow$ transition step can only occur when a $\mapsto$ transition step cannot occur. This expresses the fact that rules are applied as long as possible before any other transition can take place.

$$(\mathbf{W_1}) \quad \frac{\lhd \mathcal{P} \mid \sigma \mid R \rhd \;\mapsto\; \lhd \mathcal{P}' \mid \sigma' \mid R' \rhd}{\lhd \mathcal{P} \mid \sigma \mid R \rhd \;\rightarrowtail\; \lhd \mathcal{P}' \mid \sigma' \mid R' \rhd}$$

$$(\mathbf{W_2}) \quad \frac{\begin{array}{c}\lhd \mathcal{P} \mid \sigma \mid R \rhd \;\twoheadrightarrow\; \lhd \mathcal{P}' \mid \sigma' \mid R' \rhd, \\ \lhd \mathcal{P} \mid \sigma \mid R \rhd \;\not\mapsto\; \lhd \mathcal{P}'' \mid \sigma'' \mid R'' \rhd\end{array}}{\lhd \mathcal{P} \mid \sigma \mid R \rhd \;\rightarrowtail\; \lhd \mathcal{P}' \mid \sigma' \mid R' \rhd}$$

Figure 9: Rules for the final transition relation

3.5. Animations

Animations are obtained in a twofold manner: on the one hand, by describing the scene to be painted and, on the other hand, by primitives to place widgets, to move them, to make them appear or disappear, and more generally to change their attributes.

Several scenes may be defined in Anemone. The description of a scene consists of the specification of the size of the canvas to be used by the animation, of the background image of the animation, of a series of images to be used, and of a series of widgets. A widget in turn involves the definition of attributes, the way they are displayed and various initial values for them. Besides user-defined attributes, six special attributes are predefined: the current coordinates `wdX` and `wdY` of the widget, the rotation angle `wdA` of the widget, the scale factor `wdF` of the widget, the visibility `wdV` of the widget, and the layer `wdL` of the widget.

The following listing provides an example of the definition of the scene named `rhScene`. It is declared to be of size 640 by 640 pixels and to contain three layers, named `top`, `middle`, and `bottom`. Three images are to be used: the first one defines the background and others images to be used for a green and a red car. Finally a widget `car` is defined to represent a particular car of interest. It has a user-defined attribute `color`, whose possible values are those of the set `idColors`, assumed to be defined as {`green, red`} in a set declaration. The widget is displayed according to conditions, here as a green car if `color` is `green` and as a red car if `color` is `red`. Finally, some initial values are defined for predefined attributes: the initial positions are at (40,60), the layer is `top` and the color is `red`. It is worth noting that, by default, predefined attributes receives the following values: `wdX` = `wdY` = `wdA` = `wdV` = 0, `wdF` = 1 and `wdL` is the first element listed in `layers`.

```
scene rhScene = {
```

```
  size = (640,640).
  layers = { top, middle, bottom }.

  background = loadImage(Images/the_background_img.png).
  green_car = loadImage(Images/green_car.jpg).
  red_car = loadImage(Images/red_car.jpg).

  widget car = {
    attributes = {
      color in idColors.
    }
    display = {
      color = green -> green_car.
      color = red -> red_car.
    }
    init = {
      wdX = 40.
      wdY = 60.
      wdL = top.
      color = red.
    }
  }

}
```

Note that file names are given with respect to the path in which Anemone is executed. The canvas size and coordinates are expressed in pixels, with $(0, 0)$ being the upper-left corner of the canvas. Moreover, conditions guiding the display of widgets take the form of conditional statements defined in Subsection 3.2.

Scenes and widgets are manipulated by means of two primitives:

- `draw_scene(s)`: to draw the scene identified by s
- `att(x,w,s,v)`: to give value v to the attribute x of the widget identified by w on scene s.

As some predefined attributes are used so frequently, a few primitives have been added to cope more declaratively with these attributes, although they

are actually syntatic sugar for an initialization or the use of the `att` primitive. They consist of:

- `place_at(w,s,x,y)`: to place the widget identified by w on scene s at the coordinates (x, y)
- `move_to(w,s,x,y)`: to move the widget identified by w on scene s from its current position to the new coordinates (x, y)
- `hide(w,s)`: to hide the widget identified by w on scene s
- `show(w,s)`: to make appear the widget identified by w on scene s
- `layer(w,s,l)`: to place the widget identified by w on scene s on the layer l of that scene.

All these primitives are added to the various versions of the `tell`, `ask`, `get` and `nask` primitives of the previous subsections. Their execution does not modify the configurations used in the transitions. Consequently no modification of the previous transition relations is needed. A formal treatment of these new primitives can be achieved as a sequence of widget updates. It would however not add much to the description of Anemone and is thus left out of the scope of this paper.

3.6. A fragment of temporal logic

Linear temporal logic is a logic widely used to reason over dynamic systems. The Anemone workbench uses a fragment of PLTL [31] with, as main goal, to check the reachability of states.

As usual, the logic used by Anemone relies on propositional state formulae. In our coordination context, these formulae are to be verified on the current contents of the store. Consequently, given a structured piece of information t, we introduce $\#t$ to denote the number of occurrences of t on the store and define as basic propositional formulae, equalities or inequalities combining algebraic expressions involving integers and number of occurrences of structured pieces of information. An example of such a basic formula is $\#free(1, 1) = 1$ which states that the cell of coordinates $(1, 1)$ is free.

Propositional state formulae are built from these basic formulae by using the classical propositional connectors. As particular cases, we use *true* and *false* to denote propositional formulae that are respectively always true and

false. Such formulae are in fact shorthands to denote respectively $p \vee \neg p$ and $p \wedge \neg p$, for some basic propositional formula p.

The fragment of temporal logic used in Anemone is then defined by the following grammar:

$$TF ::= PF \mid Next\ TF \mid PF\ Until\ TF$$

where PF is a propositional formula. As an example, if the red car indicates that it leaves the grid by placing *out* on the store, a solution to the rush problem is obtained by verifying the formula

$$true\ Until\ (\#out = 1)$$

Such formulae being very often used, they are abbreviated in Anemone as

$$Reach\ (\#out = 1)$$

4. Anemone revisited

As regards user interactions, many of the features of Anemone are treated by the workbench as sketched in Section 2. Two features require additional treatments: processes perceived as active data and rules.

Interactive and autonomous agent windows can actually be seen by the user as the incarnation of processes, as their role is to compute agents, these including procedure calls. Consequently, for the Anemone workbench, computing the `tellp(P)` primitive amounts to creating a window similar to the interactive or autonomous agent windows. Accordingly, computing the `getp(P)` primitive not only consists of removing an occurrence of process P in the multiset of procedure calls $\mathcal{P}$ but also consists of removing the window associated with P. Note that we have taken the hypothesis that the interactive or autonomous property is inherited from the agent window on which the `tellp` primitive is executed.

Rules require a new kind of window. To allow the user to better grasp the current state of computation, the Anemone workbench displays the current instances of the rules that can be activated in the current context of application. It is then up to the user to select the rule to be applied.

5. Implementation

The Anemone workbench has been implemented in Scala [32] on top of the Processing library [33]. Scala is a programming language which combines the object-oriented and functional paradigms and which benefits from strong static type systems. Scala source code is compiled to Java bytecode, which eases its interface with Java libraries. Moreover, Scala includes powerful parsing facilities. All these properties make it well-suited to interpret the Multi-Bach language, which as can be appreciated from the previous sections, can be easily described by recursive definitions.

Processing is a graphical library built to teach programming to artists in a visual context. Although it is generally used through a specific IDE, Processing can be employed as a Java library, which is the case for Anemone. Processing is based on a key method, named `draw`, that is invoked several times per second (typically 60 times per second), which accordingly creates animations by modifying parameters such as the coordinates of images.

The page limit does not allow to provide details on the implementation. However, the rest of this section should enable to capture the key ideas. The interested reader is referred to `https://github.com/UNamurCSFaculty/anemone` where the whole code is available.

Internal representation of data. The store, the processes being told as active data, set, map and procedure definitions are memorized by using Scala maps and lists. Moreover Scala case classes are used to represent abstract trees of Multi-Bach instructions with a one-to-one relationship to these instructions. That property allows to keep a close link between statements expressed in Multi-Bach and their internal representation in Anemone. As a result, in contrast to tools such as mCRL2, it is quite easy to provide the user with interfaces and messages directly connected to what he has written.

Computations. Multi-Bach computations are simulated by repeatedly executing transition steps. Non-determinism induced by the choice and parallel composition operators is simulated by random variables.

Temporal logic. Anemone temporal formulae are verified by means of a home-made model-checker inspired by the techniques proposed in [34]. It essentially uses a limited depth-first search algorithm based on the simulator of computations with a recursive reasoning over the temporal formulae. Although convenient to produce first experiments, this model-checker suffers from the

state-space explosion as well as from its incompleteness. Future research will aim at improving these aspects.

6. Related work

Although many pieces of work in the coordination community have been devoted to the proposal of new languages, semantics and implementations, few articles have addressed the concerns of practically constructing programs in coordination languages, in particular in checking that what is described by programs actually corresponds to what has to be modeled. Notable exceptions include the Extensible Coordination Tools [11], ReoLive [12], and TAPAs [13].

The Extensible Coordination Tools (ECT) has been developed for the control-based coordination language Reo, a language quite different from Multi-Bach. The ECT tools consist of a set of plug-ins for the Eclipse platform that provide graphical editing facilities of Reo connectors, the animation of these connectors as well as model checking based on constraint automata or a translation to the process algebra mCRL2 [35]. Related to ECT is a tool proposed in [36] that creates animations for Reo connectors by using on-the-fly generated Flash code. Although it is certainly less elaborated, our work differs in several respects. First, it deals with tuple spaces instead of connectors. Second, it allows to grasp the modeling of real-life systems by connecting agents of Multi-Bach to animations at the application level. Consequently, although one may animate connectors in ECT, one cannot animate the modeling of the rush hour problem for instance, as we do with Multi-Bach. Finally, in contrast to our work, model checking in ECT does not preserve a one-to-one link with textual representations, in particular when mCRL2 is used.

ReoLive is also dedicated to Reo. It proposes similar tools but by means of a set of web-based tools using ScalaJS. As a consequence, the above comparison with ECT also applies to ReoLive.

TAPAs [13] is a tool developed essentially for CCSP with a plug-in for an extension of the Klaim coordination language. It allows to graphically specify systems and to verify their equivalence by means of bisimulations based equivalences (strong, weak and branching) or decorated trace equivalences (weak and strong variants of trace completed trace, divergence sensitive trace, must, testing). It also allows to model check systems by using formulae of the μ-calculus. The two main differences of our work with TAPAs are, on the

one hand, our concern for tuple-based coordination languages, and, on the other hand, the facilities offered by Multi-Bach for animations. In contrast, as written above, model checking in Multi-Bach is quite simple and is much less elaborated than that of TAPAs. Future work will aim at improving this aspect.

Klaim [8] is a Linda-like coordination language specifically designed to model and program distributed systems. Recently, its programming language declension X-Klaim has been renewed to benefit from modern IDE facilities [14]. Thanks to Xtext/Xbase it allows the user to benefit from Eclipse facilities, such as content assist, code navigation and debugging. Being focused on analyzing the programs from a semantic point of view, Anemone does not provide similar facilities. This is however an obvious improvement to be done, which will be the subject of future work. Nevertheless, Anemone offers animations which are not provided by X-Klaim.

Declarative invariant assertions are proposed in [37] to detect inconsistencies in models expressed in the Peer model, a coordination model based on shared tuple spaces, messages and Petri nets. In addition to the fact that the Peer Model is quite different from Multi-Bach, our work differs in two main respects. On the one hand, assertions in [37] are verified at runtime whereas our temporal formulae are checked statically. On the other hand, in contrast to our work, no animation facilities are provided.

Although it includes facilities to view the evolution of the shared space, TUCSON [38] does not provide facilities to animate computations nor to model-check them.

A Linda workbench is presented in [39] with the goal of providing a simple tool that allows users to experiment with a Linda-inspired language. It is integrated with Netbeans and uses the JavaSpaces language, an extension of Java supporting Linda primitives. It is consequently named JavaSpaces Netbeans. This workbench provides a tuple browser and a distributed debugger, including record facilities to replay a sequence of tuple space operations. Although our work provides facilities to explore and modify the tuple space, we do not provide debugging facilities. In contrast however we provide animation facilities as well as model checking facilities which are not included in JavaSpaces Netbeans.

The article [40] shares our concern of modeling systems by using a suitable process algebra. In this aim, the authors introduce a language based on the concept of virtual stigmergy, a distributed data structure able to model global knowledge. By using in addition attribute-based communication, they show

how several applications can be coded, such as flocking behavior, foraging, opinion formation, and population protocols. In contrast to our work, no programming environment and no animation facilities are however provided.

NetLogo [41], Jason [42], JaCaMo [43] and Repast [44] are other research efforts pursuing the same goal of modeling and simulating multi-agent systems. They are coupled to programming environments using several languages. NetLogo is based on a Logo dialect and relies on the speech-act based communication. JaCaMo is a combination of Jason, for programming autonomous agents, Cartago, for programming environment artifacts, and Moise, for programming multi-agent organizations. Repast uses statecharts to define in a visual way state transitions. Except for the support of messages being exchanged between agents, there is no real concern for coordination, as we do. Moreover, the accent is put on modeling systems and simulating them whereas our main goal is, through animations, build confidence that the model corresponds to what needs to be modeled.

It is worth noting that the idea of providing animations to help the user understand the models of real-life applications has been explored in many domains. For instance, the articles [45, 46, 47, 48] animate Event-B specifications by associating images with states and by re-drawing the images according to state transitions. In the aim of animating goal-oriented requirements, the article [49] presents the tool Faust that automatically generates parallel state machines from goal operationalizations, instantiates those machines to specific instances created by users at animation time and executes them from concurrent events input by multiple users. In [50], an animation tool, relying on Coloured Petri Nets, is presented to create visualizations of formal models. In the aim of animating reactive systems, the article [51] introduces the framework of reactive animation, which in two words, uses the Rhapsody tool and flash animations to provide users with dynamic interfaces to represent the considered reactive systems and their operational behavior. More generally, the CoSim-CPS workshops [52, 53] have provided for the last years a venue to gather researchers interested in combining formal methods with advanced simulation techniques. As a last reference, the article [54] shows how patterns can be used to verify the design of user interfaces with respect to specifications. Our work contrasts with these pieces of work by being focused on a coordination language but shares the fundamental idea of illustrating models.

Finally, this work is an extension of the Scan workbench described in [26]. Several features have been revised and new ones have been included. At the

process algebra level, we have introduced variables, generalized choices, primitives to handle processes as active data, primitives to manipulate rules and have revised the description of scenes, allowing here for multiple scenes and using widgets as main elements of scenes. We have also refined the workbench to account for these new features as well as to provide new facilities. The interactive blackboard has been reworked to incorporate a filter button enabling the user to select elements of the shared space of interest. New windows are provided for the creation of new processes induced by their view as passive data. Moreover a new kind of window is proposed to handle rules.

7. Conclusion

The article has introduced a workbench for reasoning over a Linda-like coordination language at three levels: (i) by executing instructions in a step by step or automatic manner instructions while showing their impact on the shared space, (ii) by illustrating computations by animations and (iii) by model checking properties by means of temporal formulae.

The current version has been designed to be as simple as possible yet incorporating key ideas. As a result, it can be improved in many aspects, in particular, by refining the interfaces, by integrating it in IDE's, by improving the specification of animations and by handling more sophisticated temporal logics, like the $\mu-$calculus [55].

References

[1] D. Gelernter, N. Carriero, Coordination Languages and Their Significance, Communications of the ACM 35 (2) (1992) 97–107.

[2] N. Carriero, D. Gelernter, Linda in Context, Communications of the ACM 32 (4) (1989) 444–458.

[3] M. Banville, SONIA: An Adaptation of LINDA for Coordination of Activities in Organisations, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 57–74.

[4] K. D. Bosschere, J.-M. Jacquet, μ^2 Log: Towards Remote Coordination, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the International

Conference on Coordination Languages and Models, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 142–159.

[5] M. Cremonini, A. Omicini, F. Zambonelli, Coordination in Context: Authentication, Authorisation and Topology in Mobile Agent Applications, in: P. Ciancarini, A. Wolf (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1594 of Lecture Notes in Computer Science, Springer, 1999, p. 416.

[6] C.-L. Fok, G.-C. Roman, G. Hackmann, A Lightweight Coordination Middleware for Mobile Computing, in: R. De Nicola, G. Ferrari, G. Meredith (Eds.), Proceedings of the International Conference on Coordination Models and Languages, Vol. 2949, Springer, 2004, pp. 135–151.

[7] R. Tolksdorf, Coordinating Services in Open Distributed Systems with LAURA, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 386–402.

[8] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, B. Venneri, The Klaim Project: Theory and Practice, in: C. Priami (Ed.), Proceedings of the International Workshop on Global Computing, Programming Environments, Languages, Security, and Analysis of Systems, Vol. 2874 of Lecture Notes in Computer Science, Springer, 2003, pp. 88–150.

[9] A. Murphy, G. Picco, G.-C. Roman, LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents, ACM Trans. Softw. Eng. Methodol. 15 (3) (2006) 279–328.

[10] R. De Nicola, T. Duong, O. Inverso, C. Trubiani, AErlang: Empowering Erlang with Attribute-based Communication, Science of Computer Programming 168 (2018) 71–93.

[11] N. Kokash, F. Arbab, Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools, IEEE Transactions on Services Computing 6 (2) (2013) 186–200.

[12] R. Cruz, J. Proença, ReoLive: Analysing Connectors in Your Browser, in: M. Mazzara, I. Ober, G. Salaün (Eds.), Proceedings of the STAF

collocated workshops, Vol. 11176 of Lecture Notes in Computer Science, Springer, 2018, pp. 336–350.

[13] F. Calzolai, R. De Nicola, M. Loreti, F. Tiezzi, TAPAs: A Tool for the Analysis of Process Algebras, in: K. Jensen, W. van der Aalst, J. Billington (Eds.), Transactions on Petri Nets and Other Models of Concurrency, Vol. 5100 of Lecture Notes in Computer Science, Springer, 2008, pp. 54–70.

[14] L. Bettini, E. Merelli, F. Tiezzi, X-Klaim Is Back, in: M. Boreale, F. Corradini, M. Loreti, R. Pugliese (Eds.), Models, Languages, and Tools for Concurrent and Distributed Programming: Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday, Vol. 11665 of Lecture Notes in Computer Science, Springer, 2019, pp. 115–135.

[15] A. Brogi, J.-M. Jacquet, On the Expressiveness of Linda-like Concurrent Languages, Electronical Notes in Theoretical Computer Science 16 (2) (1998) 61–82.

[16] A. Brogi, J.-M. Jacquet, On the Expressiveness of Coordination via Shared Dataspaces, Science of Computer Programming 46 (1-2) (2003) 71–98.

[17] D. Darquennes, J.-M. Jacquet, I. Linden, On Density in Coordination Languages, in: C. Canal, M. Villari (Eds.), CCIS 393, Advances in Service-Oriented and Cloud Computing, ESOCC 2013, Proceedings of Foclasa Workshop, Springer, Malaga, Spain, 2013, pp. 189–203.

[18] D. Darquennes, J.-M. Jacquet, I. Linden, On the Introduction of Density in Tuple-Space Coordination Languages, Science of Computer Programming.

[19] D. Darquennes, J.-M. Jacquet, I. Linden, On Distributed Density in Tuple-based Coordination Languages, in: J. Cámara, J. Proença (Eds.), Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, Vol. 175 of EPTCS, Springer, Rome, Italy, 2015, pp. 36–53.

[20] D. Darquennes, J.-M. Jacquet, I. Linden, On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its

Expressiveness Study, in: G. D. M. Serugendo, M. Loreti (Eds.), Proceedings of the 20th International Conference on Coordination Models and Languages, Vol. 10852 of Lecture Notes in Computer Science, Springer, 2018, pp. 81–109.

[21] J.-M. Jacquet, K. D. Bosschere, A. Brogi, On Timed Coordination Languages, in: A. Porto, G.-C. Roman (Eds.), Proc. 4th International Conference on Coordination Languages and Models, Vol. 1906 of Lecture Notes in Computer Science, Springer, 2000, pp. 81–98.

[22] J.-M. Jacquet, I. Linden, Fully Abstract Models and Refinements as Tools to Compare Agents in Timed Coordination Languages, Theoretical Computer Science 410 (2-3) (2009) 221–253.

[23] I. Linden, J.-M. Jacquet, On the Expressiveness of Absolute-Time Coordination Languages, in: R. D. Nicola, G. Ferrari, G. Meredith (Eds.), Proc. 6th International Conference on Coordination Models and Languages, Vol. 2949 of Lecture Notes in Computer Science, Springer, 2004, pp. 232–247.

[24] I. Linden, J.-M. Jacquet, On the Expressiveness of Timed Coordination via Shared Dataspaces, Electronical Notes in Theoretical Computer Science 180 (2) (2007) 71–89.

[25] I. Linden, J.-M. Jacquet, K. D. Bosschere, A. Brogi, On the Expressiveness of Relative-Timed Coordination Models, Electronical Notes in Theoretical Computer Science 97 (2004) 125–153.

[26] J.-M. Jacquet, M. Barkallah, Scan: A Simple Coordination Workbench, in: H. Riis Nielson, E. Tuosto (Eds.), Proceedings of the 21st International Conference on Coordination Models and Languages, Vol. 11533 of Lecture Notes in Computer Science, Springer, 2019, pp. 75–91.

[27] G. Papadopoulos, F. Arbab, Coordination Models and Languages, in: Technical Report SEN-R9834, Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 1998.

[28] L. Erman, F. Hayes-Roth, V. Lesser, D. Reddy, The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty, ACM Computing Surveys 12 (2) (1980) 213.

[29] J.-M. Jacquet, I. Linden, Coordinating Context-aware Applications in Mobile Ad-hoc Networks, in: T. Braun, D. Konstantas, S. Mascolo, M. Wulff (Eds.), Proceedings of the first ERCIM workshop on eMobility, The University of Bern, 2007, pp. 107–118.

[30] J.-M. Jacquet, I. Linden, M. Staicu, Blackboard Rules: from a Declarative Reading to its Application for Coordinating Context-aware Applications in Mobile Ad Hoc Networks, Science of Computer Programming 115-116 (2016) 79–99.

[31] E. A. Emerson, Temporal and Modal Logic, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), Elsevier, 1990, pp. 995–1072.

[32] M. Odersky, L. Spoon, B. Venners, Programming in Scala, A comprehensive step-by-step guide, Artemis, 2016.

[33] C. Reas, B. Fry, Processing: A Programming Handbook for Visual Designers, The MIT Press, 2014.

[34] M. Reynolds, A Traditional Tree-style Tableau for LTL, CoRR abs/1604.03962.

[35] S. Cranen, J. Groote, J. Keiren, F. Stappers, E. de Vink, W. Wesselink, T. Willemse, An Overview of the mCRL2 Toolset and Its Recent Advances, in: N. Piterman, S. Smolka (Eds.), Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 7795 of Lecture Notes in Computer Science, Springer, 2013, pp. 199–213.

[36] J. Proenca, Synchronous Coordination of Distributed Components, Ph.D. thesis, Leiden University, Leiden, Netherlands (2011).

[37] E. Kühn, S. Radschek, N. Elaraby, Distributed Coordination Runtime Assertions for the Peer Model, in: G. Di Marzo Serugendo, M. Loreti (Eds.), Proceedings of the 20th International Conference on Coordination Models and Languages, Vol. 10852 of Lecture Notes in Computer Science, Springer, 2018, pp. 200–219.

[38] A. Omicini, A. Ricci, G. Rimassa, M. Viroli, Integrating Objective & Subjective Coordination in FIPA: A Roadmap to TuCSoN, in: G. Armano, F. D. Paoli, A. Omicini, E. Vargiu (Eds.), Proceedings of the 4th AI*IA/TABOO Joint Workshop "From Objects to Agents": Intelligent Systems and Pervasive Computing, Pitagora Editrice Bologna, 2003, pp. 85–91.

[39] M. Dukielska, J. Sroka, JavaSpaces NetBeans: a Linda Workbench for Distributed Programming Course, in: R. Ayfer, J. Impagliazzo, C. Laxer (Eds.), Proceedings of the 15th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ACM, 2010, pp. 23–27.

[40] R. De Nicola, L. Di Stefano, O. Inverso, Multi-agent Systems with Virtual Stigmergy, in: M. Mazzara, I. Ober, G. Salaün (Eds.), Proceedings of the Collocated Workshops of STAF 2018, Vol. 11176 of Lecture Notes in Computer Science, Springer, 2018, pp. 351–366.

[41] The NetLogo Project, https://ccl.northwestern.edu/netlogo, accessed: 2019-11-26.

[42] The Jason Project, http://jason.sourceforge.net/wp, accessed: 2019-11-26.

[43] The JaCaMo Project, http://jacamo.sourceforge.net, accessed: 2019-11-26.

[44] The Repast Project, https://repast.github.io, accessed: 2019-11-26.

[45] L. Ladenberger, J. Bendisposto, M. Leuschel, Visualising Event-B Models with B-Motion Studio, in: M. Alpuente, B. Cook, C. Joubert (Eds.), Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems, Vol. 5825 of Lecture Notes in Computer Science, Springer, 2009, pp. 202–204.

[46] L. Ladenberger, M. Leuschel, BMotionWeb: A Tool for Rapid Creation of Formal Prototypes, in: R. De Nicola, E. Kühn (Eds.), Proceedings of the International Conference on Software Engineering and Formal Methods, Vol. 9763 of Lecture Notes in Computer Science, Springer, 2016, pp. 403–417.

[47] D. Méry, N. Singh, Real-Time Animation for Formal Specification, in: M. Aiguier, F. Bretaudeau, D. Krob (Eds.), Proceedings of the International Conference on Complex System Design and Management, Springer, 2010, pp. 49–60.

[48] T. Servat, BRAMA: A New Graphic Animation Tool for B Models, in: J. Julliand, O. Kouchnarenko (Eds.), Proceedings of the International Conference on Formal Specification and Development in B, Vol. 4355 of Lecture Notes in Computer Science, Springer, 2007, pp. 274–276.

[49] H. Van, A. van Lamsweerde, P. Massonet, C. Ponsard, Goal-Oriented Requirements Animation, in: Proceedings of the IEEE International Conference on Requirements Engineering, IEEE Computer Society, 2004, pp. 218–228.

[50] M. Westergaard, K. Lassen, The BRITNeY Suite Animation Tool, in: S. Donatelli, P. Thiagarajan (Eds.), Proceedings of the International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Vol. 4024 of Lecture Notes in Computer Science, Springer, 2006, pp. 431–440.

[51] D. Harel, S. Efroni, I. Cohen, Reactive Animation, in: F. de Boer, M. Bonsangue, S. Graf, W. de Roever (Eds.), Proceedings of the International Symposium on Formal Methods for Components and Objects, Vol. 2852 of Lecture Notes in Computer Science, Springer, 2002, pp. 136–153.

[52] A. Cerone, M. Roveri (Eds.), Proceedings of the Collocated Workshops of the International Conference on Software Engineering and Formal Methods (DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA), Vol. 10729 of Lecture Notes in Computer Science, Springer, 2018.

[53] M. Mazzara, I. Ober, G. Salaün (Eds.), Proceedings of Collocated Workshops of the International Conference on Software Technologies: Applications and Foundations, Vol. 11176 of Lecture Notes in Computer Science, Springer, 2018.

[54] M. Harrison, P. Masci, J. Campos, Verification Templates for the Analysis of User Interface Software Design, IEEE Transactions on Software Engineering 45 (8) (2019) 802–822.

[55] D. Kozen, Results on the Propositional mu-Calculus, Theoretical Computer Science 27 (1983) 333–354.

Required Metadata

Current executable software version

Ancillary data table required for sub version of the executable software:

Nr.	(executable) Software metadata description	Please fill in this column
S1	Current software version	1.0
S2	Permanent link to executables of this version	*https://github.com/UNamurCSFaculty/anemone/anemone.jar*
S3	Legal Software License	GPL-3.0
S4	Computing platform/Operating System	Linux, OS X, Microsoft Windows
S5	Installation requirements	Java 1.8
S6	If available, link to user manual	*https://staff.info.unamur.be/jmj/Anemone/documentation*
S7	Support email for questions	jean-marie.jacquet@unamur.be and manel.barkallah@unamur.be

Table 1: Executable Software Metadata

Current code version

Ancillary data table required for subversion of the codebase.

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.0
C2	Permanent link to code/repository used of this code version	$https://github.com/UNamurCSFaculty/anemone/SourceCode/Anemone-1.0$
C3	Legal Code License	GPL-3.0
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Scala, Java, Processing
C6	Compilation requirements, operating environments & dependencies	Scala 2.12
C7	If available Link to developer documentation/manual	$https://staff.info.unamur.be/jmj/Anemone/documentation$
C8	Support email for questions	jean-marie.jacquet@unamur.be and manel.barkallah@unamur.be

Table 2: Code metadata