Scan: A Simple Coordination Workbench

Jean-Marie Jacquet, Manel Barkallah

Scan: a Simple Coordination Workbench

Jean-Marie Jacquet[1][0000−0001−9531−0519] and Manel Barkallah[1][0000−0003−2608−5658]

Nadi Research Institute, Faculty of Computer Science, University of Namur
Rue Grandgagnage 21, 5000 Namur, Belgium
{jean-marie.jacquet,manel.barkallah}@unamur.be

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 on programs written in these languages. This paper proposes a simple workbench for describing concurrent systems using a Linda-like language, for animating them and for reasoning on them using a fragment of linear temporal logic. In contrast to some tools developed for traditional process algebras like CCS, a key feature of our workbench is that it maintains a direct relation between what is written by the user and its internal representation in the workbench. Another feature, particularly useful for didactic purposes, is the production of trace examples, replayable, when LTL formulae are satisfied.

Keywords: Coordination · Bach · animation · verification.

1 Introduction

In the aim of building interactive distributed systems, a clear separation between the interactional and the computational aspects of software components has been advocated by Gelernter and Carriero in [14]. Their claim has been supported by the design of a model, Linda [4], 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. The authors have themselves contributed to that trend of research, as exemplified for instance in [1, 2, 7–10, 15, 17, 21–23]. However, although many pieces of work (including ours) 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.

Based on previous work by the first author on a Linda-like dialect, named Bach, this paper aims at introducing a workbench to reason on programs written in Bach extended with several facilities. More specifically, our goal is threefold:

Fig. 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 allow the user to understand the meaning of instructions written in Bach, 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 in Bach, by connecting agents in Bach to animations, representing the evolution of the modeled system;
- to allow the user to check properties by model checking temporal logic 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 shall build it as a standalone executable file launched by a simple command line. We shall also propose a simple process algebra that allows the user to concentrate on the key coordination and animation features and consequently avoid him the burden of handling extra features typically required by sophisticated commercial systems;
- 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.

To make our developments more concrete, we shall use the rush hour puzzle as a running example. 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.

The rest of this paper is organized as follows. Section 2 describes the functionalities of Scan and, in doing so, provides an overview of the tool. Section 3

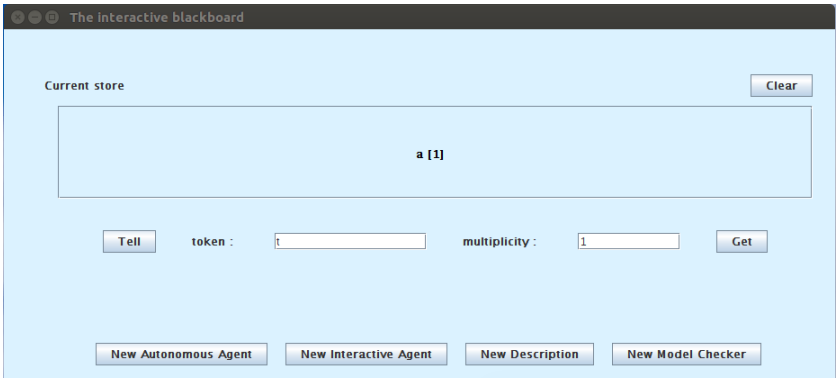


Fig. 2. The interactive blackboard window

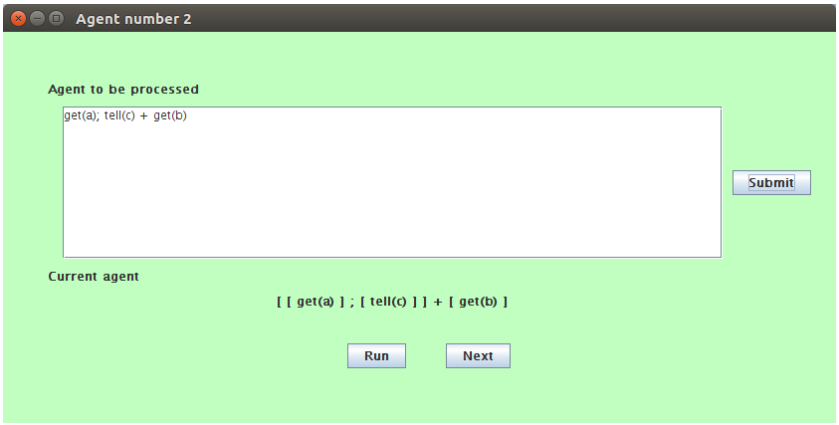
specifies the coordination language and temporal logic to be used in the tool. Section 4 sketches how Scan is implemented. Section 5 compares our work with related work. Finally, Section 6 draws our conclusion and suggests future work. For illustration purposes, a video demonstrating the use of Scan is available at `https://staff.info.unamur.be/jmj/Scan/`. A link is also proposed there to download the workbench.

2 Scan design and overview

Following Linda, the 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 [26]. As a natural consequence, following the blackboard metaphor [13], according to which a group of specialists iteratively updates knowledge on a blackboard starting from a problem specification, Scan 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 interact directly through the `tell`, `get` and `clear` buttons. Moreover, it offers to create four types of processes.

The first two processes, named respectively *Autonomous Agent* and *Interactive Agent*, allow the user to enter instructions in Bach and to execute them. As depicted in part (a) of Figure 3, windows of the first kind, perform computations step-by-step by letting the user choose which primitives to execute. In contrast, as shown in part (b) of Figure 3, windows of the second type execute computations in one run if the `run` button is activated or step by step if the `next` button is selected but in both cases with the Scan workbench deciding (in a random manner) 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.

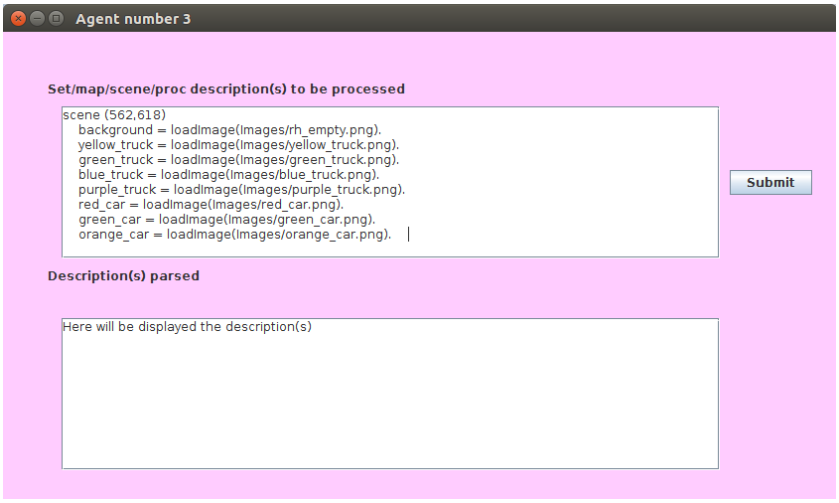
(a) The interactive agent window



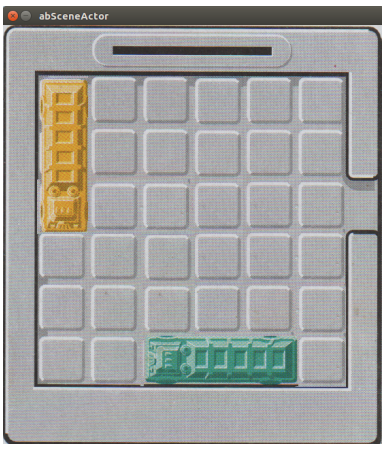
(b) The autonomous agent window

Fig. 3. Interacting with the blackboard

The facilities offered by the interactive and autonomous agents are nice to debug, at a low level, concurrent executions executed around the shared space, possibly deadlocking on data not being available. However, they do not provide much insights on whether what is described in 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, Scan provides animations through a third kind of processes launched by the `new description` button (see Figure 2). As shown in part (a) of Figure 4, such animations are obtained by describing a so-called scene from a set of pictures which are handled by means of primitives for inserting them on the scene at specific places, making them visible or invisible, and making them move to specific places. In doing so, these primitives allow to draw and animate, at a



(a) The description window



(b) The scene window

Fig. 4. Animation

high-level, pictures such as the one of part (b) of Figure 4. Note that, as these primitives may be inserted inside instructions of autonomous agents, the concurrent execution of these agents provides dynamic simulations of the problem under consideration.

Although nice, simulating graphically systems does not necessarily provide a solution to the problem under consideration. The rush hour problem is a clear example of that. To that end, the Scan workbench offers a fourth type of processes, materialized by the `new model checker` button of Figure 2 which gen-

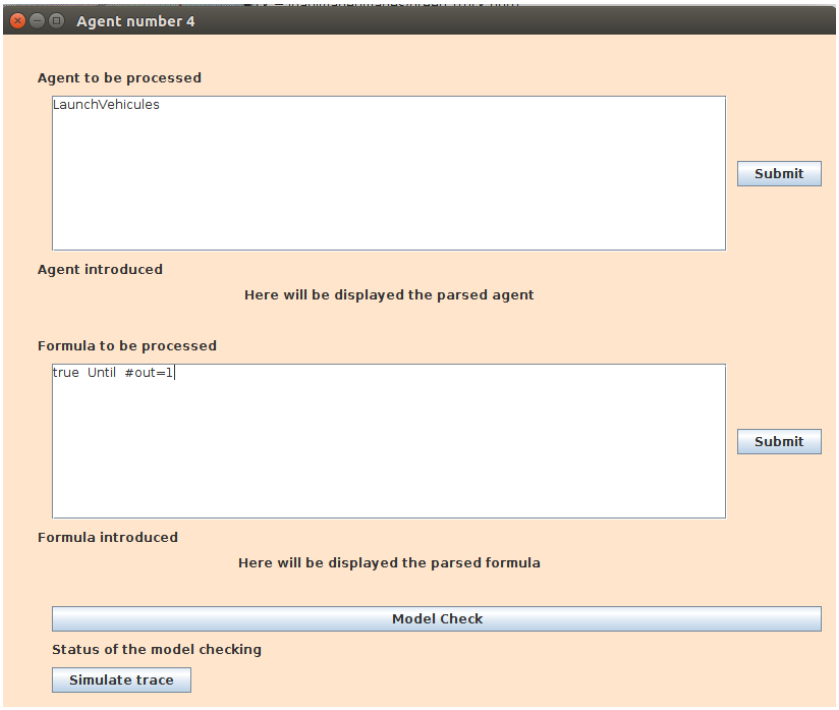


Fig. 5. Model-checking

erates windows of the type depicted in Figure 5. As illustrated in this figure, Scan allows to verify formulae written in a fragment of linear temporal logic, to determine traces of execution that establish the formulae and to replay these traces, including the primitives that generate animations.

Although simple we believe that the Scan 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 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.

3 The Anim-Bach language and its temporal logic

The facilities offered by Scan being described, let us turn to the process algebra to be used in the workbench. This algebra is subsequently referred to as Anim-Bach.

3.1 Definition of data

Following Linda, the Bach language [10, 16] 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

```
set Cols = { 1, 2, 3, 4, 5, 6}.
    Rows = { 1, 2, 3, 4, 5, 6}.
```

As the reader will have easily noticed, these two sets allow to identify an element of the grid of the rush hour example 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.

The fact that sets are written as enumerations reflects the fact that the elements are naturally ordered by their order of appearance, which then allows to compare them. Moreover, they implicitly define the `pred` and `succ` functions, providing respectively the predecessors and successors of elements (if any).

In addition to sets, maps can be defined between them as functions that take zero or more arguments. In practice, Scan 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 of a map, 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 : Rows -> Rows.
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. As an example, in the rush hour example, it is convenient to represent the free places of the game as pieces of information of the form `free(i,j)` with i a row and j a column.

In summary of this subsection, we may assume subsequently to be defined a series of sets, a series of mappings, and a set of structured pieces of information, say $\mathcal{I}$. Thanks to the mapping definitions, we additionally assume a rewriting relation $\rightsquigarrow$ that rewrites any mapping expression into a set element. With this defined, we can proceed with the definition of agents in Anim-Bach.

3.2 Agents

The primitives of Anim-Bach consist of the `tell`, `ask`, `nask` and `get` primitives already mentioned for Bach, which take as arguments elements of $\mathcal{I}$. They 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 another mechanism: conditional statements of the form $c \rightarrow s_1 \diamond s_2$, which computes s_1 if c evaluates 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 =, <, <=, >, >=).

This being given, the statements of the Anim-Bach language, also called agents by abuse of language, consist of the statements A generated by the following grammar:

$$A ::= Prim \mid Proc \mid A\ ;\ A \mid A \,||\, A \mid A\ +\ A \mid C \rightarrow A \diamond A$$

where $Prim$ represents a primitive, $Proc$ a procedure call and C a condition.

Procedures are defined similarly to mappings through the `proc` keyword by associating an agent with a procedure name. As in classical 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: Rows, c: Cols) =
  ( (r>1 & r<5) -> ( get(free(pred(r),c)); tell(free(succ(r),c);
                     VerticalTruck(pred(r),c) )
  +
  ( (r<5) -> ( get(free(down_truck(r),c)); tell(free(r,c));
               VerticalTruck(succ(r),c)) ).
```

The operational semantics of primitives and complex agents are respectively defined through the transition rules of Figures 6 and 7. Configurations consist of agents (summarizing the current state of the agents running on the store) and a multi-set of structured pieces of information (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 shall always rewrite agents of the form $(E; A)$, $(E \,||\, A)$ and $(A \,||\, E)$ as A.

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

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

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

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

Fig. 6. Transition rules for the primitives

The rules of Figure 6 follow the intuitive description of the primitives. Note however that before being processed, a structured piece of information t is rewritten in u by means of the rewriting relation $\rightsquigarrow$.

The rules of Figure 7 are quite classical. Rules (S), (P) and (C) provide the usual semantics for sequential, parallel and choice compositions. 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. Finally, 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 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 images, to make them appear or disappear and to move them.

The description of a scene is obtained by defining the size of the canvas to be used by the animation, the background image of the animation and a series of images to be used. Such a definition takes the following form:

```
scene (640,640)
  background = loadImage(Images/the_background_img.png).
  red_car = loadImage(Images/rcar.jpg).
  yellow_truck = loadImage(Images/ytruck.gif).
```

where the file names are given with respect to the path in which Scan is executed.

Images are manipulated by means of the following primitives where coordinates are expressed in pixels with respect to the canvas, with $(0, 0)$ being the upper-left corner of the canvas:

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

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

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

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

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

Fig. 7. Transition rules for the operators

- `place_at(i,x,y)`: to place image identified by i at the coordinates (x, y)
- `move_to(i,x,y)`: to move image identified by i from its current position to the new coordinates (x, y)
- `hide(i)`: to hide image identified by i
- `show(i)` : to make appear image identified by i

Such primitives are added to the tell, get, nask and ask primitives in the definition of Anim-Bach.

It is worth observing that the map constructs (introduced before) allow to declare coordinates in a symbolic manner making it easy to specify the position of images.

3.4 A fragment of temporal logic

Linear temporal logic is a logic widely used to reason on dynamic systems. The Scan workbench uses a fragment of PLTL [12] with, as main goal, to check the reachability of states.

As usual, the logic employed 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 formulae 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 Scan is then defined by the following grammar :

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

where PF is a propositional formula. As will be explained in the next section, it has been designed so as to allow for an efficient implementation.

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

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

4 Implementation

The Scan workbench has been implemented in Scala [24] on top of the Processing library [27]. Scala is a programming language which combines the object-oriented and functional paradigms and 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 Anim-Bach language, which as can be appreciated by 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 Scan. 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 enter deeply in the code of the implementation. However, the following subsections should allow the reader to understand the key elements of our implementation.

4.1 Internal representation of data

Scala case classes offer an elegant mechanism to represent data in an internal manner while keeping a close link to the textual representation in Anim-Bach. For instance, an abstract class `AB_AG` has been introduced to represent agents of Anim-Bach. Case classes have then been defined to represent particular agents, such as `AB_AST_Empty_Agent()` to represent the empty agent, `AB_AST_Primitive(primitive: String, stinfo: AB_SI_ELM)` to represent a primitive or `AB_AST_Agent(op: String, agi: AB_AG, agii: AB_AG)` to represent a composed agent using the operator `op` – for instance, `||` for the parallel composition – and two subagents `agi` and `agii`.

Other structures are used similarly to code sets, structured pieces of information, map equations, and temporal logic formulae.

As might be appreciated by this brief description, a close link is indeed made between the internal representation and the textual description in Anim-Bach. As a result, in contrast to tools such as mCRL2 [5], it is quite easy to provide the user with messages directly connected to what he has written.

4.2 Parsing Anim-Bach constructs

As exposed in chapter 33 of [24], Scala offers facilities to parse languages. The main ingredients to do so are, on the one hand, a library to define parsers, which basically allows to define the class AnimBachParsers as inherited from the class RegexParsers, parsing regular expressions, and the possibility of applying functions to the result of strings having been parsed.

4.3 The store

The store is implemented as a mutable map in Scala. Initially empty, it is enriched for each told structured piece of information by an association of it to a number representing the number of its occurrences on the store. The implementation of the primitives follows directly from this intuition. For instance, the execution of a tell primitive, say `tell(t)`, consists in checking whether `t` is already in the map. If it is then the number of occurrences associated with it is simply incremented by one. Otherwise a new association `(t,1)` is added to the map. Dually, the execution of `get(t)` consists in checking whether `t` is in the map and, in this case, in decrementing by one the number of occurrences. In case one of these two conditions is not met then the get primitive cannot be executed.

The declaration of sets, map equations and procedure definitions are memorized similarly through maps or lists for equations.

4.4 The simulator

The simulator consists in repeatedly executing transition steps. In our implementation, this boils down to the definition of function `run_one`, which assumes given an agent in a parsed form and which returns a pair composed of a boolean and an agent in parsed form. The boolean aims at specifying whether a transition step has taken place. In this case, the associated agent consists of the agent obtained by the transition step. Otherwise, failure is reported with the given agent as associated agent.

The function is defined inductively on the structure of its argument, say `ag`. If `ag` is a primitive, then the `run_one` function simply consists in executing the primitive on the store. If `ag` is a sequentially composed agent ag_i ; ag_{ii}, then the transition step proceeds by trying to execute the first subagent ag_i. Assume this succeeds and delivers ag' as resulting agent. Then the agent returned is ag' ; ag_{ii}

in case ag' is not empty or more simply ag_{ii} in case ag' is empty. Of course, the whole computation fails in case ag_i cannot perform a transition step, namely in case `run_one` applied to ag_i fails.

The case of an agent composed by a parallel or choice operator is more subtle. Indeed for both cases one should not always favor the first or second subagent. To avoid that behavior, we use a boolean variable, randomly assigned to 0 or 1, and depending upon this value we start by evaluating the first or second subagent. In case of failure, we then evaluate the other one and if both fails we report a failure. In case of success for the parallel composition we determine the resulting agent in a similar way to what we did for the sequentially composed agent. For a composition by the choice operator the tried alternative is simply selected.

The computation of a procedure call and of a conditional statement are performed similarly as one may expect.

4.5 The scene

The scene and its animation are implemented by means of Processing. The declaration of a scene induces dedicated declarations in the `setup` method used by Processing. Moving an image is obtained by an update in the `draw` method employed by Processing using a linear interpolation of the initial and final coordinates. Placing images and hiding or showing them is achieved by modifications of the corresponding variables and attributes.

4.6 Temporal formulae

Scan temporal formulae are verified by means of a home-made program inspired by the techniques proposed in [28]. It essentially uses a limited depth-first search algorithm based on the simulator described in Subsection 4.4 with a recursive reasoning on the temporal formulae. More concretely, the key function `check_lts` takes as arguments an integer I, a temporal formula F, an agent A and a trace T. The first argument is the length of the remaining search allowed. The second and third arguments are the temporal formula to be checked against the agent. The path consists of the trace prefix already computed. The function returns a boolean, stating whether the formula has been checked, together with a path, describing the last path explored. It provides an execution witness of the truth of the formula in case the return boolean is true.

The `check_lts` function is coded by using a recursive reasoning on the formula F:

- if F is a propositional formula, then the current contents of the store should verify it. If this is the case, `true` is returned together with the path P. Otherwise, `false` is returned together with P.
- if F is of the form *Next* TF and if I is strictly positive, then `check_lts` is successively called on the list of next possible agents returned by `run_one` with $I-1$ as integer, the agent produced by `run_one` as agent, TF as formula and P augmented with a reference to the computation step as path. In case

one of these calls succeed, namely returns `true` with the associated path, then this result is returned. Otherwise or in case $I = 0$, then `false` is returned together with the path P.
- the case where F is of the form PF *Until* TF is treated similarly. Either TF holds on the current store, in which case `true` is returned together with P, or PF holds in the current store and there is one successor agent (explored as for the above case) for which TF holds. In this latter case, `true` is returned together with the discovered path. In case none of the two situations holds, then `false` is returned with the path P.

It is worth noting that the algorithm is not complete. If it returns `true` then the considered formula has been established and the returned path provides a witness execution that can be replayed. Otherwise, because of the limited depth-first search, `false` may be returned wrongly because the formula could have been proven by using a more exhaustive search. Nevertheless such a simple algorithm is in practice already useful to establish formulae.

Note also that, in case of success, the algorithm is sound because of the limited form of the temporal formulae considered in Scan, which in particular, does not involve negations.

5 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 [18], ReoLive [6], and TAPAs [3].

The Extensible Coordination Tools (ECT) has been developed for the control-based coordination language Reo, a language quite different from Anim-Bach. ECT consists 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 [5]. 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 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 did with Anim-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 [3] 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 Anim-Bach for animations. In contrast, as written above, model checking in Anim-Bach is quite simple and is much less elaborated than that of TAPAs. Future work will aim at improving this aspect.

Declarative invariant assertions are proposed in [20] 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 Anim-Bach, our work differs in two main respects. On the one hand, assertions in [20] 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 [25] does not provide facilities to animate computations nor to model-check them.

Finally, a Linda workbench is presented in [11] 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 hence 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.

6 Conclusion

The paper has introduced a workbench for reasoning on a Linda-like coordination language at three levels: (i) by executing 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 [19].

References

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

2. Brogi, A., Jacquet, J.M.: On the Expressiveness of Coordination via Shared Dataspaces. Science of Computer Programming **46**(1-2), 71–98 (2003)
3. Calzolai, F., De Nicola, R., Loreti, M., Tiezzi, F.: TAPAs: A Tool for the Analysis of Process Algebras. In: Jensen, K., van der Aalst, W., Billington, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency. Lecture Notes in Computer Science, vol. 5100, pp. 54–70. Springer (2008)
4. Carriero, N., Gelernter, D.: Linda in Context. Communications of the ACM **32**(4), 444–458 (1989)
5. Cranen, S., Groote, J., Keiren, J., Stappers, F., de Vink, E., Wesselink, W., Willemse, T.: An Overview of the mCRL2 Toolset and Its Recent Advances. In: Piterman, N., Smolka, S. (eds.) Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7795, pp. 199–213. Springer (2013)
6. Cruz, R., Proença, J.: ReoLive: Analysing Connectors in Your Browser. In: Mazzara, M., Ober, I., Salaün, G. (eds.) Proceedings of the STAF collocated workshops. Lecture Notes in Computer Science, vol. 11176, pp. 336–350. Springer (2018)
7. Darquennes, D., Jacquet, J.M., Linden, I.: On Density in Coordiantion Languages. In: Canal, C., Villari, M. (eds.) CCIS 393, Advances in Service-Oriented and Cloud Computing, ESOCC 2013, Proceedings of Foclasa Workshop. pp. 189–203. Springer, Malaga, Spain (2013)
8. Darquennes, D., Jacquet, J.M., Linden, I.: On the Introduction of Density in Tuple-Space Coordination Languages. In: Science of Computer Programming. Springer (2013)
9. Darquennes, D., Jacquet, J.M., Linden, I.: On Distributed Density in Tuple-based Coordination Languages. In: Cámara, J., Proença, J. (eds.) Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems. EPTCS, vol. 175, pp. 36–53. Springer, Rome, Italy (2015)
10. Darquennes, D., Jacquet, J.M., Linden, I.: On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study. In: Serugendo, G.D.M., Loreti, M. (eds.) Proceedings of the 20th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 10852, pp. 81–109. Springer (2018)
11. Dukielska, M., Sroka, J.: JavaSpaces NetBeans: a Linda Workbench for Distributed Programming Course. In: Ayfer, R., Impagliazzo, J., Laxer, C. (eds.) Proceedings of the 15th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. pp. 23–27. ACM (2010)
12. Emerson, E.A.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), pp. 995–1072. Elsevier (1990)
13. Erman, L., Hayes-Roth, F., Lesser, V., Reddy, D.: The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. ACM Computing Surveys **12**(2), 213 (1980)
14. Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Communications of the ACM **35**(2), 97–107 (1992)
15. Jacquet, J.M., Bosschere, K.D., Brogi, A.: On Timed Coordination Languages. In: Porto, A., Roman, G.C. (eds.) Proc. 4th International Conference on Coordination Languages and Models. Lecture Notes in Computer Science, vol. 1906, pp. 81–98. Springer (2000)
16. Jacquet, J.M., Linden, I.: Coordinating Context-aware Applications in Mobile Adhoc Networks. In: Braun, T., Konstantas, D., Mascolo, S., Wulff, M. (eds.) Pro-

ceedings of the first ERCIM workshop on eMobility. pp. 107–118. The University of Bern (2007)
17. Jacquet, J.M., Linden, I.: Fully Abstract Models and Refinements as Tools to Compare Agents in Timed Coordination Languages. Theoretical Computer Science **410**(2-3), 221–253 (2009)
18. Kokash, N., Arbab, F.: Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools. IEEE Transactions on Services Computing **6**(2), 186–200 (2013)
19. Kozen, D.: Results on the Propositional mu-Calculus. Theoretical Computer Science **27**, 333–354 (1983)
20. Kühn, E., Radschek, S., Elaraby, N.: Distributed Coordination Runtime Assertions for the Peer Model. In: Di Marzo Serugendo, G., Loreti, M. (eds.) Proceedings of the 20th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 10852, pp. 200–219. Springer (2018)
21. Linden, I., Jacquet, J.M.: On the Expressiveness of Absolute-Time Coordination Languages. In: Nicola, R.D., Ferrari, G., Meredith, G. (eds.) Proc. 6th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 2949, pp. 232–247. Springer (2004)
22. Linden, I., Jacquet, J.M.: On the Expressiveness of Timed Coordination via Shared Dataspaces. Electronical Notes in Theoretical Computer Science **180**(2), 71–89 (2007)
23. Linden, I., Jacquet, J.M., Bosschere, K.D., Brogi, A.: On the Expressiveness of Relative-Timed Coordination Models. Electronical Notes in Theoretical Computer Science **97**, 125–153 (2004)
24. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, A comprehensive step-by-step guide. Artemis (2016)
25. Omicini, A., Ricci, A., Rimassa, G., Viroli, M.: Integrating Objective & Subjective Coordination in FIPA: A Roadmap to TuCSoN. In: Armano, G., Paoli, F.D., Omicini, A., Vargiu, E. (eds.) Proceedings of the 4th AI*IA/TABOO Joint Workshop "From Objects to Agents": Intelligent Systems and Pervasive Computing. pp. 85–91. Pitagora Editrice Bologna (2003)
26. Papadopoulos, G., Arbab, F.: Coordination Models and Languages. In: Technical Report SEN-R9834. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X (1998)
27. Reas, C., Fry, B.: Processing: A Programming Handbook for Visual Designers. The MIT Press (2014)
28. Reynolds, M.: A Traditional Tree-style Tableau for LTL. CoRR **abs/1604.03962** (2016)