An Experimental Framework for Behavior Extraction over Communication Flows in Multi-Agent Systems

Jean-Paul Sansonnet, Erika Valencia

LIMSI-Cnrs, Bât 508, University of Paris XI,
B.P. 133 F-91403 Orsay Cedex
phone +33 (0) 1 69 85 80 12 fax +33 (0) 1 69 85 80 12 mail {jps,erika}@limsi.fr

ABSTRACT

In this paper, we are interested in extracting agent's behaviors from regularities occurring in sustained communication conversations. In this study we are dealing with a restricted conversation model using only three basic speech acts (Ask, Tell and Satisfy) and simplified interactions. The experimental framework is based on reactive agents, viewed as concurrent communicating processes. This framework embeds a generic mechanism to register execution traces of communicating flows. This mechanism 1) captures the contextual information associated with each interaction 2) registers it into so-called *chronicles* attached to the communication channels of each agent 3) makes use of an induction tool to compact the chronicles into an intensionalized form. The agents, together with their intensionalized chronicles, turn into reflexive entities, as far as communication is concerned. Indeed, they can build a representation of the interactional behavior of other agents, achieve some causal reasoning about those behaviors and act at a higher semantic level.

KEY WORDS : Communication Flows, Interactional Behavior Modeling, Execution Traces, Chronicles, Intensionalization, Causal Reasoning.

1. INTRODUCTION

1.1 The primacy of interaction flows.

Research work in Multi-agent Systems (MAS) has long been based on methodological dichotomies like reactive *vs* cognitive agents [1], agent/environment [2] etc. Now it seems to focus on the separation between agent technology [3] and agents interaction [4] which emphasizes the importance of *interaction flows*. The study of interaction flows is currently carried out at several levels: in reactive systems [5], in macroscopic work-flow analysis systems [6], and recently, at a more cognitive level, with the emerging concept of dialogical agents [7]. These various approaches share the claim that interactions must be considered as first class citizens. In this paper, we propose a model for extracting behaviors from sustained interaction flows and performing some causal reasoning operations over them.

1.2 The need for dynamical analysis in behavior extraction.

Extracting or even proving the behavior of a program has issued in two main approaches: static and dynamic analysis. Static analysis has been used in functional programming and database systems [8]; situation calculus [9] and temporal logic [10] have been used in artificial intelligence. However the proposed models are static and apply only to semantically closed worlds. In the framework of MAS, we have to deal with dynamically created interaction flows between agents in an *open* world. So we just cannot work on static code to infer behaviors "at compile time", we have to work on dynamic observations to infer behaviors at runtime. Therefore, we propose a dynamical analysis of *observed* interactions. This approach, has already been tackled in the MAS framework [11, 23, 24] and in sequences recognition [12, 26].

1.3 MAGENTA: a simple test-bed for interaction flows.

Our primary interest being the study of interaction flows, we developed a simple test-bed called MAGENTA[1], based on the well proven CSP formalism of C.A.R. HOARE [13]. In our model, agents are concurrent dynamical processes, interacting *physically*, *continuously* and *regularly* on internals of each other, producing flows of events, in the sense of SIMD data-flow languages [14]. The originality of MAGENTA is that it provides an automatic mechanism for registering flows of events into *observations*.

Let A be an agent undergoing a series of updates on some of its attributes, registered into the observation (U_i). Typically, agent A will have to wonder "What is happening?" and to extract from (U_i) the behavior of other agents of the world, the environment, or even itself, which is consistent with (U_i). This issue is closely related with techniques of intensionalization of series and, more generally, machine learning [15] and law discovery [16] which remain open questions in artificial intelligence. This is the reason why we have restricted our study 1) to four basic interactional operators, and 2) to the intensionalization of arithmetic series. In spite of these restrictions,

[1] MAGENTA stands for "Agents in MATHEMATICA". The testbed has been developed with the MATHEMATICA symbolic processing language of Wolfram Research and it makes intensive use of its formal features, especially rewriting rules.

we could show interesting results of causal reasoning over interaction flows.

2. A MODEL BASED ON GUARDED DYNAMICAL PROCESSES

2.1 Notations

In MAGENTA, the agent world, noted Ω, is viewed as a set of single agents: $\Omega = \langle A_i \rangle$. The environment is modeled like an ordinary agent $A_\varepsilon \in \Omega$. Agents can't have a recursive structure. Formally, a MAGENTA agent a is a couple of sets : a set of definitions Φ_a and a set of processes Π_a. The internal structure of an agent can be represented as

$$Agent \equiv \{ \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ ... \\ \varphi_n \end{bmatrix}, \begin{bmatrix} \pi_1 [\gamma_1, \sigma_1] \\ \pi_2 [\gamma_2, \sigma_2] \\ \pi_3 [\gamma_3, \sigma_3] \\ ... \\ \pi_m [\gamma_m, \sigma_m] \end{bmatrix} \} \,;\, n, m \geq 0.$$

where φ_i are definitions and π_i are processes types wrapping guards γ_i and scripts σ_i. The external syntax[2] of an agent definition is

$$\varphi \equiv \texttt{Define}[\text{AgentSymbol}^\dagger, \text{ Rule } |\{\text{Rule},*\}]$$

where Rule is a rewriting rule of general form $pat = expr$ where pat is a pattern expression[3] and $expr$ a non-pattern expression. The `Define` operator links the given Rules to the AgentSymbol. Rewriting rules are a simple and powerful tool for representing into a unified formalism:

1. the internal state of agents as simple attributes: $symbol = value$,
2. the capacities of agents as functions: $f[args] = body$,
3. knowledge about other agents in Ω: $Bel_x = proposition$.

2.2 Guarded Processes

The external syntax of a process $\pi_i [\gamma_i, \sigma_i]$ is

$$\pi \equiv \texttt{DefineProcess}[\text{AgentSymbol}^\dagger, \text{ ProcessSymbol}^\dagger, \text{Guard}^\dagger, \texttt{Script}[\text{AtomicAction}]]$$

The `DefineProcess` operator links the Guard and the Script to ProcessSymbol which in turn is associated with AgentSymbol. A guard contains a predicate whose variables are definitions belonging to agents of Ω. For a given process $\pi_i [\gamma_i, \sigma_i]$, the guard γ_i controls the activation of the associated script σ_i when the process π_i is selected by the interpreter. Semantically, guards are categorized into three main types

- `Always`[4] is a constant guard whose associated script is always activated,
- `When`[Predicate] the associated script is activated iff the predicate evaluates to `True`,
- `Goal`[Predicate] the associated script is activated iff the predicate evaluates to `False`.

At the internal semantic level :

$$\texttt{When}[\text{Predicate}] = \neg\texttt{Goal}[\text{Predicate}]$$

However we will need to make the distinction at the causal level (See section 4.1).

2.3 Interactional Scripts

In our model, simple agents interaction is carried out through scripts. A script is an expression with type `Script` containing a single atomic interactional operation. When a script is activated by the MAGENTA interpreter, its content is evaluated as an atomic operation, i.e. it is not interleaved[5] with other processes execution.

In the first version of our model, four types of interactional operators are taken into account within scripts so that an agent $a \in \Omega$ can mainly 1) read, 2) write into, 3) call as a function, the right part of a definition of another[6] agent $b \in \Omega$. Let a be the symbolic name of the agent performing the script and b be the symbolic name of the agent undergoing the script, and pat a pattern, then:

- `Ask`[b, pat] allows a to consult the left term of the definition matching pat in b.
- `Tell`[b, pat, $expr$] allows a to update the left term of the definition matching pat in b with the result of the evaluation of $expr$. If no definition matches pat in b, then a new definition $pat = expr$ is created and linked to b.
- `Achieve`[b, pat] allows a to launch the evaluation of the left term of the definition matching pat in b. The underlying rule system takes in charge actual to formal parameter passing.
- `Satisfy`[b, γ] performs an incremental alteration of the state of b in order "tend to deactivate the guard γ" (See details in section 4.3).

2.3.1 *Syntactical abbreviation.*

`Ask`[b, pat] can be expressed $b[pat]$ and, within the scope of agent b, even more simply pat. `Tell`[b, pat, $expr$] can be expressed $b[pat] = expr$ and, within the scope of b, $pat = expr$ (note that being within the type `Script` "=" denotes here an update operation and not a definition).

[2]In this paper, Keywords are expressed in the typewriter font (e.g. `Define`). The notation {item,*} stands for a (possibly empty) list of items separated by "," and item† indicates an argument that can be omitted, provided there is no ambiguity in the context.

[3]It can be a simple symbol (viewed as a constant pattern) or an arbitrary complex pattern expression.

[4]When omitted into a process definition, a guard has the constant value `Always`.

[5]In [22], a coroutined version of the interpret was developed, where interleaving *within* scripts is provided but it is not detailed here because this feature is not really useful for our main purpose.

[6]a can be equal to b, when an agent acts upon itself.

2.3.2 Tuning the granularity of interaction.

Within a `Tell` operator, $expr$ can recursively contain interactional operators. However, as a script is evaluated atomically, all the interactional operations performed within the script will take place at the same logical step and will not be "seen" by the observational mechanism of MAGENTA described below. In this way, the ability of tuning the granularity of an observable interaction is offered to the designer: he can either explicit an interaction, making it a `Script` or hide an interaction, putting it into an internal $expr$.

2.4 The Execution Model

In the MAGENTA model, agents have a static structure (definitions) and a dynamic behavior (guarded processes). At runtime, the active processes (the ones whose guard evaluates to `True`) operate on the static structure of agents in Ω. The execution model of MAGENTA is based on the concept of stochastic iteration. It is a synchronic model based on discrete execution steps. Processes are iterated and interleaved, down to the script level, thanks to a random choice between the activable scripts at the current step. Therefore, a new time step, noted θ in the following, is generated each time an atomic action (a script) is performed. A simplified version of the execution algorithm is given below

LET A **BE** the list of active agents in Ω, N **BE** the maximum number of execution cycles, i **BE** the i^{th}execution cycle and θ **BE** the logical time

WHILE $A \neq \emptyset \wedge i \leq N$

an agent a^* is chosen randomly in A

IF a^* has at least one process

THEN a process π^* is chosen randomly in a^*

IF the guard of π^* evaluates to `True`

THEN the script of π^* is evaluated;

θ is incremented

i is incremented

2.4.1 Management of the phase space.

Among the already proposed execution models for MAS, most popular are the ATN-based models where the state space is a finite set of nodes with deterministic transitions of the form request→reaction. Our model is based on guarded processes, as in HOARE CSP [13] but here, guards can either be programmed as a logical *partition* or *covering* over all the possible state of agents in Ω (the so-called *phase space* of the dynamical system). When guards partition the phase space, the model is consistent with the deterministic ATN model; when guards are a covering of the phase space the model is more versatile and more general (it can take into account not only single cases but classes of states), at the price of non complete predictability.

3. OBSERVATION OF AGENTS' INTERACTIONAL ACTS

3.1 Reflected Acts

If we consider the definition of the four interactional operators described in section 2.3 we can make three main criticisms from a conventional MAS point of view

- the operators are mandatory: an agent cannot prevent another agent from reading and writing into its body.
- the references are global: an agent must know about the symbolic names of the agents it interact with (i.e. Ω is closed) and, more important, it must know about their internals.
- the operators are similar to the remote control operators in Distributed Components Systems. As Searle pointed out [17], such entities "are not aware of their actions".

We are currently working on the first two points: a system managing *permissions* is studied involving the implementation of the BEG speech act. As for the "knowing about internals" problem [18], we are developing analogical reasoning tools which make intense use of the patterned structure of definitions in order to resolve semantically heterogeneous interactions. In this paper, we will focus on the third point: how is it possible to make obvious the behavior implicitly embedded in interacting processes.

What we really need is a class of *reflected acts* where the agents are provided, in parallel with the execution of an operator, with the representation of "what is happening". Therefore, each occurrence of a script activation will be captured and registered into an *observation.* In the following, we will call "reflected act" or simply "act" the conjunction of the *execution* and the *observation* of an activated script.

For an activated script σ the three main semantical parts of its observation $\Theta(\sigma)$ are

3.1.0.1 1. The situation of the act:

when a script is activated, we catch the contextual information about

- which agents are concerned: source and destination agents,
- when the act is performed: execution step θ,
- what action is performed: elementary interactional operation.

3.1.0.2 2. The reason of the act:

in order to catch the reason why the script was activated, we take into account the guard of the process containing the currently executed script. Thus, one can see that our guards achieve two purposes: they control the activability of scripts and they provide knowledge about the causal nature of acts (`Always`, `When`, `Goal`).

```
W1 = AgentWorld[
   Agent[A, Define[x=0]],
   Agent[B, DefineProcess[p, Goal[A[x]>=20], Script[A[x]=A[x]+2]]]
   ];
Exec[W1,50];Body[A,B]
```

```
============================== TETA: 10 BODY: A =============================
x:     10
CHin:  WHO  WHAT   WHEN  WHY              ACTION       VALUE
       B    Tell   1     Goal[A[x]>=20]  A[x]=A[x]+2  2
       B    Tell   2     Goal[A[x]>=20]  A[x]=A[x]+2  4
       B    Tell   3     Goal[A[x]>=20]  A[x]=A[x]+2  6
       B    Tell   4     Goal[A[x]>=20]  A[x]=A[x]+2  8
       B    Tell   5     Goal[A[x]>=20]  A[x]=A[x]+2  10
       B    Tell   6     Goal[A[x]>=20]  A[x]=A[x]+2  12
       B    Tell   7     Goal[A[x]>=20]  A[x]=A[x]+2  14
       B    Tell   8     Goal[A[x]>=20]  A[x]=A[x]+2  16
       B    Tell   9     Goal[A[x]>=20]  A[x]=A[x]+2  18
       B    Tell   10    Goal[A[x]>=20]  A[x]=A[x]+2  20
Kin:   Kin[B[Goal[A[x]<=20][Tell[A[x]=A[x]+2[Range[0,10][Range[0,20,2]]]]]]

============================== TETA: 10 BODY: B =============================
p:     GUARD: Goal[A[x]==7]   SCRIPT: A[x]=A[x]+1
CHout: WHO  WHAT   WHEN  WHY              ACTION       VALUE
       A    Tell   1     Goal[A[x]>=20]  A[x]=A[x]+2  2
       A    Tell   2     Goal[A[x]>=20]  A[x]=A[x]+2  4
       A    Tell   3     Goal[A[x]>=20]  A[x]=A[x]+2  6
       A    Tell   4     Goal[A[x]>=20]  A[x]=A[x]+2  8
       A    Tell   5     Goal[A[x]>=20]  A[x]=A[x]+2  10
       A    Tell   6     Goal[A[x]>=20]  A[x]=A[x]+2  12
       A    Tell   7     Goal[A[x]>=20]  A[x]=A[x]+2  14
       A    Tell   8     Goal[A[x]>=20]  A[x]=A[x]+2  16
       A    Tell   9     Goal[A[x]>=20]  A[x]=A[x]+2  18
       A    Tell   10    Goal[A[x]>=20]  A[x]=A[x]+2  20
Kout:  Kout[A[Goal[A[x]<=20][Tell[A[x]=A[x]+2[Range[0,10][Range[0,20,2]]]]]]
```

Table 1: The AgentWorld `W1` has only two agents : B acts upon A by incrementing its attribute x with the intended goal that x reaches the value 20. Exec[W1,50] performed 50 cycles of execution. After that, Body[A,B] displayed the internals of both agents. The display of the body shows in order, from the top of the figure: The title bar with the current logical time and the agent name; the definitions (if any); the processes (if any). Then, the chronicles are given (if not empty): CHIN, Kin, CHOUT, Kout. During the first 10 cycles of execution, the process p of agent B has its guard evaluated to True and the associated script is performed. From cycle 11 to 50 nothing happens because the guard of p being satisfied no script is activable. Hence, at the end of the execution, the value of x is 20, and the value of θ is 10. The interaction between B and A was symmetrically recorded in the CHIN of A and the CHOUT of B; these chronicles are intensionalized in their compacted counterparts Kin and Kout.

3.1.0.3 3. The valuation of the act:

after the activation of the script, we try to make a measure of its success as an interactive act. This depends on its operator:

- `Ask`[a, pat] $\Rightarrow$ IF there is a definition $pat = value$ matching pat in a THEN $value$ ELSE `NotFound`[pat].
- `Tell`[a, pat, $expr$] $\Rightarrow$ EVAL[$expr$]
- `Achieve`[a, pat] $\Rightarrow$ IF there is a definition $pat = body$ matching pat in a
  THEN EVAL[$body$ /. pat]
  ELSE `NotFound`[pat].
- `Satisfy`[a,γ] $\Rightarrow$ IF $\gamma_{\theta+1} = \neg\gamma_\theta$ THEN `True` ELSE `False`.

3.2 Structure of the Chronicles

When a script σ is activated its related observation $\Theta(\sigma)$ is captured and subsequently stocked into a special data structure called a *chronicle*. Every agent a is equipped with two chronicles: the output chronicle of a, named CHOUT_a, that memorizes the observations of acts performed by a (upon other agents or itself); and the input chronicle of a, named CHIN_a, that memorizes the observations of acts performed upon a (by other agents or itself). A chronicle can be viewed as a table whose rows are observations. An observation $\Theta(\sigma)$ is composed of six criteria listed below

Criteria	Semantics
`Who`	Agent performing the act in CHin
`What`	Type of act (`Ask`, `Tell`, ...)
`When`	Logical time θ of the act
`Why`	Guard triggering the act
`Action`	Operator within the script
`Value`	Measure of success after execution

3.3 Intensionalization of Agents' Chronicles

The example of Agent World `W1`, given in table 1, shows the CHIN and CHOUT chronicles constructed at runtime in order to track a very simple interaction flow: an agent B continuously increments the attribute x of another agent A, until the value of x satisfies the goal of B. In this example, process p of agent B became inactive after 10 steps. Assume now the goal of B be `Goal[x>=1000000]`: we can easily see that chronicles CHIN and CHOUT will grow up to 1000000 rows and remark that they will contain little difference in information due to the fact that the interaction flow upon A is very regular. Consequently, we are confronted with two closely related questions: 1) is it possible to build a more compact representation of chronicles? 2) is it possible to extract from chronicles some regularities upon interaction flows in order to detect behaviors?

3.3.1 Principle of intensionalization.

From a formal point of view, an entity can be represented either in *extension* when it is explicitly given (e.g. `L={1,2,3,4}`) or in *intension* when it is defined by either a predicate (e.g. $i^2 = -1$) or an algorithm (constructive definition). Here, we will obviously adopt a constructive approach and try to transform CHIN and CHOUT, considered as extensions, into their intensional counterparts called Kin and Kout. Moreover, in order to overcome the memory overflow problem, we don't want to deal with extensional CHIN and CHOUT chronicles anymore; we want them to be directly replaced with Kin and Kout[7].

We give now the four main properties of the MAGENTA intensionalizer (see [19] for more details):

3.3.2 Property 1: it is an algorithm.

Unless heuristic approaches, mainly used in information compacting techniques, the intensionalizer keeps all the information about observations and consequently, it is possible to reproduce exactly the extensional chronicle CHIN from Kin (resp. CHOUT from Kout).

3.3.3 Property 2: it has a static ontology.

The basic structure of an intensional chronicle is a decision tree as used in machine learning applications [20]. The levels of the decision tree are related with the six criteria that we have retained for observations, thus forming an arbitrary ontology. Moreover, we defined a static order over the criteria which is, from the root: `Who`, `Why`, `What`, `Action`, `When`, `Value`. The static structure of the decision tree makes the implementation of the intensionalizer easier but its major drawback is that the obtained tree is generally not minimal. In fact, techniques of tree balancing, like in ID5 [21], should be integrated but it issues in a dynamic structure for chronicles, much more complex to undertake.

3.3.4 Property 3: it is incremental.

Each time a new observation $\Theta(\sigma)$ is made, it is aggregated with the intensional output chronicle Kout of the agent performing the act (resp. with the Kin of the agent undergoing the act) so we have $Intensionalizer(Kout_i, \Theta(\sigma)) \rightarrow Kout_{i+1}$. The main principle is to check for *regularities* between $\Theta(\sigma)$ and Kout (resp. Kin): if there is some, the observation is absorbed by the chronicle (at small memory cost, e.g. the incrementing of an integer in the `When` criterion iterator), on the contrary, the observation is *differentially* aggregated to the chronicle (at full memory cost) by creating a new branch in the decision tree from the level of the current criterion.

3.3.5 Property 4: it handles time series[8].

The logical time θ being discrete, we have to deal with integer series. Our main principle is to detect linear segments of time so as to transform a series into a discrete *iterator* (e.g. `{0,1,2,3,4}` $\Rightarrow$ `Range[0,4]` and `{0,2,4,6,8}` $\Rightarrow$ `Range[0,8,2]`). When a "break" appears in a series, a new iterator is started (e.g. `{0,1,2,5,8,11,14}` $\Rightarrow$ `{Range[0,2], Range[5,14,3]}`).

Consider the example `W1`, given in table 1. It shows a very regular interaction flow. Consequently, the intentional chronicle Kin_A of agent A is a very simple tree which graphic representation is given in figure 1

[7] In Table 1, we showed CHIN and CHOUT chronicles for the reader to compare them with Kin and Kout, but in the implementation of MAGENTA, they do *not* exist; we build directly Kin and Kout from observations.

[8] The same algorithm is provided for the `Value` criterion.

```
W2 = AgentWorld[
  Agent[A, Define[x=0],
           DefineProcess[p, Script[x=x+1]],
           DefineProcess[q, Script[C[a]=x]]],
  Agent[B, Define[x=0],
           DefineProcess[p, Script[x=x-1]],
           DefineProcess[q, Script[C[b]=x]]],
  Agent[C, Define[x=0, a=0, b=0],
           DefineProcess[p, Script[x=x+a+b]]]
  ];
Exec{W2,40]

AGENT A:
   Kin [A[Always[Tell[x=x+1[Time[{3,1},{11,2},{18,3},{24,4},{26,5},{27,6},{28,7},{35,8}]]]]]
   Kout[C[Always[Tell[C[a]=x[Time[{1,0},{9,1},{10,1},{17,2},{21,3},{25,4},{33,7},{36,8}]]]]]

AGENT B:
   Kin [B[Always[Tell[x=x+1[Time[{2,-1},{7,-2},{12,-3},{19,-4},{20,-5},{31,-6},{32,-7},{37,-8},{39,-9}]]]]]
   Kout[C[Always[Tell[C[b]=x[Time[{4,-1},{8,-2},{15,-3},{16,-3},{23,-5},{29,-5},{38,-8}]]]]]

AGENT C:
   Kin [A[Always[Tell[C[a]=x[Time[{1,0},{9,1},{10,1},{17,2},{21,3},{25,4},{33,7},{36,8}]]]],
        B[Always[Tell[C[b]=x[Time[{4,-1},{8,-2},{15,-3},{16,-3},{23,-5},{29,-5},{38,-8}]]]],
        C[Always[Tell[x=x+a+b[Time[{5,2},{6,2},{13,2},{14,4},{22,7},{30,7},{34,7},{40,7}]]]]]
   Kout[C[Always[Tell[x=x+a+b[Time[{5,-1},{6,-2},{13,-3},{14,-4},{22,-4},{30,-5},{34,-3},{40,-3}]]]]]
```

Table 2: The intensional chronicles of agents A, B and C of AgentWorld W2, after 40 steps of execution. A and B exhibit a very regular input/output behavior (on themselves and on C) while C has a far more complicated `Kin`. Note that, due to the interleaving between the activation of the five processes, the time range is no longer intensionalizable; in that case, `Time` is a list of couples {`When`, `Value`}.

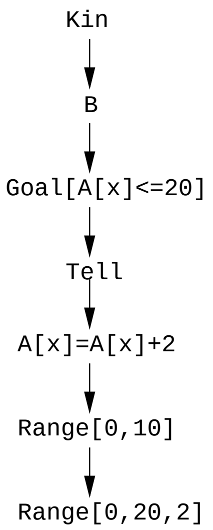

FIGURE 1: Intensional input chronicle Kin_A of agent A in W1.

Now, consider the following AgentWorld `W2`, defined in table 2 where two agents A and B increment their internal counter x and send the current value to another Agent C which, in turn, makes a final summing; all guards are equal to `Always`. The state of the intensional chronicles of agents in `W2`, after 40 execution steps, is given in table 2. The graphic representation of the Kin of agent C, is given in figure 2

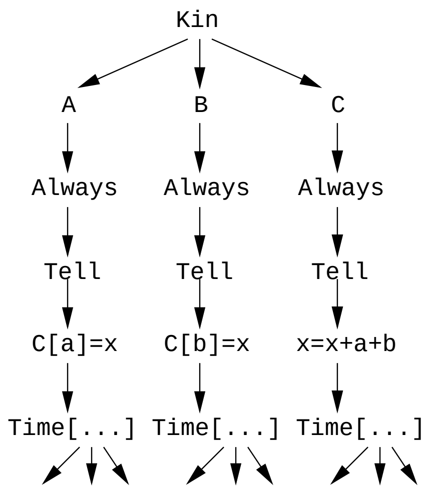

FIGURE 2: Intensional input chronicle Kin_C of agent C in W2.

Kin_C is a relatively complex decision tree, reflecting the fact that agent C has been undergoing three different interaction flows during the runtime.

4. REASONING OVER CHRONICLES

4.1 Causal Reasoning

Building intensional chronicles is a first attempt to provide reasoning tools about interaction flows between agents. Indeed, agent A, provided with the chronicle displayed in figure 1, is *aware* of (*i.e.* has a representation of) what is currently happening. It can produce the following explanation: "there is another agent in the world, called B, with the intent that my attribute x reaches the value 20, so that B updates regularly my x by increments of 2". Likewise, agent C, provided with the chronicle displayed in figure 2, is aware that its three internal attributes a, b and x are updated (always, *i.e.* with no particular "intent") by two other agents and by himself (ability of introspection).

However our main objective was to go further than passive explanatory reasoning. We wanted to offer proactive reasoning capabilities to the agents. To achieve this purpose, two internal heuristics `CauseAct` and `CauseWho` (dealing with causal reasoning over intensional chronicles), and an external heuristic act `Satisfy` were developed.

4.2 Causal Heuristics

The causal heuristics `CauseAct` and `CauseWho` are internal operators that can be used within a script expression.

4.2.0.1 *Looking for the cause of a guard to become active.*

`CauseAct`$[\gamma]$ elaborates the pattern structure of an act which, being applied *regularly* and *incrementally* upon the internals of the current agent, could have been a possible cause for the guard γ to become *active* (*e.g.* evaluates to `False` for a `Goal`). When several acts can apply for the cause, a list of patterns is returned. The pattern structure returned by `CauseAct`, referred to below as *actpat*, is in a form suitable to be applied to an input intensional chronicle (a Kin). The implementation of the `CauseAct` operator is based on a rule system. An example of some rules, dealing with satisfiable guards of type `When`, are given in table 3.

4.2.0.2 *Looking for an agent to perform an act.*

`CauseWho`$[actpat \mid \{actpat,^{+}\}]$ takes as an argument a pattern of act structure, as issued by `CauseAct` (it can be a single *actpat* or a list of *actpat*) and looks into the Kin of the current agent for all the agents that performed an act compatible with this structure. `CauseWho` returns a list of couples in the form $\{\{X_1,R_1\},\{X_2,R_2\},\ldots\{X_n,R_n\}\}$; $n \geq 0$ where X_i is an agent which performed the act with reason R_i.

4.3 The Act of Satisfaction

Interactional operators `Ask`, `Tell` and `Achieve` operate upon the definitions of agents. We wanted the operator `Satisfy` to operate upon the processes of agents; as a matter of fact, it operates upon the *cause* of processes, indirectly controlling the activability of processes. Let a be an agent performing a `Satisfy` act on an agent of symbolic name b:

`Satisfy`$[b,\ \gamma]$ performs an incremental change[9] upon the internals of b (mainly by updating values of attributes) in order for the guard γ, evaluated in the context of b, to tend to become *inactive*. The underlying idea is: by iteratively applying a `Satisfy` act on an agent one can eventually prevent some of its guarded processes to be active.

Like `CauseAct`, `Satisfy` is based on a rule system, with the same domain restriction: in the current implementation, the syntax of satisfiable guards is

$$\texttt{Goal} \mid \texttt{When}\ [x \mid a[s]\ (< \mid \leq \mid == \mid \geq \mid > \mid \neq)\ int\,expr]$$

where a is an agent name (it can be omitted), s is the symbol of an attribute and the expression $int\,expr$ must evaluate to an integer. The rule system of `Satisfy` returns a λ-expression to be applied to the attribute s specified in γ. For example, if an agent A has the definition `i=5` and some other agent B performs the act `Satisfy[A, Goal[A[i]>10]` then $\lambda x \,.\, x \leftarrow x+1$ will be applied to the attribute `i` of agent A.

4.4 Putting Things Together

Consider the AgentWorld `W3` in table 4 where we defined an agent A whose process p (activable if `x<5`) increments the attribute y of another agent B; thus the causal relation holds:

$$x{<}5 \longmapsto y{+}{+}.$$

The agent B has a definition `g=When[y≥5]`[10] which is the representation that B watches on a certain state of its attribute y via its process q; thus the causal relation holds:

$$y{\geq}5 \longmapsto \text{"Do something about it!".}$$

Doing something about y overstepping the value `5` is accomplished in the script of process q by piping the three causal operators: `CauseAct - CauseWho - Satisfy`, in order to

1. determine an operation O which is likely to produce the effect on y,
2. look in the Kin of B for an agent X who performed this particular operation O, and look for the guard G attached to it,
3. try to satisfy the guard G in the context of the agent X with a λ-expression F so that the process attached to G would be eventually deactivated.

In this example, we will obtain for the different informations: O = `Tell[_[y]=_[y]+1]`; X = A; G = `When[x<5]`; F = $\lambda u \,.\, u \leftarrow u+1$ (to be applied to y).

Results of a particular execution of `W3` are displayed in Table 4. Note that, instead of registering into the chronicle the guard g of process q as the cause of the act `Satisfy`, G was given with the keyword `ToSatisfy[When[x<5]]`. Thus, we loose some information (the actual cause of q) but we gain other information on what is going on: the fact that the attribute x of A is the subject of the operation F in order to disable the process associated with G.

5. CONCLUSIONS AND FUTURE WORK

The MAGENTA test-bed was implemented with S. WOLFRAM Mathematica 3.01 and it has showed interesting causal effects on AgentWorlds that could not be detailed

[9] The change has to be incremental because our model is dedicated to the study of interactional flows were internal state evolution is the result of *iterated* processes.

[10] As mentioned before, scripts and guards can be handled like any other knowledge information within definitions.

Guard pattern ⟹	Pattern of act structure
_[x_Symbol] === y_Integer	Tell[_[x]=_[x]+1] | Tell[_[x]=_[x]-1]
_[x_Symbol] =!= y_Integer	Tell[_[x]=_[x]+1] | Tell[_[x]=_[x]-1]
_[x_Symbol] < y_Integer	Tell[_[x]=_[x]-1]
_[x_Symbol] <= y_Integer	Tell[_[x]=_[x]-1]
_[x_Symbol] > y_Integer	Tell[_[x]=_[x]+1]
etc.	...

Table 3: Part of the CauseAct rule system, given in the MATHEMATICA syntax.

```
W3 = AgentWorld[
  Agent[A, Define[x=1],
           DefineProcess[p, When[x<5], Script[B[y]=B[y]+1]]],
  Agent[B, Define[y=1, g=When[y≥5]],
           DefineProcess[q, g, Script[Satisfy[CauseWho[CauseAct[g]]]]]]
  ];
Exec[W3,15]

AGENT A:

Kin[B[ToSatisfy[When[x<5]][Satisfy[#=#+1&[Time[{6,2},{8,3},{9,4},{11,5},{12,5},{13,6},{14,7},{15,8}]]]]
Kout[B[When[x<5][Tell[B[y]=B[y]+1[Time[{1,2},{2,3},{3,4},{4,5},{5,6},{7,7},{10,8}]]]]]

AGENT B:

Kin [A[When[x<5][Tell[B[y]=B[y]+1[Time[{1,2},{2,3},{3,4},{4,5},{5,6},{7,7},{10,8}]]]]]
Kout[A[ToSatisfy[When[x<5]][Satisfy[#=#+1&[Time[{6,2},{8,3},{9,4},{11,5},{12,5},{13,6},{14,7},{15,8}]]]]]
```

Table 4: The intensional chronicles of agents A and B of AgentWorld W3 after 15 steps of execution. The history is decomposed in three time segments: during period 1 to 4 only A is active (doing increments on y); during period 5 to 11 A and B are simultaneously active and interleaved (A continues to increment y while B increments x); during period 12 to 15 only B is active because, as soon as x reaches 5, A is disabled. Note that during period 5 to 11 A is not instantaneously disabled. Note also that after the 12^{th} step B continues infinitely to increment x even if it achieved its aim because its guard g remains active; we need to reason about the second order (not on the current value of an attribute but on its derivative) to solve this problem.

here (see [22]). However, this work is a first attempt in the domain of communication flows intensionalization and behavior modeling. Within the paper, we made several restrictions that prevent this model from being practical in work-flow analysis or in computer-mediated collectives for example. In future work we want to improve the model on three major points:

1) it is essential for the intensionalization algorithm to become dynamic: one can see in the Kin of agent A in Table 1 that the criterion `When` is not regular (thus making the intensionalizer to fail) but the criterion `Value` is! The capacity to re-order criteria so as to minimize the decision tree is well known since ID5 [21]. It should be applied here, not only for memory-saving purposes but also because the more regular a criterion is, the more pertinent the intended behavior can be.

2) we must improve the rule systems of `CauseAct` and `Satisfy` to deal with more complex expressions than mere arithmetics.

3) the class of acts should be extended to handle not only physical acts but also dialogical acts (e.g. BEG, briefly discussed in section 3.1).

6. REFERENCES

[1] J. FERBER, Reactive distributed artificial intelligence : principles and applications, In Foundations of DAI, Wiley&Sons, 1995.

[2] Y. SHOHAM, Agent Oriented-programming, Artificial Intelligence vol 60-1, 1993

[3] N.R. JENNINGS, M.J. WOOLDRIDGE (Eds), Agent technology, Springer, 1998.

[4] J. DORAN, From computer simulation to artificial societies. Transactions on SCS, 14-2, 1997

[5] R. CONTE, C. CASTELFRANCHI, Cognitive and social action, UCL Press London, 1995.

[6] T. SCHAEL, (Ed), Workflow management systems for process organisations, LNCS 1096, Edition 2, 1999

[7] P-M. RICORDEL, S. PESTY, Y. DEMAZEAU, About conversations between multiple agents, Int. Workshop CEEMAS '99, St. Petersbourg, 1999.

[8] A. AIKEN, Joseph M. HELLERSTEIN, J. WIDOM, Static analysis techniques for predicting the behavior of active database rules, ACM Trans. Database Systems Vol 20, 1:3-41, 1995

[9] H.J. LEVESQUE, R. REITER, Y. LESPERANCE, F. LIN, R.B. SCHERL Golog: a logic programming language for dynamic domains. journal of Logic Programming, Special issues on actions, 32:59-84, 1997

[10] D.V. Mc DERMOTT, Temporal logic for reasoning about processes and plans. Cognitive Science, 6:101-155, 1982

[11] P. CARLE, Ph. BENHAMOU, M. ORNATO, F-X. DOLBEAU, La reconnaissance d'intentions comme dynamique des organisations, 6th JFIADSMA, Pont-à-Mousson, France, 1998.

[12] C. DOUSSON, P. GABORIT, M. GHALLAB, Situation recognition: representation and algorithms. In Proc. 13th IJCAI 166-172, 1993

[13] C.A.R. HOARE, Communicating Sequential Processes, Prentice Hall 1986.

[14] E.A ASHCROFT, W.W. WADGE, *Lucid* - A formal system for writing and proving programs. SIAM Journal. Comp. 5, 3, 519-526. sept 1976.

[15] J. SHRAGER, P. LANGLEY, Ed. Computational models of scientific discovery and theory formation, Morgan-Kaufmann, 1990

[16] M. MOULET, Comparison of three inductive numerical law discovery systems. Machine Learning and Statistics, R. Taylor and R. Nakhaezadeh (Eds), Wiley, 1996.

[17] J.R. SEARLE, *Speech Acts*: an essai in the philosophy of language, Cambridge University Press 1969

[18] L. STEELS, F. KAPLAN, Linguistic evolution through language acquisition: formal and computational models, Chapter Bootstrapping grounded word semantics, Cambridge University Press, 1999

[19] N. SABOURET, J-P. SANSONNET, Un modèle de raisonnement sur les systèmes évolutifs fondé sur l'intensionnalisation de chroniques, RFIA, Paris, feb. 2000.

[20] J. R. QUILAN, Induction of decision trees. Machine learning, 1:81-106, 1986

[21] P. UTGOFF, ID5: An incremental ID3. Proc 5th Int. Workshop on Machine Learning, p107-120, J. Laird Ed., Morgan Kaufmann, 1988

[22] J-P. SANSONNET, *Magenta*: a Multiagent toolkit, Technical Report #99-11, LIMSI-CNRS, Orsay France, 1999

[23] D. JENSEN, M. ATIGHETCHI, R. VINCENT, V. LESSER, Learning quantitative knowledge for multiagent coordination, AAAI 99, pp 24-31, Orlando Augut 1999.

[24] M. LITCH, P. R. COHEN, On explaining behavior, European conference on philosophical psycology, 1999

[25] M. BARBUCEANU, M. S. FOX, COOL: A language for describing coordination un multi agent systems. Proceedings of ICMAS 95 p 17-25

[26] T. OATES, Identifying distinctive subsequences in multivariate time series by clustering.