Automated Answers to Questions about a Running Process

N. Sabouret J.P. Sansonnet

LIMSI-CNRS, Orsay, France
email: {nico,jps}@limsi.fr

Abstract

We propose here a formal model and a classification of the questions that a human user can ask about the actions, behaviour and activity of an active component. The components are described using a specific language and have access *at runtime* to a formal specification of their actions and current states. We present a formal model for requests[1] about actions based on the combination of speech acts and *procedural types* that determines the answering mechanism. We also study some procedural problems raised by the formalisation of such common sense questions. Eventually, we show some examples of questions represented in our request model.

1 Introduction

1.1 Context of the study

In this paper, we tackle a problem that is emerging in the domain of human-computer dialogical interaction. Since computer software services are more and more used by non specialist end-users, they ought to be able to explain "*what they are doing*".

Within the *InterViews* [19] project, we try to develop and to implement such active software components, capable of representing their own actions, reasoning about them and interacting in natural language with users, in order to answer a large range of questions about their behaviour.

One of our aims is to provide these components with reasoning capabilities as a support to explanation mechanisms. The work done within the fields of diagnosis [22, 14], planning [5, 2] and more generally "reasoning about actions" [13, 16, 10] constitutes the foundations of this approach, since it provides the tools required when producing an explanation [21] for the user.

We will take here an interest in defining a model for procedural requests, *i.e.* requests about the components dynamical structure and execution. Our aim is to propose a formal model for the user interaction, in order to link natural language processing tools with reasoning about actions formalisms. To this end, we shall use a specific component representation model that allows us to know about its dynamic and static structure at runtime.

[1]We will call *questions* the natural language interactions and *requests* the formal ones, should they come from a human question or from an other active components over the network.

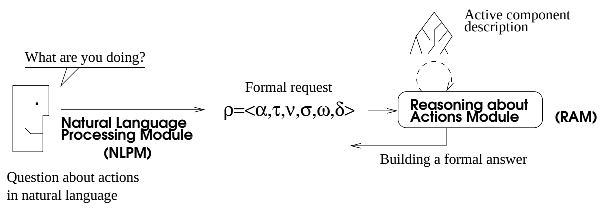


Figure 1: Processing questions about actions within the *InterViews* project.

We will concentrate on *questions* about actions, thus ignoring temporarily command processing. The study of procedural commands, however, led us to solve similar problems to those presented here and we will show that our model can be easily extended to such commands.

Our object is to have a formal request structure such that:

- a Natural Language Processing Module (NLPM) could translate a wide class of questions about actions into a request,
- a Reasoning about Actions Module (RAM) can interpret it to build an answer following the same scheme that'll be used in turn to produce a natural language answer.

  The production of this formal answer makes use of reasoning models about causality [11, 15].

Figure 1 illustrates this process.

1.2 Considerations about the formal requests model

In order to build our request model, we first classified the questions that a human could ask about an active component's actions and behaviour, using a "Wizard of Oz" approach. We then separated the natural language processing problems from the procedural ones.

In this paper, we use a complementary approach: given a component actions' description formalism [17], we propose a request model that takes into account the behaviour study along with the possibility to integrate the different kinds of questions considered. We classify the possible requests according to the answering mechanisms they involve.

We first present the representation formalism used for the components described here. We then propose a model for requests about actions and we give the general framework for building a formal answer. We then study some of the problems that arise, and we propose some clues. Eventually, we show that this model fits well for modelling the class of questions about actions we brought out.

2 Presentation of VDL

2.1 Syntax

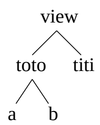
In the *InterViews* project, components are described in a specific language called VDL for ***V****iew* ***D****esign* ***L****anguage*. The software component's description in VDL is a tree whose nodes are characters strings called *concepts*. We call this concept tree a *view* like in Forbus QPT [7]. A view is the formal representation in VDL of the data structure and actions corresponding to the operating world defined by the component.

A component's runtime is based on rewriting the view at each *execution step*, like in Gurevich's Evolving Algebras [9]: we defined a ϕ_{exec} function that looks in the v_t view for specific concepts that are interpreted to build the v_{t+1} view. The components thus are modeless: they can *run* without any user interaction.[2]

2.2 Definitions

In this paper, we will call any view's subtree a *term* and for a given term, we will call *attributes* its direct subterms.

The views will be represented using a textual form: a term whose concept is c and whose attributes are $st_1, ..., st_n$ will be written $c[st_1, ..., st_n]$. For instance, $view[toto[a, b], titi]$ corresponds to the view :



The concepts containing special characters (brackets, commas, spaces) will be written between quotes (e.g. *"greater than"[x,y]*). The set of all concepts is denoted A^* and Υ is the set of all VDL terms.

[2]We totally defined this language's operational semantics in [17] and we implemented it in Java 1.1.7. Demos are available on our web page: http://www.limsi.fr/Individu/nico/exemples

2.3 Semantics

We present here the *procedural concepts* used by ϕ_{exec} to define the operational semantics of the views.

2.3.1 Basic actions

We showed in [17] that there exists only three elementary actions to modify a view. Each action is described by a specific concept used by ϕ_{exec}: **add** to add attributes to a term, **put** to replace all the attributes of a term by others and **del** to remove a term from the view.

These actions can use the **path** and **get** concepts to refer other subterms of the view. For instance, if v is the following view:

```
v = view[ table,
          add[path[table],pencil] ]
```

then $\phi_{exec}(v)$ is the view:

```
ϕexec(v) = view[ table[pencil],
                 add[path[table],pencil] ]
```

The *path* and *get* concepts affect multiple occurences of their values elsewhere in the view. For instance, the view:

```
view[ a[b], a[c], put[path[a],d] ]
```

is transorfmed by ϕ_{exec} into:

```
view[ a[d], a[d], put[path[a],d] ]
```

2.3.2 Arithmetics

We also defined some concepts corresponding to the most common arithmetic operators:

- **plus**, **times**, **minus** (opposite) and **inverse**,
- **equals** and **"greater than"** for relations,
- **and**, **or**, **not**, **true** and **false** for booleans operators.

2.3.3 Processes and events

To represent a component's interaction with a human or with other components, we proposed a mechanism based on *external events*. External events are VDL terms that one can send to the component during runtime.

The **event** concept allows the programmer to tell the component which events can be triggered and how they must be processed. For instance, let's consider the following view:

```
view[value[0], "add one"[ event[plus],
put[path[value],plus[get[value],1] ]]
```

The *value* term's value is incremented by the "*add one*" action each time the component gets the *plus* event.

Components can also act without any external interaction: guards of processes are described in VDL using the **guard** concept. For instance, in the view:

```
view[value[0], "add one"[ guard[
"greater than"[5,get[value]]], a[...] ]
```

The action a in "*add one*" is performed at each execution step as long as *value* is lesser than 5.

Combining these two concepts (*guard* and *event*) makes it possible to write guarded reactions to external event. Many examples are given in [17].

2.4 Definitions

2.4.1 Actions

Every term t such that $concept(t) \in \{add, put, del\}$ is a *basic action*.
Every term $t = c[st_1, ..., st_n]$ such that $\exists i \in [1, n] \mid concept(st_i) \in \{guard, event, add, put, del\}$ is an *action*.

2.4.2 Preconditions

The *preconditions* of an action a are given by the set Π_a of all terms with concept *guard* or *event* within its siblings, uncles, great-uncles and the siblings of all ancestors:

$$\Pi_a = \{t \in \pi(a) \mid concept(t) \in \{guard, event\}\}$$

where $\pi(a)$ is defined recursively by:

$$\pi(a) = \{t \mid parent(a) = parent(t)\} \cup \pi(parent(a))$$

2.4.3 Process

Every action a such that $\forall t \in \Pi_a, concept(t) = guard$ (*i.e.* not triggered by any external event) is a *process*.

2.4.4 Behavior and activity

A component's *behavior* is defined by the set of all its actions and it's *activity* is defined by the set of its processes.

2.5 Example: *Coco*

In this document, we will often use a voluntarily rather simple example. It is a mere counter called *Coco*[3] whose speed can be modified and that can be stopped or restarted as required by the user using external events. Its description in VDL is given below, with procedural concepts written in *italics* for easier reading.

[3] You can interact with *Coco* on the web: http://www.limsi.fr/Individu/nico/exemples/coco.html

```
view[value[0], running[true], speed[1],
     count[ guard[get[running]],
            put[path[value],
                 plus[get[value],get[speed]] ],
     stop[ event[stop],
            put[path[running],false] ],
     start[ event[start],
             put[path[running],true] ],
     "speed up"[ event[faster],
          put[path[speed],plus[get[speed],1]] ],
     "slow down"[ event[slower],
          put[path[speed],plus[get[speed],-1]] ]
   ]
```

2.6 Natural Language Questions Processing

In this paper, we don't tackle problems related to natural language processing, but we propose a mechanism for linking the reasoning about actions module (RAM) to the natural language processing tools.

The natural language processing module (NLPM) developped by S. Gérard [8] in the *InterViews* project simply ignores the procedural concepts in the view. It uses the action concepts as a set of natural language patterns and tries to match them with the user's question, using a semantics classification such as *WordNet* [4]. It thus builds a formal request whose arguments are made of those concepts.

The idea we support here is that separating the question understanding from the component's behaviour analysis allows us to solve more efficiently each problem and especially to answer a wider class of questions about actions.

We then encounter the crucial *reference problem*, which consists of determining which subterm is the subject of the user's question.[4] Within the *InterViews* project, this problem is part of the NLPM development and we will suppose it is solved for the rest of this document.

Examples

Here are some examples of questions for *Coco* that can be represented and answered with the model presented here. We also give some answers the NLPM should be able to produce:

- What is your speed? *My speed is 3.*
- What do you do? *I'm couting.* or: *Nothing.*
- What do you count? *I count my value.*
- Are you stopped? *Yes I'm stopped: I'm not running.*

[4] For instance, when a user interacting with *Coco* asks for "the value of the speed", the subject of the question is the term with concept *speed*. This example shows how important the concepts are in the view, since they define the semantics with regard to question understanding.

- Why do you count? *Because I'm running.*
- How do you count? *I increase my value by my speed.*
- When will you value be 42? *My value will be 42 within 3 execution steps.*
- What where you doing 5 steps before I changed your value? *10 steps ago, I was counting.*
- Why did you stop counting? *Because you asked me to stop.*
- What can you do? *I can count.*

Other examples are given in section 4.

2.7 Registering the view's past into chronicles

In order to answer questions about the view's past execution, we proposed a model based on *chronicles* [18] to perform a dynamic analysis on the component's behavior.

For each event that occurs in the system, we build a representation of the *situation* in which it occurs (*i.e.* the state of the involved variable and some procedural elements of information about the action that occurred) and we store this situation. This makes up the chronicles on which further reasoning can be carried out. A chronicle is a sort of attribute-value table where each line corresponds to a situation and each column to a specific criterion characterising the situation.

These chronicles are then *intensionalised* to extract their regularities and we gave some tools for questioning the views using requests about the chronicles (see [18]). An originality of this work is to use in the same description a static representation of actions (like in situations calculus [13]) and a memorisation of past executions with chronicles [3], as McCarthy suggests in Elephant 2000 [12].

We are now working on improving this chronicle model in order to reduce the search complexity and to answer questions about actions related with time in a more efficient way.

3 A model of request about actions

In this section, we present a formal request model for representing and answering the class of questions about actions that a human user can ask an active component.

3.1 General framework

A request about actions is defined by a sextuple:

$$\rho = \langle \alpha, \tau, \nu, \sigma, \omega, \delta \rangle$$

The elements of the request correspond to six characteristic criteria used to take into account all the relevant informations contained in the user's question:

- $\alpha \in \{Ask, Assert, What, Why, How, When, Error, Unknown\}$ is the request's *act*.

  As we will see in section 3.3, the answers to these requests can also be requests, with the $Assert$, $Error$ or $Unknown$ act.
- $\tau \in \{is, do, can, order\}$ is the request's *procedural type*. It allows us to determine which kind of procedural element (action, process, ...) is involved in the request.

  Each couple (α, τ) corresponds to a specific kind of question and a specific processing that will be briefly presented in section 3.3.
- $\nu \in \{true, false\}$ is a boolean tag. We'll see in section 4 how it can be used.
- $\sigma \in \mathcal{P}(\Upsilon)$ is a reference to the question's *subject*, *i.e.* the view's subterm concerned by the question (it might simply be the whole view). The reference module guarantees that σ is correct.

  As explained in [17], a *reference* is a set of terms ($\mathcal{P}(\Upsilon)$) describing a partial path in the view. For example, in *Coco*, the reference $\{speed\}$ has for result the view's subterm whose concept is *speed*.
- $\omega \in \mathcal{P}(\Upsilon)$ is the question's *object*. It is a set of VDL terms corresponding to the question's parameters. They are not necessarily part of the view (*i.e.* view's subterms).
- $\delta \in \Upsilon$ is the *date* the question is about, given in VDL. We'll see in section 3.4 the different forms it can take.

The answer build by the MRF are also requests. We give many examples in section 4.

We will now study the different possible values for each criterion. Afterwards, we will deal with some specific problems raised by questions about actions.

3.2 The act: α

Each request act corresponds to a specific kind of request, as *speech acts* [20] define the interpretation of a dialog in Natural Language Processing. Request acts define the question's frame, while the type (τ) defines the kind of problem that has to be solved (from a procedural point of view). For questions about actions, we have the following acts :

- **What** to specify that the question's object is unknown. For instance, "*What are you doing?*".

  If $\alpha = What$, then $\omega = \emptyset$, except for two request models with very specific semantics that will be studied later on.

- **Ask** to specify that the question's object is supposed to be a possible answer to the same $What$ request. For instance, "*Is your value 42?*" or "*Do you count?*".

- **Why** for diagnosis questions, dealing with either states or actions. For instance, "*Why did you stop?*".

- **How** for questions about the view's present or future execution. These requests can give rise to a plan generation (in the "future" case. For instance: "*How can I stop you?*") or to an action body's study (in the "present" case).

- **When** for temporal questions, leading to research dates of states or actions in the component's past or future execution. For instance: "*When was your value 42?*". We will not study these requests in this paper.

- **Assert** to give a result. In the frame of *questions* about actions, this act will only get used in answers built by the RAM. The object ω can't be empty in an $Assert$ request.

- **Error** to specify that the given request is not syntactically correct, and **Unknown** to specify that none of the arguments proposed in the ω criterion could be matched with a view's subterm.

  These last two acts are only used within answers.

3.3 The type: τ

3.3.1 Semantics for τ

τ is the procedural type of the request. It is used to determine the kind of procedural elements (actions, processes, ...) the request is dealing with. For questions about actions, we have the following procedural types:

- $\tau = is$ for questions about states (or *situations*, using Reiter's terminology [16]). For instance: "*Why is your value 47?*" corresponds to $(\alpha, \tau) = (Why, is)$.

- $\tau = do$ for questions about the component's activity (the *processes*). For instance, "*How can I make you count?*" corresponds to $(\alpha, \tau) = (How, do)$.

- $\tau = order$ for questions about reactions to external events. For instance, "*What can I ask you to do?*" corresponds to $(\alpha, \tau) = (What, order)$.

- $\tau = can$ for questions about the actions the component can perform. For instance, "*Can you count?*" corresponds to $(\alpha, \tau) = (Ask, can)$.

Each couple (α, τ) corresponds to a specific kind of question and a specific processing for the RAM.

3.3.2 Answering requests

While building an answer to a request about actions, the RAM searchs the view for either values of variables or action (depending on τ), using ω as a filter.

We will now study the different kinds of request, as well as the answering mechanisms corresponding to each type τ for each request act α. We only present here the general framework, whereas specific problems will be presented in section 3.5.

In this section, we will write $\rho = \langle\alpha, \tau, \nu, \sigma, \omega, \delta\rangle$ for the user's request and $\overline{\rho} = \left\langle\overline{\alpha}, \overline{\tau}, \overline{\nu}, \overline{\sigma}, \overline{\omega}, \overline{\delta}\right\rangle$ for the answer, built by the RAM.

3.3.3 The *do* type

Requests with type *do* concern the view's processes activity. The ν criterion is then used to specify whether the user is looking for active processes (*i.e.* processes whose guard is true) or inactive processes. Answering those requests always consists of looking for processes in the view, filtering with the elements of ω (if any). The answer $\overline{\rho}$ thus depends on both the act and the result of this search:

- If $\alpha = What$ with $\omega = \emptyset$, the answer has the form $(Assert, do)$ with $\overline{\omega}$ being the set of active or inactive processes (depending on ν) found within the view's subterm referred by σ. $\overline{\omega} = \{nothing\}$ when no process could be found.

  We will study the case $\alpha = What$ and $\omega \neq \emptyset$ in section 3.5.

- If $\alpha = Ask$, we compute the set of active (inactive if $\nu = false$) processes found in the view, filtered by the elements of ω. If no process is found, we try again with the set of processes in the opposite state and, in case of failure again, with all actions, including reactions to external events. If no such action could be found, the answer has then the form $(Unknown, is)$ which means that all the objects given in ω are unknown in the term refered by σ.

  Otherwise, the answer has the form $(Assert, do)$, meaning that the processes found $(\overline{\omega})$ are in the state $\overline{\nu}$, with other informations in case of reactions to events.

- If $\alpha = Why$ with $\omega \neq \emptyset$, we also build the set of processes found among those proposed in ω. We then build the set Π of their true or false preconditions, depending on ν.[5] The answer has the form $(Assert, is)$ with $\overline{\omega} = \Pi$.

- If $\alpha = How$, let ω' be the set of processes found among those proposed in ω. We then build an *intensional* representation of these processes' bodies[6]. Let ω'_ξ be the set of intensional representation of ω' elements: $\omega'_\xi = \{\xi(t), t \in \omega'\}$. The answer has then the form $(Assert, do)$ with $\overline{\omega} = \omega'_\xi$.

3.3.4 The *is* type

Requests with type *is* concern the view's subterms "*values*" and more generally the component's current state. Answering these requests always consists of looking in the view for the value of the term referred by σ. The ω criterion is mostly unused.

The answer thus depends on the act α:

- If $\alpha \in \{What, Ask\}$, the answer has the form $(Assert, is)$ with $\overline{\omega}$ the set of values of the subterm referred by σ. Generally, these *values* simply are the subterm's attributes. The ν and ω criteria are not used.

- If $\alpha = Why$, we build the set of actions whose *static consequences*[7] contain elements of ω. It is thus a subset of the view's actions. We then compute the set Π of their true or false preconditions, depending on the ν parameter. The answer has the form $(Assert, is)$, with $\overline{\omega} = \Pi$, asserting that these preconditions are true (or false) and that this is the reason why the current state is reached.

- If $\alpha = How$, we build a plan leading to a state in which σ's value is one of those proposed or ω (or, on the contrary, is *no more* an element of ω, when $\nu = false$). This plan is represented by a set π of external events to be received at specific dates, and the answer is thus $\overline{\rho} = \langle Assert, do, true, \{view\}, \pi, \delta\rangle$.

  Our plan generation uses a STRIPS-like method [5], based on regression planning, where a state corresponds to a VDL view and where actions are described using the VDL formalism, as a set of *add*, *put* and *del*.

Remark about diagnosis

As a general rule, when $\alpha = Why$, the answer gives the different values and actions explaining the state or actions in the request object (depending on τ).

Classical diagnosis tools build a set of litteral as the cause of a given state. On the contrary, *explanatory diagnosis* [14] builds a sequence of actions corresponding to "what happened".

The mechanisms being developped in the RAM, based on static consequences collection and causal reasonning about actions, tends to unify these two approaches. Our aim is to be able to use requests to express both structural or procedural causes of either actions ($\tau = do$) or states ($\tau = is$).

We will develop this problem in section 3.5.4.

3.3.5 The *can* type

Requests with type *can* concern the component's possible actions. Answering such requests always consists of looking into the view for actions in a specific state, possibly filtering with ω.

However, questions with type *can* are ambiguous since "to be able to perform an action" may have two different meanings on a procedural point of view:

- to have the physical ability to: the action exists in the view as a concept linked to the question by a *WordNet* relation,
- to be authorised to: all the action's preconditions are true.

What makes our model quite powerful is that the RAM can solve this ambiguity: looking for an action in the view leads to know which actions exist, whereas checking their preconditions tells us not only what are the possible actions, but also what is the *question frame* (ability or authorisation).

We then have to build an answer that depends on both α and the search result:

- If $\alpha = What$, the answer has the form $(Assert, can)$ with $\overline{\omega}$ the set of actions (processes and reactions to events) whose preconditions are satisfied or not (depending on ν), without taking external events into account[8]. If no action can be found, we try again with $\neg\nu$ into account. If actually no action is found, $\overline{\omega} = \{nothing\}$.

- If $\alpha = Ask$, we have to solve the "can" ambiguity. We first build the set ω' of found actions, filtering with ω, should they have external events in their preconditions or not. We then build Π_{true}, the set of their preconditions evaluated to true and Π_{false}, the set of false preconditions.

[5] The method used to build Π would take too long to be presented here, since it involves many parameters of the process activity.

[6] The intensionalisation process is briefly presented in section 3.5. It follows our work done in the field of chronicle intensionalisation [18].

[7] We call *static consequences* the action's result that don't depend on the current state. They can easily be tracked down with a simple algorithm that looks for basic actions (*add*, *put* and *del*) and computes their results.

[8] We are currently studying an extension of this model based on chronicle memorisation that could determine whether external events should be considered in *can* requests.

If $\Pi_{false} = \emptyset$, the request concerns a physical possibility: the answer has the form $(Assert, can)$, with $\overline{\nu} = true$ and $\overline{\omega} = \omega'$.

If $\Pi_{false} \neq \emptyset$, some actions exist in the view and are not performed[9].

In this case, when $\Pi_{true} \neq \emptyset$, the request concerns an authorisation and the answer is "positive" for some of the proposed actions: these actions exist and they can be performed (or they are being performed in case of processes). We then produce an answer of the form $(Assert, can)$ with $\overline{\omega}$ the set of actions with true preconditions. This answer gives the list of the *possible* actions.

Otherwise ($\Pi_{true} = \emptyset$), the request concerns an authorisation and the answer is "negative": some proposed actions exist but their guards forbid their execution. The answer must not only contain an assertion of this impossibility, but also an *explanation*, *i.e.* a set invalid preconditions.

The answer is then a multiple request, *i.e.* a set of requests $\overline{\rho_i}$ such that:

$$\begin{aligned} \overline{\rho_1} &= \langle Assert, can, \perp, \sigma, \omega, \delta \rangle \\ \overline{\rho_{i\geq 2}} &= \langle Assert, is, \perp, \overline{\sigma_i}, \overline{\omega_i}, \delta \rangle \end{aligned}$$

Each $\overline{\rho_{i\geq 2}}$ asserts the invalidity of one guard ($\overline{\sigma_i}$) in $\Pi_\perp$.

- If $\alpha = Why$, let ω' be the set of the view's processes (among those proposed in ω). Let Π_{true} be the set of their valid (invalid if $\nu = false$) preconditions. The answer has then the form $(Assert, is)$ with $\omega = \Pi_{true}$.

- If $\alpha = How$, we build a plan leading to a state where the set Π_{false} of false preconditions is empty ($\Pi_{false}(\omega) = \emptyset$). This plan is represented by a set π of external events with dates. The answer thus has the form $(Asssert, do)$ with $\omega = \pi$.

3.3.6 The *order* type

Requests with type *order* concern the view's interaction, *i.e.* external events. As far as questions about actions are concerned, there exists only two kinds of requests with $\tau = order$:

- When $\alpha = What$, the aim is to find the possible external events in the view, using a similar algorithm to the one seen for $\tau = do$.

- When $\alpha = Assert$, the request simply asserts that an external event has been sent (or has not been sent if $\nu = false$) and that this is an explanation for the user's question. The syntax thus is:

$$\overline{\rho} = \langle Assert, order, \nu, \sigma, \emptyset, \delta \rangle$$

with σ being the external event and δ the date it was sent at.

We will see some examples in section 4.

3.3.7 Remark

For all requests such that the act $\alpha \in \{Ask, Why, How, Assert\}$, we supposed that $\omega \neq \emptyset$. There exists one exception to this rule which is the case where $(\alpha, \tau) = (Why, is)$, corresponding to questions of the form "*What is the use of ...?*". The answer is then build using a global definition which has to be given *a priori* by the programmer.

All other cases where $\omega = \emptyset$ are syntax errors, producing an answer with the request act $\alpha = Error$.

3.4 The date

All questions about actions can concern a date in the present time, in the past or even in the future. We chose different forms to express dates in our request model:

- $\delta = \surd$ for present,
- $\delta = date[x]$ ($x \in \mathbf{Z}$) represents a date in the past, (if $x < 0$), the present ($x = 0$) or the future ($x > 0$), given in number of execution steps (see section 2.1),
- $\delta = date[x, \delta']$ ($x \in \mathbf{Z}$ and δ' a date) represents a date shifted by x steps from δ',
- $\delta = global[x]$ ($x \in \mathbf{N}$) represents a date in steps since the component's execution started,
- $\delta = changed[r]$ (r a reference to some view's subterm) represents the date when the term referred by r was last modified.
- $\delta = \Delta[is[get[r], x]]$ (r a reference, x a set of values and $\Delta \in \{past, future\}$) represents the first date when r's value was ($\Delta = past$) or will be ($\Delta = future$) an element of x.

The mechanism used to answer questions about past or future is based on the use of chronicles, as we explained in section 2.7, and won't be discussed furthermore.

[9] Note that if $\omega' = \emptyset$, the answer has the form $(Unknown, is)$, asserting that none of the proposed actions in ω is known to the view. It is a case of physical ability with a complete "*no*" answer ("*no such action*").
The *Unknown* answer is provided for any request about action ($\tau \in \{do, can\}$) such that $\omega' = \emptyset$.

3.5 Six main problems arising in solving questions about actions

In section 3.3, we only studied a general framework. We present here some of the most important problems that arise in answering questions about actions. We do not claim to have solved these problems. We rather try to show how our model takes them into account and how it could be used as a guideline to some solutions.

3.5.1 References in ω

We explained in section 2.6 that the NLPM should guarantee both the existence and the uniqueness of the request's subject σ.

However, another reference problem arises in ω for questions with type $\tau = do$ and $\tau = can$ since the elements of ω refer to possible actions in the view. Let $\Xi_{ref}(r, v_t)$ be the set of terms referred by the reference r in the view v_t. For all $r \in \omega$, if $|\Xi_{ref}(r, v_t)| \neq 1$, then there exists a specific reference problem whose solution depends on the request itself!

For instance, in *Coco*, the question "*Are you counting or sleeping?*" is translated into the request:

$$\langle Ask, do, true, \{view\}, \{sleep, count\}, \surd \rangle$$

but $\Xi_{ref}(\{sleep\}, v_t) = \emptyset$.

We implemented answering algorithms that partially take this difficulty into account.

3.5.2 Procedural reference

We would like the user to be able to refer to actions using not only their names, but also the "perceived behaviour". This is the reason why an element of ω can be a complex VDL term.

For instance, the user could ask *Coco* "*What is the action that increases the value*". The "action that increases the value" corresponds to the VDL description $put[path[value], plus[get[value], \#1]]$ and as a consequence, the object of the request should be:

$$\omega = \{put[path[value], plus[get[value], \#1]]\}$$

We implemented algorithms in the RAM in order to be able to look into the view for actions referred by their "*name-concept*" or by their VDL description, possibly with parameters.

3.5.3 Actions intensionalisation

We explained in section 2.6 that the NLPM won't use the procedural concepts. It will work with more general concepts, representing natural notions like "*increase x*" (represented in VDL by $increase[x, \#1]$). Within the *InterViews* project, we call *ground concepts* these macro-concepts which are close to natural language.

This is the reason why we need intensionalisation and extensionalisation algorithms such that:

$$\xi(put[path[\#1], plus[get[\#1], \#2]]) = increase[\#1, \#2]$$

However, we also need to have *ground relations* between VDL actions and procedural constraints. For instance, the question "*Why do you count backward?*" corresponds to the request:

$$\langle Why, do, true, \{view\}, \{backward[count]\}, \surd \rangle$$

The RAM needs to know that $backward[count] = true$ iff $speed < 0$. This relation is represented in the ground by:

$$backward[\, increase[x, y],\ greaterThan[0, y]\,]$$

The *nothing* concept seen in section 3.3.5 is also represented by a ground relation.

3.5.4 Ramification problems

The ramification problem [6] is the impossibility to describe with logical formulaes all the indirect consequences of an action. In VDL and in all other programming languages, the ramifications are automatically computed by the component actions.

However, the ramification problem arises also in questions about actions: human users often refer to actions by describing their consequences. The RAM has then to "trace back" the ramifications, *i.e.* to discover the actions responsible for the modifications described by the user and, because of the ramification problem, this can't be reduce to a single matching the right formulae operation. Moreover, we encounter two major difficulties:

- The first one is to determine what action is the actual *cause* [15], *i.e.* where to stop in the backtrack process.
- The second difficulty is that an action's consequences may be *another action.* For instance, answering the request $\langle Why, do, true, \{view\}, \{count\}, \surd \rangle$ ("*Why do you count?*") required to look for actions whose consequences validate the *count* action's guard.[10]

We are now studying algorithms based on logical approaches [1, 11] to "trace back" ramifications. Our aim is to use an explanatory diagnosis method such as [14] that could determine "*what happened*", in order to determine the *cause* of a *modification* that we have to give in the answer, be it a *state* or an *action*.

[10] The NLPM knows about the view's subterms nature (if they are actions, what are the modified terms, ...). This information is required to avoid requests with wrong type like $\rho = < why, is, true, \{view\}, \{stop\}, \surd >$ (*stop* is an *action* and thus can't be in the ω set of a request with type $\tau = is$).

3.5.5 Preconditions

We saw in section 3.3 that the sets of preconditions Π_{true} and Π_{false} can be used by the RAM to solve the ambiguity and to give the right answer. However, in the "authorisation" case, any negative answer should contain an explanation, *i.e.* a set of invalid preconditions.

For instance, to the question "*Can you count?*", corresponding to:

$$\langle Ask, can, true, \{view\}, \{count\}, \surd\rangle$$

we should answer:

- "*I can count.*" when the guard of the *count* action is true:

  $$\langle Assert, can, true, \{view\}, \{count\}, \surd\rangle$$

- "*I can't count: I'm not running.*" otherwise:

  $$\{ \quad \langle Assert, can, false, \{view\}, \{count\}, \surd\rangle \,; \\ \langle Assert, is, false, \{running\}, \{true\}, \surd\rangle \quad \}$$

The answer is thus a set of requests, the first one being the negative answer, other ones representing the invalid preconditions. Each $\overline{\rho}_{i\geq 2}$ concerns only one invalid precondition.

3.5.6 $\alpha = What$ and $\omega \neq \emptyset$

Requests with the act $\alpha = what$ generally have an empty object ω (*e.g.* $\langle What, do, true, \{view\}, \emptyset, \surd\rangle$ for "*What are you doing?*"). There exists however two exceptions each having very specific semantics:

- When $\tau = can$, the elements of ω are actions (generally represented by their concepts). The aim is then to find the arguments of these actions (the external events they use). For instance, suppose we have a view with a *take* action. The question "*What can you take?*" will be represented by:

  $$\langle What, can, true, \{view\}, \{take\}, \surd\rangle$$

  The answer is a set of $(Assert, can)$ requests $\overline{\rho_i}$ with each $\omega_i = take[x_i]$ where x_i is a possible argument of the *take* action (associated with the *take* external event).

- When $\tau = do$, the elements of ω are also actions and the aim is to find which subterm they modify. The anwser is a set of $(Assert, do)$ with $\omega_i = c[x_i]$ with c the concept of the action and x_i a reference to a term it modifies.

  For instance, in *Coco*, "*What do you count?*" will produce:

  $$\langle What, do, true, \{view\}, \{count\}, \surd\rangle$$

  and the answer is:

  $$\langle Assert, do, true, \{view\}, \{count[value]\}, \surd\rangle$$

4 Examples

In this section, we show how the user's questions in natural language can be represented in our request model and what answers the RAM should provide.

4.1 Questions about the static structure

The model presented here allows us to represent and to answer simple questions about the component's static structure, as a case of *is*-type requests[11]. We can then represent:

- "*What is your speed?*" by:

  $$\langle What, is, true, \{speed\}, \emptyset, \surd\rangle$$

  The answer is then:

  $$\langle Assert, is, true, \{speed\}, \{7\}, \surd\rangle$$

  that will produce "*My speed is 7.*".

However, this answer is very limited (no time unit, no link with the value increment) and we should study an improvement of our algorithms that could add the information: "*I increase my value of 7 at each execution step.*", *i.e.* formally:

$$\langle Assert, do, true, \{count\}, \{increase[value, 7]\}, \surd\rangle$$

4.2 Questions about the component's activity

The component's *activity* is defined by the set of its *processes*. We can have the following questions:

- "*What do you do?*":

  $$\langle What, do, true, \{view\}, \emptyset, \surd\rangle$$

  whose answer is "*I count.*", represented by:

  $$\langle Assert, do, true, \{view\}, \{count\}, \surd\rangle$$

- "*Do you count or sleep?*":

  $$\langle Ask, do, true, \{view\}, \{count, sleep\}, \surd\rangle$$

  whose answer is "*I count.*", represented by:

  $$\langle Assert, do, true, \{view\}, \{count\}, \surd\rangle$$

When $\nu = false$, we look for inactive processes.

[11] Within the *InterViews* project, we study a more complete modelisation of questions about the static structure as a specific problem. The requests with type *is* are very simple and can only be used to get straight informations about the terms values.

4.3 Explanation requests

Answering explanation requests can be very simple in the general case, but we showed how the ramification problem can be an obstacle:

- "*Why don't you count anymore?*":

$$\rho = \langle Why, do, false, \{view\}, \{count\}, \surd \rangle$$

whose answer is:

  - $\langle Assert, is, true, \{running\}, \{false\}, \surd \rangle$ *i.e.* "*Because I'm not running.*" if we ignore the ramifications,
  - $\langle Assert, order, true, \{stop\}, \emptyset, \surd \rangle$ *i.e.* "*Because you asked me to stop.*" if we trace ramifications back. We can see that it is a diagnosis-like problem focused on the actions described in the component (except that we are not dealing with failures).

In other cases, explaining an action can simply be done by studying the view. But building the answer may still be a problem when a ground concept appears:

- "*What do you do in order to count?*":

$$\langle How, do, true, \{view\}, \{count\}, \surd \rangle$$

whose answer is:

$$\left\langle \begin{array}{l} Assert, do, true, \{view\}, \\ \quad \{count[increase[value, speed]]\}, \surd \end{array} \right\rangle$$

i.e. "*I increase my value with my speed.*".

- "*Why do you count backwards?*":

$$\langle Why, do, true, \{view\}, \{backward[count]\}, \surd \rangle$$

whose answer is:

$$\langle Assert, is, true, \{speed\}, \{negative\}, \surd \rangle$$

i.e. "*Because my speed is negative.*".

Finally, using planning tools, we can help the user to reach an aim:

- "What *can I do for your value to be 10?*":

$$\langle How, do, true, \{value\}, \{10\}, \surd \rangle$$

The planner will provide an answer as a set of requests $\rho_i = \langle Assert, order, true, \sigma_i, \emptyset, \delta_i \rangle$ where σ_i is an external event to be sent and δ_i indicates when to send it.

4.4 Temporal questions

We can use the date to formalise questions about the view's past or future execution:

- "*What will your value be within 10 steps?*":

$$\langle What, is, true, \{value\}, \emptyset, date[10] \rangle$$

The hypothesis is that the user wont disturb the component's execution. We then can build the following answer:

$$\langle Assert, is, true, \{value\}, \{42\}, date[10] \rangle$$

"*My value will be 42 within 10 steps.*".

- "*What was you value 10 steps before I changed your speed?*":

$$\langle What, is, true, \{value\}, \emptyset, date[-10, changed[speed]] \rangle$$

To build the answer, we use the chronicle model briefly presented in section 2.7.

4.5 Compound requests

Many questions can't be represented by a single request. This is the reason why we introduced *requests sets* also called *compound requests*: $\rho = \{\rho_1; ...; \rho_n\}$. The RAM must then determine the semantic link between the different ρ_i :

- "*What don't you do that you could do?*":

$$\begin{array}{rcl} \rho & = & \{\langle What, do, false, \{view\}, \emptyset, \surd \rangle\,; \\ & & \langle What, can, true, \{view\}, \emptyset, \surd \rangle\} \end{array}$$

The type set is then provided with an order $\preceq_\tau$ that will be used to determine the answer's type (τ). The answer object $\overline{\omega}$ is then determined by computing the intersection of all $\overline{\omega_i}$, which produces the answer "*I could count*" in our example:

$$\overline{\rho} = \langle Assert, can, true, \{view\}, \{count\}, \surd \rangle$$

- "*Why did you restart counting whereas I didn't ask you to?*" will be represented by:

$$\begin{array}{rcl} \rho & = & \{\langle Why, do, true, \{view\}, \{count\}, \surd \rangle\,; \\ & & \langle Assert, order, false, \{start\}, \emptyset, \surd \rangle\} \end{array}$$

We are now working on algorithms based on chronicles that could produce the answer:

$$\overline{\rho} = \langle Assert, order, true, \{start\}, \emptyset, date[-2] \rangle$$

meaning "*You asked me to! Two steps ago!*".

5 Conclusion and current work

We proposed in this paper a model for requests that can be used to represent questions that a human user can ask about a component's behaviour. This request model is used within the *InterViews* project to link reasoning tools with NLPM.

We now study in details the algorithms used to answer these questions and we presented here the main problems that arise. The mechanisms we use, briefly presented in section 3.3, are general enough to answer question about actions for any component described in our formalism VDL [17, 19]. We want to add as soon as possible some solutions to the classical problems in the domain of reasoning about actions and change (ramification problem, plan generation, ...).

We showed informally that it was possible to represent a wide class of questions in natural language using our model. We partly implemented this work in the *InterViews* project and we are now trying to integrate the chronicle model and the command requests.

This work is a first proposal for processing questions about actions on a formal way. It is based on common sense reasoning mechanisms using a semantic analysis of active component's descriptions. However, this problem is very difficult to formalise whereas it appears to be more and more crucial in the field of human-computer interaction and, more generally, in computer interaction over the networks.

References

[1] M.A. CASTILHO, L.F. DEL CERRO, O. GASQUET, and A. HERZIG. Reasoning about actions in modal logic. In *Working Notes of the Symposium on Reasoning about Actions: Foundations and Applications - ESSLLI'98*, 1998.

[2] E. DAVIS. Knowledge Preconditions for Plans. *Journal of Logic and Computation*, 4(5):721–766, 1994.

[3] O. DESPOUYS, F. INGRAND, M. GHALLAB, and J.P. GOUYON. Extension d'IxTeT pour son application à la supervision d'un réseau électrique. In *Proc. RFIA'98*, pages 265–274, 1998.

[4] C. FELLBAUM, editor. *WordNet, An Electronic Lexical Database* . MIT-Press, 1998.

[5] R.E. FIKES, P.E. HART, and N.J. NILSSON. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3:251–288, 1972.

[6] J.J. FINGER. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford, 1987.

[7] K.D. FORBUS. Qualitative Process Theory. *Artificial Intelligence*, 24:85–167, 1984.

[8] S. GERARD and J.P. SANSONNET. A spatio-Temporal Model for the representation of Situations Described in Narrative Texts. In *Proc. NLP2000*, Patras, June 2000.

[9] Y. GUREVICH. Evolving Algebra Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[10] H.J. LEVESQUE, R. REITER, Y. LESPERANCE, F. LIN, and R.B. SCHERL. Golog: a logic programming language for dynamic domains. *Journal of Logic Programming, special issues on actions*, 31:59–84, 1997.

[11] V. LIFSCHITZ. Situation calculus and causal logic. In *Proc. KR'98*, pages 536–546, 1998.

[12] J. McCARTHY. Elephant 2000: A programming language based on speech acts. Technical report, Stanford University, 2000. draft version.

[13] J. McCARTHY and P.J. HAYES. Some philosophical problems from the standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[14] S. McILRAITH. Explanatory Diagnosis: Conjecturing Actions to Explain Observations. In *International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 167–177, 1998.

[15] J. PEARL. Reasoning With Cause and Effect. In *Proc. IJCAI'99*, 1999.

[16] R. REITER. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* Draft of book available at http://www.cs.toronto.edu/cogrobo/book.ps.Z, 1999.

[17] N. SABOURET. VDL Procédural : syntaxe et sémantique opérationnelle. Technical Report 2000-09, LIMSI-CNRS, 2000.

[18] N. SABOURET and J.P. SANSONNET. Chronicle intensionalization as a support for learning behavior over autonomous agents' runtime. In *Proc. JIOSC 2000*, Orsay, France, 2000.

[19] J.P. SANSONNET. Présentation de VDL 0.1. Technical Report 99-10, LIMSI-CNRS, 1999.

[20] J.R. SEARLE. *Speech Acts.* Cambridege University Press, 1969.

[21] M. SHANAHAN. Explanation in the Situation Calculus. In *Proc. IJCAI'93*, pages 160–165, 1993.

[22] M. THIELSCHER. A Theory of Dynamic Diagnosis. *Electronic Transactions on Artificial Intelligence (ETAI)*, 1997.