Querying knowledge about actions in the semantic web

N. Sabouret J.P. Sansonnet
LIMSI-CNRS, Orsay, France
email: {nico,jps}@limsi.fr

Abstract
In this paper, we address the problem of answering formal requests about actions for active components that can be found in online services on the Internet. 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 about actions, based on the combination of speech acts and procedural types, that allows us to represent the different kinds of questions that a human user can ask about the actions, behaviour and activity of an active component. We propose answering algorithms and we illustrate them on some examples.

1 Presentation of the problem
1.1 Active components in the semantic web
The semantic web is an extension of the current web in which information is described in a well-defined manner [3], in order to allow software components to perform automatic processing based on the page contents and the semantics of these contents [17] (extraction and presentation of the information, automatic services activation, etc.). The current trends propose to make use of knowledge representation techniques [2] to enhance web pages information with semantics [6]. The emergence of standards for structural knowledge representation, like DAML-OIL and RDF, allowed researchers to offer better services in information retrieval: semantic requests [30], semantic document indexing [5], etc.

In this context, online services on the Internet led to the notion of active components, i.e. software components characterised by actions and capable of providing services to the users [12, 14]. Active components cannot be reduced to simple software systems used to process or produce information in the web. They are part of this information which is no longer only structural but also functional. As a consequence, in the same manner structural knowledge can be queried, active components in the semantic web should be able to explain to users “what they are doing”.

Within the InterViews [27] project, we try to develop and to implement such active 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 and execution.

The work done within the field of “reasoning about actions” [18, 15, 22] constitutes the foundations of this approach, since it provides the tools required when producing an explanation [29] for the user. We take here an interest in defining a model for questions about the components dynamical structure and execution. Our aim is to propose a formal request model such that:
What are you doing?

To this end, we shall use a specific component representation model (see section 2.1) that allows us to know about its dynamic and static structure at runtime.

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: starting from a description formalism used to describe a component’s actions, 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. Eventually, we show that this model fits well with modelling the class of questions about actions we brought out.

2 Presentation of VDL

2.1 General principle

In the InterViews project, components are described in a specific language called VDL for View Design Language. 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 [10]. A view is the formal representation in VDL of the active component’s data structure and actions.

A component’s runtime is based on rewriting the view at each execution step, like in Gurevich’s Evolving Algebras [13]: we defined a $\phi_{exec}$ function that looks in the $v_t$ view for specific concepts

---

1 We will call questions the natural language interactions and requests the formal ones, should they come from a human question or from another active component over the network (e.g. in the web).
that are interpreted to build the $v_{t+1}$ view. The components thus are *modeless*: they can *run* without any user interaction.

Our first aim was to propose an extension to XML that could allow the programmer to attach actions to structural information, in order to describe and to run an active component as a web page. However, it appeared that DTDs could only be used to describe *validating static parsers, i.e.* function that link tags with static semantics. Our aim was to describe not only static parsers, but also one *dynamic transducer* ($\phi_{\text{exec}}$) that gives tags procedural semantics in order to build a new XML page. Therefore, we decided to base our work on a more classical model, that could help us to focus on the actions representation problem: the rewriting trees. This led us to propose a specific language, whose structural part is very simple (concepts are reduced to mere character strings), but that could easily be linked to markup languages like XML. We totally defined this language’s operational semantics in [23] and we implemented it in Java 1.1.7.²

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, $\text{view}[\text{toto}[a,b], \text{titi}]$ corresponds to the view:

```
  view
   toto
     titi
      a
      b
```

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 $\cal A^*$ and $\cal T$ is the set of all VDL terms.

2.3 Semantics

We present here the *procedural concepts* used by $\phi_{\text{exec}}$ to define the operational semantics of the views.

2.3.1 Basic actions

We showed in [23] that there exists only three elementary actions to modify a view, which form a basis (each one cannot be defined in function of the two other). Each action is described by a specific concept used by $\phi_{\text{exec}}$:

- **add** to add attributes to a term,
- **put** to replace all the attributes of a term by others,
- **del** to remove a term from the view.

In each basic action’s description, we can use two other concepts to refer other subterms of the view: **path** and **get**. A reference $r = [t_1, ..., t_n]$ is a series of VDL terms describing a path in the view’s tree. The **path** and **get** concepts are used to give references inside an action.

²Demos are available on our web page: [http://www.limsi.fr/Individu/nico/examples](http://www.limsi.fr/Individu/nico/examples)
Example. If \( v \) is the following view:

\[
v = \text{view} \{ \text{table}, \text{add[\text{path[\text{table}]}, \text{pencil}] \}
\]

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

\[
\phi_{\text{exec}}(v) = \text{view} \{ \text{table[pencil]}, \text{add[\text{path[\text{table}]}, \text{pencil}] \}
\]

i.e. the initial view with \( \text{pencil} \) added in the attributes of the \( \text{table} \) term, corresponding to the reference \{\text{table}\} in the term with concept \text{path}.

2.3.2 Arithmetics

We also defined some concepts corresponding to the most common arithmetic operators: \text{plus} and \text{times} for addition and multiplication, \text{minus} and \text{inverse} for the opposite and the inversion; \text{equals} and "greater than" for relations between terms; \text{and}, \text{or}, \text{not, true} and \text{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 at runtime.

The \text{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:

\[
\text{view[\text{value[0]}, \text{"add one"[ \text{event[\text{foo}], put[\text{path[\text{value}], plus[\text{get[\text{value}], 1}]}} ]}}
\]

The \text{value} term’s value is incremented by the "add one" action each time the component gets the \text{foo} event.

Moreover, we can provide basic actions with \text{boolean preconditions} described in VDL, using the the \text{guard} concept. For instance, in the view:

\[
\text{view[\text{value[0]}, \text{"add one"[ \text{guard[\text{"greater than"[5, get[\text{value}]}, a[...]}} ]}}
\]

When this component is active, the action \text{a} in “add one” is performed at each execution step as long as \text{value} is lesser than \text{5}, without any user interaction.

2.4 Definitions

- Every term \( t \) such that \( \text{concept}(t) \in \{\text{add, put, del}\} \) is a \text{basic action}.
- Every term \( t = \text{c}[s_{t1}, ..., s_{tn}] \) such that \( \exists i \in [1, n] \mid \text{concept}(s_{ti}) \in \{\text{guard, event, add, put, del}\} \) (i.e. having at least one attribute whose concept is \text{guard, event, add, put or del}) is an \text{action}.
- The \text{preconditions} of an action \( a \) are given by the set \( \Pi_a \) of all terms with concept \text{guard} or \text{event} within its siblings, uncles, great-uncles and the siblings of all ancestors:

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

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

\[
\pi(a) = \{ t \mid \text{parent}(a) = \text{parent}(t) \} \cup \pi(\text{parent}(a))
\]
Every action \( a \) such that \( \forall t \in \Pi_a, \text{concept}(t) = \text{guard} \) is a process.

Every action \( a \) such that \( \exists t \in \Pi_a, \text{concept}(t) = \text{event} \) is a reaction.

A component’s \textit{behaviour} is defined by the set of all its actions and its \textit{activity} is defined by the set of its processes.

Components described in VDL can thus work either in a modal way, \textit{i.e.} in reaction to external events sent by the user (like \texttt{CGI} scripts that process a query to produce a new web page) or in a modeless way, \textit{i.e.} without any interaction (like an applet or a flash animation). VDL processes represent modeless behaviours.

### 2.5 Example: \textit{Coco}

In this document, we will often use a voluntarily rather simple example. It is a mere counter called \textit{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 \textit{italics} for easier reading.

```plaintext
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]]
]
```

Coco is provided with a process (\texttt{count}) which increases the counter’s value and four reactions: \textit{stop}, \textit{start} to stop or restart the counting process, \textit{speed up} and \textit{slow down} to modify the speed.

### 2.6 Natural Language Questions Processing

In this paper, we take an interest in processing formal requests coming from natural language questions that a human user can ask about the actions of an active component. The Natural Language Processing Module (NLPM) developed by S. Gérard \cite{11} in the \textit{InterViews} project matches action concepts with the user’s question using a semantic classification such as \textit{WordNet} \cite{7}. It thus builds a formal request whose arguments are part of the procedural description of the component.

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. Answers to questions in an integrated environment like the \textit{InterViews} project result then from an interaction between the NLPM and the Reasoning about Actions Module (RAM), each one solving specific problems.

The mechanism used within the NLPM, based on well defined rules, is too complex to be presented in this paper.

\(^3\)You can interact with \textit{Coco} on the web: \url{http://www.limsi.fr/Individu/nico/examples/coco.html}
Examples

Here are some examples of questions about actions for Coco, coming from our corpus, that can be represented and answered with the model presented in this paper. We also give the answer the NLPM should be able to produce:

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

In the next section, we will give examples of such request. Note that all the question of our corpus can be represented using our model.

3 A model of request about actions

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

3.1 Presentation of the model

In our model, a request about actions is defined by a sextuple:

\[ \rho = (\alpha, \tau, \nu, \sigma, \omega, \delta) \]

The elements of the request correspond to six characteristic criteria used to take into account all the relevant information contained in the user’s question.

The answer build by the RAM are also requests. We give many examples in section 3.2.

3.1.1 The request act: \( \alpha \)

The \( \alpha \) criteria is the request act. Each request act corresponds to a specific kind of request, as *speech acts* [28] define the interpretation of a dialog in Natural Language Processing. Request acts define the question’s frame. For questions about actions, we have the following acts:

- **What** to specify that the question’s object (given by \( \omega \)) is unknown. For instance, “*What are you doing?*”.
- **Ask** to specify that the question’s object is supposed to be a possible answer. For instance, “Is your value 42?” or “Do you count?”.

- **Why** for diagnosis questions about the component’s past runtime, dealing with either states or actions. For instance, “Why did you stop?”.

- **How** for plan generation requests (for instance: “How can I stop you?”) or questions about an action’s body (for instance: “How do you count?”).

- **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 and, as a general rule, all RAM answers will have $\alpha = Assert$.

- **Error** to specify that the given request is not syntactically correct, and **Unknown** to specify that none of the arguments proposed in the $\omega$ criterion could be matched with a view’s subterm ($\forall t' \in \omega, t' \notin subterms(v_t)$).

These last two acts are only used within answers, in case of errors.

### 3.1.2 The type: $\tau$

$\tau$ 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 [22]). 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 $(\alpha, \tau)$ corresponds to a specific kind of question and a specific processing for the RAM.

### 3.1.3 The $\nu$ boolean

The $\nu$ criterion is a boolean mark. Its semantics depends on the question’s frame, defined by $(\alpha, \tau)$. For instance, when $\tau = do$, $\nu$ is generally used to indicate whether the question is about active processes ($\nu = true$) or inactive ones ($\nu = false$).
3.1.4 The subject: $\sigma$

$\sigma \in \mathcal{P}(\Upsilon)$ is a reference to the question’s subject, i.e. the view’s subterm which is concerned by the question (it might simply be the whole view). For example, for “Is your value 47?”, $\sigma = \{value\}$. The reference module guarantees both the existence and the uniqueness of the $\sigma$.

3.1.5 The object: $\omega$

$\sigma \in \mathcal{P}(\Upsilon)$ is the request’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), as we will see in the examples given in section 3.2. For instance, for “Are you counting or sleeping?”, $\omega = \{count, sleep\}$.

3.1.6 The date: $\delta$

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:

- $\delta = \sqrt{\cdot}$ for present,
- $\delta = date[x] \ (x \in \mathbb{Z})$ 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),
- $\delta = date[x, \delta'] \ (x \in \mathbb{Z}$ and $\delta'$ a date) represents a date shifted by $x$ steps from $\delta'$,
- $\delta = global[x] \ (x \in \mathbb{N})$ 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 $x$.

We do not present here the mechanism used to answer questions about the component’s past and future execution. Our aim is to provide every subterm in the view with a chronicle [24] that catches and register the situations corresponding to past event and action occurrences. This chronicle is intensionalised in order to extract its regularities, used as a support to a reasoning model based on dynamic analysis on the component’s behaviour. An originality of this work is to use in the same description a static representation of actions (like in situation calculus [18]) and a memorisation of past executions with chronicles. We are now working on heuristics that could help reduce the size of the chronicles and the complexity of our search algorithms.

3.2 Answering requests

3.2.1 General principle

While building an answer to a request about actions, the RAM searches the view tree for either values of variables or action (depending on $\tau$) in the subterms of $\sigma$, using $\omega$ as a filter. We categorised the different possible cases in [25].

If $\alpha \in \{What, Ask\}$, then the answer has the same type ($\tau$) as the request, with the act $\alpha = Assert$ and its object $\omega$ is the result of this search.

---

4The reference problem in real life (i.e. not constrained natural language dialogs) is still an open problem. However, we suppose for our study that $\sigma$ was correctly given by the NLPM. This can always be achieved by restraining the NL interactions.
Examples

Let us consider a simple but representative example of question about the component’s activity: “What are you doing?”. The NLPM translates it into the request:

\[
\langle \text{What}, \text{do}, \text{true}, \{\text{view}\}, \emptyset, \sqrt{\text{.}} \rangle
\]

From a procedural point of view, it consists of looking in the view for active processes (\(\tau = \text{do}\)) with no filter (\(\omega = \emptyset\)). Using a rather simple search algorithm (see [26]), the RAM can build the following answer:

\[
\langle \text{Assert}, \text{do}, \text{true}, \{\text{view}\}, \{\text{count}\}, \sqrt{\text{.}} \rangle
\]

that will be used by the NLPM to produce: “I’m counting”.

Now, we can also have questions about the activity in which the object actions are given. For instance: “Are you counting or sleeping?”. The NLPM translates it into the request:

\[
\langle \text{Ask}, \text{do}, \text{true}, \{\text{view}\}, \{\text{count}, \text{sleep}\}, \sqrt{\text{.}} \rangle
\]

Then the RAM looks in the view for active processes (\(\tau = \text{do}\)) with concept \text{count} and \text{sleep}. It first looks for the corresponding actions (considered as simple VDL terms). Depending on the request type (\(\tau = \text{do}\)) and action preconditions, it then filters the set of VDL terms that was found so as to build the correct answer. The answer strongly depends on what kind of actions were found. For example, in a counting \text{Coco}, we shall have:

\[
\langle \text{Assert}, \text{do}, \text{true}, \{\text{view}\}, \{\text{count}\}, \sqrt{\text{.}} \rangle
\]

that will be used by the NLPM to produce: “I’m counting”. However, for another component, if the actions were not processes but reactions, or if no such action were found, the answer would have been completely different.

Finally, let us consider a simple but representative example of question about the component’s state: “What is your speed?”. The NLPM builds the request:

\[
\langle \text{What}, \text{is}, \text{true}, \{\text{speed}\}, \emptyset, \sqrt{\text{.}} \rangle
\]

The RAM looks for the value of terms corresponding to the reference \{\text{speed}\} to build the answer. A rather simple algorithm leads it to produce:

\[
\langle \text{Assert}, \text{is}, \text{true}, \{\text{speed}\}, \{7\}, \sqrt{\text{.}} \rangle
\]

that will be translated into: “My speed is 7”.

Reference problem

We have to face a reference problem [21] in \(\omega\) when \(\tau \in \{\text{do}, \text{can}\}\), even in the simple case where \(\alpha \in \{\text{Ask, What}\}\), since the elements of \(\omega\) refer to possible actions in the view. Each element in \(\omega\) can refer to 0, 1 or \(n\) actions. For instance, in \text{Coco}, the question “Are you counting or sleeping?” is translated into the request:

\[
\langle \text{Ask}, \text{do}, \text{true}, \{\text{view}\}, \{\text{sleep, count}\}, \sqrt{\text{.}} \rangle
\]

but no sleep action exists in the view!

In the current RAM algorithms implementation, we use the following partial solution:

- if no action is found, then the answer has the form \(\langle \text{Unknown}, \text{is}, \text{true}, \sigma, \omega, \delta \rangle\).

- otherwise, let \(\omega'\) be the set of actions found with the correct type (processes or reactions, active or not) depending on \(\tau\) and \(\nu\). If \(\omega' \neq \emptyset\), then the answer is \(\langle \text{Assert}, \tau, \nu, \sigma, \omega', \delta \rangle\). Otherwise, it has the form \(\langle \text{Assert}, \tau, -\nu, \sigma, \omega' \setminus \omega, \delta \rangle\) where \(\nu\), \(\sigma\) and \(\omega\) are given by the initial request.
3.2.2 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 \( \omega \).

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).

For instance, to the question “Can you count?”, corresponding to:

\[
(\text{Ask, can, true}, \{\text{view}\}, \{\text{count}\}, \sqrt{\sigma})
\]

we should answer:

- “I can count.” when the guard of the count action is true:
  \[
  (\text{Assert, can, true}, \{\text{view}\}, \{\text{count}\}, \sqrt{\sigma})
  \]
- “I can’t count: I’m not running.” otherwise:
  \[
  \{ \langle \text{Assert, can, false}, \{\text{view}\}, \{\text{count}\}, \sqrt{\sigma} \rangle; \\
  \langle \text{Assert, is, false}, \{\text{running}\}, \{\text{true}\}, \sqrt{\sigma} \rangle \}
  \]

The answer is thus a set of requests, the first one being the negative answer, other ones representing the invalid preconditions. Each \( \overline{\sigma_i} \) concerns only one invalid precondition.

As a consequence, requests with type \( \tau = \text{can} \) are not as simple as request with type do since they must take into account different schemes.

3.2.3 Plan generation: \( \alpha = \text{How} \)

If \( \alpha = \text{How} \), the RAM builds a plan and the answer is the expression, as a set of requests, of the different actions of the plan.

Our plan generation uses a STRIPS-like method [8], 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.

3.2.4 Explanation generation: \( \alpha = \text{Why} \)

If \( \alpha = \text{Why} \), the answer gives the different values and actions explaining the state or actions in the request object (depending on \( \tau \)).

Classical diagnosis tools build a set of literals corresponding to the cause of a given state. On the contrary, explanatory diagnosis [19] builds a sequence of actions corresponding to “what happened”.

The mechanisms being developed in the RAM, based on static consequences\(^5\) collection and causal reasoning 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 (\( \tau = \text{do} \)) or states (\( \tau = \text{is} \)).

\(^5\)We call static consequences the action’s result that do not 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.
Example

For the question “Why do you count?”, we shall have the request:

\( <\text{Why}, \text{do}, \text{true}, \{\text{view}\}, \{\text{count}\}, \sqrt{\text{v}} > \)

The RAM looks in the view for valid preconditions of the \textit{count} action and builds the answer:

\( <\text{Assert}, \text{is}, \text{true}, \{\text{running}\}, \{\text{true}\}, \sqrt{\text{v}} > \)

that will be used to produce: “Because I’m running”.

The ramification problem

The ramification problem [9] is the impossibility to describe with logical formulae all the indirect consequences of an action. In VDL and in all other programming languages, the actions “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, \textit{i.e.} to discover the actions responsible for the modifications described by the user and, because of the ramification problem, this can’t be reduced to a single matching of the right formula. Moreover, we encounter two major difficulties:

- The first one is to determine what action is the actual \textit{cause} [20], \textit{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 \( <\text{Why}, \text{do}, \text{true}, \{\text{view}\}, \{\text{count}\}, \sqrt{\text{v}} > \) (“Why do you count?”) required to look for actions whose consequences validate the \textit{count} action’s guard.\(^6\)

We are currently studying algorithms based on logical approaches [4, 16] to “trace back” ramifications. Our aim is to use an explanatory diagnosis method such as [19] that could determine “what happened”, in order to determine the \textit{cause} of a \textit{modification} that we have to give in the answer, be it a \textit{state} or an \textit{action}.

Answering \textit{Why} request is however very difficult since it involves many fundamental issues, such as the ramification problem. We do not pretend to have solve this case and we are still studying efficient answering heuristics.

3.2.5 Temporal requests

We can use the date to formalise questions about the view’s past or future execution. For instance, “What will your value be within 10 steps?” corresponds to the request:

\( <\text{What}, \text{is}, \text{true}, \{\text{value}\}, \emptyset, \text{date}[10] > \)

The hypothesis is that the user will not disturb the component’s execution. Using the chronicle model briefly presented in section 3.1.6, we can then build the following answer:

\( <\text{Assert}, \text{is}, \text{true}, \{\text{value}\}, 42, \text{date}[10] > \)

\(^6\)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 \( \rho = < \text{why}, \text{is}, \text{true}, \{\text{view}\}, \{\text{stop}\}, \sqrt{\text{v}} > \) (\textit{stop} is an \textit{action} and thus can’t be in the \( \omega \) set of a request with type \( \tau = \text{is} \)).
“My value will be 42 within 10 steps.”.

The date model we presented makes it possible to represent more complex questions. For instance: “What was your value 10 steps before I changed your speed?” can be represented by:

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

Finally, we can use the When act, like for “When will your value be 10?”, which corresponds to:

\[ \langle \text{When}, \text{is}, \text{true}, \{\text{value}\}, \{10\}, \text{future} \rangle \]

In both cases, the RAM only has to look in the chronicle for the corresponding value. The use of chronicles turns difficult questions about the component’s past into very simple search in a table [24].

4 Conclusion and perspectives

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

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 and the web.

Active components like those we propose to describe in VDL are already integrated in the Internet, as applets displayed in the web page of the InterViews project\(^7\). We now intend to integrate them in the semantic web. In order to do this, we propose to use the VDL language in a more expressive structural model, so as to obtain a real procedural XML-based representation. We intend to make use of XML rewriting tools like XSLT [1].

The mechanisms we use within the RAM are general enough to answer questions about actions for any component described in our formalism VDL [23]. However, we saw that answering requests leads to new issues in reasoning about actions (reference problem, explanation with ramifications, ambiguity of can, etc). We think that the study of these problems shows new perspectives in this domain and especially in explaining in natural language the functioning of actions and services in the semantic web.

References


\(^7\)http://www.limsi.fr/Individu/nico/examples


