A Logic (Programming) based Controller for Smart Environments

Sebastian Bader
Institute of Computer Science, Rostock University, Germany
first.last@uni-rostock.de, http://mmis.informatik.uni-rostock.de

Abstract. We describe an implementation of a logic (programming) based controller for an instrumented environment. The controller is able to assist users of a smart meeting room in setting up the equipment and the environment. The goal is to define a general scheme for the construction of controllers. Due to the modular structure of the implementation it is easily extendable and applicable to other application areas. Based on the concrete implementation we discuss implications on the representation of and the reasoning within a given context.

Keywords: logic program, smart environment, user preferences

1 Introduction and Motivation

As more and more technical devices are embedded in our environment the interaction becomes increasingly complex. A good example are smart meeting rooms. In this paper, we present a simple controller running in our meeting room to support its users. It is based on a modular implementation in Prolog, which is not yet meant to solve every problem, but to be easily extendable. After presenting a typical scenario, we continue with a discussion of problems and related work. In Section 2 we describe our implementation in detail.

Assume a meeting room equipped with a number of projectors, screens, sunshades, whiteboards and a video cross bar that can connect every computer to every projector. The room is also equipped with a positioning system and a sensor to detect whether a laptop has been attached to some VGA-cable. The screens can move up and down, and some of them are blocking whiteboards behind them, which can detect if a user has taken a pen to draw a picture. An intelligent room should support its users as good as possible. For example it should automatically show the contents of every connected laptop on some screen. It should also move up a screen whenever the user takes a pen to draw on the whiteboard behind the screen.

To create a controller for the example described above and for similar applications, we definitely need to

1. describe the capabilities of the room and its devices,
2. model the users intentions, goals and preferences,
3. describe or infer possible resource conflicts,
4. detect the current context, i.e., the current state of the world and to
5. infer the best sequence of actions to actually support the user.

In several projects researcher around the world try to address these and similar problems. To name just a few we would like to point to projects like MAVHOME [CHGY03], the AWARE HOME [KPJ+08], the INTELLIGENT CLASSROOM [Fra98] and the EMBASSI-project [KHS01]. A nice introduction into the general area of smart environments and its challenges can be found for example in [CD05,CFBS97] and [EBM05].

But even though a number of systems are around for quite some time now, most intelligent rooms are as intelligent as they used to be 10 years ago, only more complex due to the larger number of available devices. In this paper, we describe a very small and simple system which is able to control an environment as described above. All necessary steps of inference are shown on a logical level as well as on an implementation level.\(^1\) The controller itself is completely implemented in PROLOG.\(^2\) We hope that by discussing a possible realisation on a very detailed level we enable others to follow our road and to build more intelligent rooms in the future.

After describing the controller in detail we discuss the general methodology behind it on a more abstract level and finally we try to identify a set of language constructs necessary to model both: an intelligent controller and user preferences.

## 2 A Realised Controller

In this section, we discuss our implementation of a controller for an instrumented environment in some detail. The controller is split in five independent parts:

1. Detection of the intentions of the users
2. Goal inference and refinement
3. Identification of maximal consistent subsets of goals
4. Computation of possible action sequences to fulfil the goals
5. Determination of the best sequence of actions

We rely on PROLOG as a description language to have its full descriptive power at hand. If at some later point, it turns out that a less powerful language suffices to describe all necessary rules, we might use a custom made language instead. In Section 3 we will briefly discuss a number of prerequisites for such a language. PROLOG rules have the following form:

\[
\text{conclusion} :: \text{condition1, condition2.}
\]

\(^1\) The complete code and a small demo are available during the workshop and as download from the authors homepage.

\(^2\) For the implementation we used SWI-PROLOG: [http://www.swi-prolog.org](http://www.swi-prolog.org)
and can be read as implications from the right to the left. I.e., the above example represents the rule \( \text{condition}_1 \land \text{condition}_1 \rightarrow \text{conclusion} \). Rules end with a dot and rules without \( :- \) represent facts. Words starting with an upper case letter indicate variables (written in italic letters below), whereas lower case words represent constants. \([1,2,3]\) indicates a list of three elements. We use some built-in predicates like \( \text{member}(\text{Element}, \text{List}) \) and \( \text{findall}(V, \text{condition}(V), \text{Vs}) \) meaning that \( \text{Element} \) is an element of \( \text{List} \) and that \( \text{Vs} \) should be unified with the list of all instances of \( V \) fulfilling the given condition, respectively.

The controller itself consists of a number of sub-programs described in detail below – basically one for each of the steps listed above. To separate the configuration of the room from the other definitions, it is encoded in the program \( \Delta_K \) as shown in Figure 1. The configuration we use is a simple list of named properties.

\begin{verbatim}
config(lectureLocations, [stage1, stage2]).
config(vgaInputs, [vga1, vga2]).
config(screens, [screen1, screen2, screen3]).
\end{verbatim}

Fig. 1. Part of the configuration details of the environment.

### 2.1 Interfacing with the World

To give the controller access to the current state of the world, it is given a set of assertions in the form of triples. Each triple consists of an object, a property name and a value as for instance:

\[ \langle \text{User}_1, \text{location}, \text{Stage}_1 \rangle \]

denoting that some user \( \text{User}_1 \) is at the location \( \text{Stage}_1 \). Whenever the state of the world changes, the set of triples is updated accordingly and given to the controller as a set of facts as presented in Example 1. For a given set \( S \) of triples, the corresponding program \( P_S \) (of facts) is defined as

\[ P_S := \{ \text{worldstate}(O, P, V) \mid \langle O, P, V \rangle \in S \} \]

**Example 1.** As a running example let us consider the following state of the world:

\begin{verbatim}
worldstate(user1, location, stage1).
worldstate(vga1, connected, true).
worldstate(window1, open, true).
\end{verbatim}

This program states that the user \( \text{user1} \) is at the location \( \text{stage1} \), that there is a laptop connected to the input \( \text{vga1} \) (in our lab \( \text{vga1} \) denotes a VGA-input on a table to connect laptops) and that \( \text{window1} \) is open.
2.2 Inferring Intentions from the State of the World

As described above, we first try to infer a set of current intentions from the state of the world. The set $\mathcal{I}$ of intentions is defined as the logical consequence of some program $\Delta_f$ and the program $P_S$ as defined above as follows:

$$\mathcal{I} := \{ I \mid \Delta_K \cup P_S \cup \Delta_f \models \text{intention}(I) \}$$

The program $\Delta_f$ basically determines how to infer intentions from the state of the world and thus encodes the background knowledge we have about possible user intentions. To infer all current intentions we use the predicate $\text{intentions/1}$ shown in Figure 2. The figure does also show the definition of the intention $\text{lecture}(L)$ which is parametrised by some location $L$ meaning that there is a user giving a lecture at that location.

Example 2. Using the state of the world from Example 1 the system infers:

```prolog
?- intentions(Is).
Is = [lecture(stage1), discussion([vga1])]
```

Meaning that there is a lecture to be giving at location $\text{stage1}$ and that there is a discussion involving the input $\text{vga1}$.

Fig. 2. The predicate $\text{intentions/1}$ to find all current intentions and a description of the intentions $\text{lecture}(L)$ for stage L and $\text{discussion(Vs)}$ for the VGA-Inputs Vs. The variable $\text{User}$ is a free variable meaning that there must be at least one user at location L.

2.3 Inferring Goals from Intentions

After computing the set of current intentions we need to determine the corresponding set of goals to be satisfied by the controller. For this we compute the logical consequences of some program $\Delta_g$ with respect to the intentions $\mathcal{I}$:

$$\mathcal{G}(\mathcal{I}) := \{ G \mid I \in \mathcal{I}, \Delta_K \cup \Delta_g \models \text{goal}(I,G) \}$$
Figure 3 shows the implementation of the corresponding predicate and of the goal for the intention discussion.

**Example 3.** The goals corresponding to the current set of intentions are:

```
?- intentions(Is), goals(Is, Gs).
Is = [lecture(stage1), discussion([vga1])]
Gs = [show(vga1), setupLights(stage1)]
```

The part of the code to infer `show(vga1)` is shown in Figure 3.

The predicate `goals/2` to find all goals and the definition of a goal for the intention `discussion` stating that every connected input should be shown.

So far we have not been concerned with consistency (with respect to some given background knowledge) of the set of current goals, but it can very well be that there are goals contradicting each other. For example, it could be that the goal `closeSunShade(window1)` and `freshAir(window1)` are inferred at the same time, which – at least in our lab – is impossible because the sunshade can not be closed while the window is open. How to solve this problem is discussed below, but first we discuss our approach to refine goals.

### 2.4 Refining the Set of Goals

Some of the goals identified so far are not specific enough to result in executable actions. E.g., one of the goals mentioned above is `show(vga1)`, but this does not specify where to show the content coming from input `vga1`. Therefore we have to refine the set of goals resulting in a set of possible refinements. Possible refinement steps are defined in some program $\Delta_R$ and we recursively compute the set of refined sets of goal-sets $\mathcal{R}(\mathcal{G})$ as follows:

$$\mathcal{R}(\mathcal{G}) := r(\{\mathcal{G}\}) \text{ with }$$

$$r(\mathcal{G}) := \bigcup_{G' \in \mathcal{G}} \bigcup_{g' \in G'} r'(G', g') \text{ with }$$

$$r'(G, g) := r(\{G \setminus g \cup g' \mid \Delta_K \cup \Delta_R \models \text{refinement}(g, g')\})$$

This definition simply replaces every refine-able goal with all possible refinements and this is done in all possible combinations. The implementation of `refinements`-predicate and the definition to refine the above mentioned `show` goal is given in Figure 4.
Example 4. For the goal \texttt{show(vga1)} we obtain three refinements \texttt{show(vga1, screen1)}, \texttt{show(vga1, screen2)} and \texttt{show(vga1, screen3)}.

\begin{verbatim}
refinements(Goals, RefinedGoals) :-
    select(G, Goals, RGoals),
    findall(RG, refinement(G, RG), PRs),
    member(PR, PRs),
    refinements([PR | RGoals], RefinedGoals).
refinements(Goals, Goals).
refinement(show(V), show(V, S)) :-
    config(screens, Screens), member(S, Screens).
\end{verbatim}

Fig. 4. The definition of the refinement and a predicate defining possible refinements of the \texttt{show/1} goal.

2.5 Finding Maximal Consistent Sets of Goals

As argued above, the computed (refined) sets of goals are not necessarily consistent. Obviously one could interweave the definitions of intentions, goals and refinements with integrity constraints, but this will render those definitions unreadable and furthermore make them depend on each other. Therefore we follow another road by collecting all goals as above and detect consistent subsets afterwards. Given a set of refined sets of goals \( R \) we compute the set of maximal consistent subsets of goals with respect to some program \( \Delta_C \) specifying consistency constraints as follows:

\[
\mathcal{M}'(R) := \{ M | R \in R, M \subseteq R \text{ and } \Delta_K \cup \Delta_C \models \text{consistent}(M) \}
\]

\[
\mathcal{M}(R) := \{ M | M \in \mathcal{M}'(R) \text{ and there is no } M' \in \mathcal{M}' \text{ with } M \subset M' \}
\]

The set \( \mathcal{M}' \) contains all consistent subsets of goals with respect to the definition \( \Delta_C \) of consistency and \( \mathcal{M} \) contains only those which are no proper subsets of other consistent sets. For all those sets we compute the set of corresponding actions next and evaluate those actions with respect to some cost function afterwards. Figure 5 shows the definition of a consistent set of goals, which is defined as containing no conflict. The figure also shows the definition of a conflict arising from two goals which require different contents to be shown on the same screen.

2.6 Inferring Actions from Goals

After identifying the set of all refined maximally consistent sets of goals, each of them is mapped to a sequence of actions. In simple scenarios this can be done by
consistent(Gs) :- \+ conflict(Gs).

conflict(Gs) :-

member(show(V, S), Gs),
member(show(W, S), Gs), V \= W.

**Fig. 5.** The definition of a consistent set of goals simply checks whether the set contains a conflict. In this case it is not consistent, otherwise it is. According to the given definition of conflict a set of goals is inconsistent if two different inputs V and W shall be shown on the same screen S.

encoding the direct mappings from a goal to sequence of actions or alternatively employing a planning algorithm to find a plan to achieve a set of goals. In our case the action show(V, S) will be mapped to a sequence of actions which show the input V on screen S by turning on the corresponding projector and setting up the video cross-bar such that V and the projector are connected. For a given program $\Delta_A$ defining the actions with respect to goals we compute the set of actions for a given set of goals $G$ as follows:

$$A(G) := \bigcup_{g \in G} \{ a | \Delta_K \cup \Delta_A \models action(g, A), a \in A \}$$

The definition of a predicate collecting all actions for a given set of goals is shown in Figure 6, which also contains a definition of actions corresponding to the goal show(V, S), i.e., a small part of our program $\Delta_A$.

**Fig. 6.** The predicate actions to find all actions corresponding to a set of goals and the (static) definition of the actions corresponding to the goal show(V, S).

**Example 5.** Reconsidering the refined the set of goals from above [show(vga1, screen1), setupLights(stage1)] we obtain the following sequence of actions [screen1:moveDown, proj1:turnOn, proj1:unmute, lamp1:dimLight(0.5), crossbar:bind(vga1, proj1)]. Executing this sequence would put the room in a good state to give a lecture.
2.7 Finding an Optimal Sequence of Actions

After identifying all possible sequences of actions we need to identify one which should be executed. The selection is done by minimising a given cost function, which can be used to model user preferences. This cost function is specified by attaching costs to certain actions. To compute the cost of a given sequence of actions, we simply sum up the costs over all actions and pick the sequence with minimal costs. I.e., for a given set $\mathcal{A}$ of sequences of actions we select the sequence with minimal cost as follows:

$$\text{best}(\mathcal{A}) := \arg \min_{\mathcal{A} \in \mathcal{A}} \sum \{ p \mid a \in \mathcal{A}, \Delta_K \cup \Delta_R \models \text{price}(a, p) \}$$

Figure 7 shows a part of the cost function which result in preferring screen2 over screen1 because it is cheaper to move down the former. To compute the total cost of a given sequence of actions we simply need to sum up the costs over all actions involved. Afterwards we can select the sequence with the minimal total cost and execute it.

$$\text{price}(\text{screen1:moveDown}, 4).$$
$$\text{price}(\text{screen2:moveDown}, 2).$$

Fig. 7. Part of the specification of the cost function. This snippet already results in a preference of screen2 over screen1 because it is cheaper to move it down.

For now we use a hard-coded cost function, but as again discussed below, this could also be learnt by some machine learning algorithm from the user interactions. But the specification of our lab contains less than ten cost statements and does already cover most interesting cases. Nonetheless the cost function is the most easy way of actually modifying the behaviour of the room. A possible extension of a simple cost function is to not only consider the costs but also some kind of quality of the achieved solution and then balance both values. But using a simple cost function seems to suffice for at least some cases and it is very easy to specify.

3 Summary and Conclusions

Even though intelligent controllers for instrumented environments exist for some time, they are not available for a daily usage yet. The purpose of this paper has been to present a modular controller which covers at least some interesting aspects of environment control and is as simple as possible and hence very easy to get to work and to set up. The proposed system has been implemented with respect to the following ideas and assumptions:
– The principle steps of an environment control system as listed in Section 2 should be as independent as possible.
– The capabilities of devices should be formalised independently, with overlaps only while specifying possible conflicts.
– User preferences are modelled using a simple cost functions over actions.

The proposed system has some implication. In particular, there are no feedback loops from later stages of the process into the former ones. This has been done on purpose to keep the system as simple as possible. And so far this feedback has not been necessary. In the system proposed above, we basically specified the steps of the inference process and its interfaces as summarised in Table 1.

<table>
<thead>
<tr>
<th>Inference Step</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recognise intention</td>
<td>Set of atomic intentions</td>
</tr>
<tr>
<td>Infer and refine goals</td>
<td>Set of sets of goals</td>
</tr>
<tr>
<td>Identification of consistent goals</td>
<td>Set of sets of goals</td>
</tr>
<tr>
<td>Computation of action sequences</td>
<td>Set of sequences of actions</td>
</tr>
<tr>
<td>Determination of best actions</td>
<td>Sequence of actions</td>
</tr>
</tbody>
</table>

Table 1. Steps of inference and representation of the result

The described controller contains a (completely independent) program for each of the five inference steps. And each of the programs has access to the current state of the world, global configuration information and the result of the predecessor step.

In the near future we will cover increasingly more complex situations without leaving the architecture and implementation described above. After gaining more experience with our controller running on a daily basis and supporting real lectures and meetings, we will try to identify a set of sufficient and necessary language constructs. Those will probably include:

– if-then-rules as a general schema,
– negation to model the absence of certain situations and objects, and
– a possibility to identify all possible objects fulfilling a given condition.

So far we did not model time-dependent conditions, but this will probably be necessary. For example a projector should not be turn off immediately after disconnecting the laptop, because the laptop might have been disconnected to be replaced by a second one. Turning the projector off in this situation would probably not reflect the users intention and might even damage the hardware due to repeated on- and off-switching. Therefore, we will very likely have to include time-constructs or model them as properties of the environment which can then be accessed using the constructions presented above.
As a possible future extension of the system we will replace the currently fixed cost-function by some trainable system. This would enable the whole controller to adapt to user preferences in a dynamic way.

Another possible extension of the system is to base it completely on some probabilistic calculus. That is, all currently crisp statements are extended with a probability value. This might allow to infer the probability that a given sequence of actions does indeed yield a desired state.

Acknowledgements

The author is thankful for the comments of two anonymous reviewers who helped to improve the quality of this article.

References