Modelling Device Actions in Smart Environments

Christiane Plociennik, Christoph Burghardt, Florian Marquardt, Thomas Kirste, and Adelinde Uhrmacher

University of Rostock, Albert-Einstein-Str. 21, 18059 Rostock, Germany
firstname.lastname@uni-rostock.de

Abstract. Smart environments are places that contain numerous devices to assist a user. Those devices' actions can be modelled as planning operators. A problem when modelling such actions is the *persistent action problem*: Actions are not independent of one another. This is especially relevant when regarding persistent actions: An action that is being executed over a longer timespan may be terminated by a subsequent action that uses the same resources. The question is how to model this adequately. In dynamic environments with a high fluctuation of devices an additional challenge is to solve the persistent action problem with as little global information as possible. In this paper, we introduce two approaches: The first one locks resources which are being used by an action to prevent other actions from using the same resources. The second interleaves planning and execution of actions and is thus able to use software agents as "guards" for actions that are being executed. We furthermore compare the characteristics of both approaches and point out some implications those characteristics have on the modelling and execution of device actions in smart environments.

1 Introduction

Smart environments are places that contain numerous devices to help users accomplish certain tasks. Meeting rooms, for example, are typically equipped with projectors, canvasses, computers, cameras, and lights. In addition, users can bring mobile devices with them. What makes a smart meeting room smart is that it is able to integrate all those devices into one coherent ensemble that proactively assists the user, enabling her to focus on her core activities rather than on configuring the environment. For example, if the user has the intention to give a talk, a smart meeting room should relieve her of the tasks of manually connecting her notebook to the projector, adjusting the light level, etc. The device ensemble should perform these tasks for the user. Many researchers are concerned with the question how to enable such assistance [8,12]. One particularly promising approach is to assist users in smart environments using AI planning [7,1].

The key elements in classical AI planning are *operators*. These operators are actions that are described in terms of *preconditions* and *effects*. Preconditions

D. Tavangarian et al. (Eds.): IMC 2009, CCIS 53, pp. 213–224, 2009.

and effects are conjunctions of propositions that can be either true or false. For the action to be executed, its preconditions must hold. After the execution of the action, its effects hold. Consider the simple planning operator *CanvasDown*:

```
(:action CanvasDown
  :parameters (?c - Canvas)
  :precondition (not (CanvasDown ?c))
  :effect (CanvasDown ?c))
```

This operator is described in PDDL [11], a language widely used for planning problems. It describes the action of lowering a canvas. It has a single precondition which states that the canvas must not be down for the action to be executed. After execution the world state will have changed: now the canvas is down.

A planning problem consists of a domain description, a set of objects, a set of true conditions specifying the initial world state (all conditions not mentioned are assumed to be false) and a set of conditions specifying the goals of the planning process. The domain description is a set of operators. Objects are used to instantiate planning operators: All variables in operator descriptions are bound to an object. The variable *?c* in the operator description of *CanvasDown* is instantiated with all objects of type *Canvas* defined in the problem description. Thus, for each *Canvas* object one instance of the CanvasDown operator is generated. To solve a planning problem means to find a sequence of instantiated operators (a *plan*) which transforms the initial world state into the goal state. A comprehensive introduction to planning is beyond the scope of this paper, but can be found in [14].

The possible actions of devices in smart environments can be modeled as planning operators. This has the advantage that user assistance can be very flexible. Whenever a new user goal becomes apparent, a planner can consider all possible actions of all devices in the ensemble and search for a sequence that fulfills the goal. The action sequences need not be precompiled by a domain expert. Every device can carry descriptions of all its possible actions. Upon entering a new environment, it can provide these descriptions to the devices already present. This way, the device ensemble is constructed of modular pieces and can be dynamically extended. This is key for smart environments which are typically characterized by a high fluctuation of devices, yet it poses a special requirement on the planning domain: It should be modeled in a way that avoids global knowledge as much as possible. In other words, device actions should need as little as possible information about other planning operators. A naive way of modelling a smart environment is the following domain *smartenvironment-naive*:

```
(define (domain smartenvironment-naive)
  (:requirements :strips :equality :typing)
  (:predicates (Pointing ?c - Canvas ?p - Projector)
               (CrossbarIn ?n - Notebook)
               (CrossbarOut ?p - Projector)
               (DocShown ?d - Document ?c - Canvas)
               (isDown ?c - Canvas)
```

```
                (Hosts ?d - Document ?n - Notebook)
                (isMax ?d - Document ?n - Notebook)
                (Connected ?n - Notebook ?p - Projector))

(:action CanvasUp
  :parameters (?c - Canvas)
  :precondition (isDown ?c)
  :effect (not (isDown ?c)))

(:action CanvasDown
  :parameters (?c - Canvas)
  :precondition (not (isDown ?c))
  :effect (isDown ?c))

(:action MoveProjector
  :parameters (?c1 - Canvas ?c2 - Canvas)
  :precondition (and (Pointing ?c1 NEC-MT1065 )
                (not (Pointing ?c2 NEC-MT1065)))
  :effect (and (not (Pointing ?c1 NEC-MT1065))
          (Pointing ?c2 NEC-MT1065)))

(:action SwitchCrossbar
  :parameters (?n - Notebook ?p - Projector)
  :precondition (and (CrossbarIn ?n)(CrossbarOut ?p))
  :effect (forall (?x - Notebook)(when (= ?x ?n)(Connected ?x ?p)
          (not (Connected ?x ?p)))))

(:action Maximize
  :parameters (?n - Notebook ?d - Document)
  :precondition (and (Hosts ?d ?n))
  :effect (forall (?x - Document)(when (= ?x ?d)(isMax ?x ?n)
          (not (isMax ?x ?n)))))

(:action ShowDoc
  :parameters (?n - Notebook ?p - Projector ?d - Document ?c - Canvas)
  :precondition (and (Connected ?n ?p)(Pointing ?c ?p)
                (isMax ?d ?n)(isDown ?c))
  :effect (forall (?x - Document)(when (= ?x ?d)(DocShown ?d ?c)
          (not (DocShown ?d ?c))))))
```

The domain *smartenvironment-naive* models a smart meeting room containing two notebooks (NB1, NB2), two documents (Doc1, Doc2), eight canvasses (LW1, LW2, LW3, LW4, LW5, LW6, VD1, VD2), three fixed projectors that can each point to one fixed canvas (EPS3, EPS6, Panasonic), and one steerable projector (NEC-MT1065) which can point to any of the eight canvasses. This domain can be used for solving the planning problem *presentation*:

```
(define (problem presentation)
  (:domain smartenvironment-naive)
  (:objects NB1 NB2 - Notebook
```

```
            Doc1 Doc2 - Document
            LW1 LW2 LW3 LW4 LW5 LW6 VD1 VD2 - Canvas
            EPS3 EPS6 Panasonic NEC-MT1065 - Projector)
  (:init (Hosts NB1 Doc1)
         (Hosts NB2 Doc2)
         (CrossbarIn NB1)
         (CrossbarIn NB2)
         (CrossbarOut EPS3)
         (CrossbarOut EPS6)
         (CrossbarOut Panasonic)
         (CrossbarOut NEC-MT1065)
         (Pointing LW4 NEC-MT1065)
         (Pointing LW3 EPS3)
         (Pointing LW6 EPS6)
         (Pointing VD2 Panasonic))
  (:goal (and (DocShown Doc1 LW3)(DocShown Doc2 LW1))))
```

The goals of the planning problem *presentation* are to show document Doc1 on canvas LW3 and document Doc2 on canvas LW1. Using the domain description *smartenvironment-naive*, a planner could generate the following plan:

```
(CanvasDown LW1)
(CanvasDown LW3)
(Maximize NB1 Doc1)
(Maximize NB2 Doc2)
(SwitchCrossbar NB1 NEC-MT1065)
(MoveProjector LW4 LW3)
(ShowDoc NB1 Doc1 NEC-MT1065 LW3)
(SwitchCrossbar NB2 NEC-MT1065)
(MoveProjector LW3 LW1)
(ShowDoc NB2 Doc2 NEC-MT1065 LW1)
```

In this plan, the steerable projector (NEC-MT1065) is used to show both Doc1 on LW3 and Doc2 on LW1. In the real world, this is not possible, of course. A single projector cannot be used to show two documents on two canvasses simultaneously. Hence, the modelling of the domain is inadequate. Lowering a canvas, for example, is a very short action. In contrast, showing a document is an action that persists over a longer timespan. We thus call this a *persistent* action.We need to express somehow that if a projector shows a document, it is occupied. As soon as we maximize another document on the same notebook screen, connect the projector to a different computer via the video crossbar, or move the projector to another canvas, the first document is not visible anymore. Hence, the effects of the first *ShowDoc* action are not valid anymore. Thus, there is a dependency between the actions. We call this the *persistent action problem* because it applies to actions that persist as long as no other action is carried out on the same resources. This paper addresses the question how the persistent action problem can be solved. The additional challenge in smart environments is that we want to solve it with as little global information as possible. The remainder of this paper is structured as follows: In the next section we review

some approaches that try to express dependencies among actions in planning. In Sections 3 and 4 we introduce two approaches that solve the persistent action problem. The first locks resources which are being used by an action to prevent other actions from using the same resources. The second interleaves planning and execution of actions and is thus able use software agents as "guards" for actions that are being executed. In Section 5 we point out some of the similarities and differences of the two approaches and discuss their implications for the modelling and execution of device actions in smart environments before concluding the paper in Section 6. In a nutshell, this paper makes the following contributions:

- We identify the persistent action problem.
- We describe and compare two approaches that solve the persistent action problem without requiring global knowledge.
- As a by-product of our work, we provide an example domain description that shows how smart environments can be modeled in PDDL.

2 Related Work

For decades, the planning community has strived to extend the classical planning paradigm to better express dependencies among actions. In the early problem solver Hacker [15], Sussman incorporated a protection mechanism for already achieved subgoals. Hacker employs a primitive backward chaining mechanism: It first chooses an action X that fulfills a goal. As long as action X has an unfulfilled precondition, it goes on to select an action that can fulfill this precondition. This process is repeated for every action that has open preconditions. If an action Y has been selected because it can fulfill one of action X's preconditions, this means that action Y must be executed at some timepoint prior to action X. Action Y now adds an expression called a *purpose comment* to a comment repository stating that action Y fulfills a precondition for action X. This purpose comment is kept in the repository until action X is executed. Whenever another action Z is chosen during this interval, the comment repository is checked to see if action Z's effects conflict with the purpose comment. This means that action Z would be executed after action Y, but before action X, and that action Y undoes the precondition action Y has achieved for action X. Should this occur, the process backtracks to avoid the violation. This protection mechanism resembles the concept of causal links introduced by McAllester and Rosenblitt [10]. Unfortunately, both mechanisms only protect conditions shared by two subsequent actions. Hence, they do not solve the persistent action problem.

A number of researchers have incorporated linear logic into planning. Linear logic allows to handle resources: Any precondition of an action that is not in an effect of the same action is "consumed" upon execution of the action, i.e. unlike in classical planning, it is not valid anymore. This allows to easily block certain (unwanted) actions that use the same resource. Chrpa [5] demonstrates how this can be used e.g. in the BlocksWorld domain: before performing the action *(PickUp ?Box ?Slot)* the condition *(canPut ?Box ?Slot)* is true (i.e. the action is allowed). Performing *(PickUp ?Box ?Slot)* renders the condition *(noPut*

?Box ?Slot) true, i.e. it blocks the inverse action *(PutDown ?Box ?Slot)* – once picked up, the box cannot be put back into the same slot, it must be moved. In our domain, however, this concept is not applicable because we would not only have to block inverse actions, but several instantiations of the same action with different parameters. For example, to prevent that the action *(Maximize Notebook1 Document1)* is undone, we would have to block *(Maximize Notebook1 Document2)*, *(Maximize Notebook1 Document3)* and so on. This would lead to an explosion of effects. Furthermore, it requires global knowledge.

Another approach to better represent dependencies among actions is to extend classical planning with temporal logic. Weld and Etzioni [16] introduce two kinds of *safety conditions*: *dont-disturb* constraints and *restore* constraints. Dont-disturb constraints are conditions specified in the initial state of the planning problem and must not be violated by the plan at any timepoint. Restore constraints are somewhat weaker. They may be violated, but must be restored by the end of the plan. Safety conditions do not solve the persistent action problem, however. We cannot specify which conditions must not be violated in the initial state because they are not known at this point. They arise in the course of planning. A more expressive language is MITL (Metric Interval Temporal Logic) by Bacchus und Kabanza [3]. Using certain formulas one can e.g. specify that a robot should only open a door if it intends to move through the door, and that the next action after moving through the door must be to close it: If it opens the door at timepoint t, it must pass through the door at timepoint t+1 and close the door at timepoint t+2. Thus, one can specify conditions that must hold at a certain timepoint or during an interval relative to another fixed timepoint, not only relative to the initial or goal state. A similar approach is TAL (temporal action logics) by Doherty and Kvarnström [6]. Control formulas allow to specify which actions may or may not be executed if certain conditions are fulfilled. For our domain this means that e.g. if at timepoint t something is projected onto a canvas, at timepoint t+1 this canvas must not be raised and the projector must not be moved: `[t] (DocShown ?d ?c)` ∧ `(Pointing ?c ?p)` ∧ `(IsProjecting ?p)` → `[t+1] (CanvasDown ?c)` ∧ `(Pointing ?c ?p)` In other words, this forces the planner to terminate the projecting activity before moving the projector or raising the canvas. This would be expressive enough to solve the persistent action problem. However, MITL, its predecessor TLPlan [2] and TAL all suffer from one serious drawback: They need domain dependent search control knowledge (that is, the control formulas) to be able to solve practical problems. These control formulas must be written by a domain expert who has global knowledge. Hence, it is not applicable in our domain because the developers of our operators do not have global knowledge – they do not know which other operators will be present at the time of planning.

3 Planning: The Locks Approach

One possibility to solve the persistent action problem in classical planning is to introduce certain conditions which we call *locks*. During the planning process,

locks prevent chains of actions from being "destroyed" by conflicting actions that use the same resources. Consider the following domain *smartenvironment-locks*:

```
(define (domain smartenvironment-locks)
  (:requirements :strips :typing)
    (:types Notebook Document Projector Canvas - Device)
    (:predicates (isLocked ?d - Device)
                 (isActive ?d1 ?2 - Device)
                 (isConnected ?d1 ?d2 - Device)
                 (Hosts ?d - Document ?n - Notebook)
                 (isDown ?c - Canvas)
                 (CrossbarIn ?n - Notebook)
                 (CrossbarOut ?p - Projector)
                 (Pointing ?c - Canvas ?p - Projector))

  (:action CanvasUp
    :parameters (?c - Canvas)
    :precondition (and (not (isLocked ?c)) (isDown ?c))
    :effect (not (isDown ?c)))

  (:action CanvasDown
    :parameters (?c - Canvas)
    :precondition (and (not (isLocked ?c))(not (isDown ?c)))
    :effect (isDown ?c))

  (:action Maximize
    :parameters (?d - Document ?n - Notebook)
    :precondition (and (Hosts ?d ?n)(not (isLocked ?n)))
    :effect (and (isLocked ?n)(isActive ?d ?n)
            (isConnected ?d ?n)))

  (:action Unlock-Maximize
    :parameters (?d - Document ?n - Notebook)
    :precondition (and (isActive ?d ?n)(isLocked ?n))
    :effect (and (not (isActive ?d ?n))(not (isLocked ?n))
            (not (isConnected ?d ?n))))

  (:action MoveProjector
    :parameters (?c1 - Canvas ?c2 - Canvas)
    :precondition (and (Pointing ?c1 NEC-MT1065)
                  (not (Pointing ?c2 NEC-MT1065))
                  (not (isLocked ?c1)))
    :effect (and (not (Pointing ?c1 NEC-MT1065))
            (Pointing ?c2 NEC-MT1065)))

  (:action SwitchCrossbar
    :parameters (?n - Notebook ?p - Projector ?d - Document)
    :precondition (and (not (isLocked ?p))(isActive ?d ?n)
                  (CrossbarIn ?n)(CrossbarOut ?p))
    :effect (and (isLocked ?p)(not (isActive ?d ?n))
```

```
            (isActive ?d ?p)(isConnected ?n ?p)))

  (:action Unlock-SwitchCrossbar
    :parameters (?n - Notebook ?p - Projector ?d - Document)
    :precondition (and (isLocked ?p)(isActive ?d ?p)
                  (isConnected ?n ?p))
    :effect (and (not (isLocked ?p))(isActive ?d ?n)
            (not (isActive ?d ?p))(not (isConnected ?n ?p))))

  (:action ShowDoc
    :parameters (?p - Projector ?c - Canvas ?d - Document)
    :precondition (and (not (isLocked ?c))(isActive ?d ?p)
                  (isDown ?c)(Pointing ?c ?p))
    :effect (and (isLocked ?c)(not (isActive ?d ?p))
            (isActive ?d ?c)(isConnected ?p ?c)))

  (:action Unlock-ShowDoc
    :parameters (?p - Projector ?c - Canvas ?d - Document)
    :precondition (and (isLocked ?c)(isActive ?d ?c)
                  (isConnected ?p ?c))
    :effect (and (not (isLocked ?c))(isActive ?d ?p)
            (not (isActive ?d ?c))(not (isConnected ?p ?c)))))
```

We omit the problem description here as is the same as in the problem *presentation* described in Section 1, except for the goal statement, which is now `(:goal (and (isActive Doc1 LW3)(isActive Doc2 LW1)))`

Three locks are required for every persistent action: The first one locks the resource in question *(isLocked ?d)* such that no other action can use this resource. Thus, the set of locked resources states which resources are currently parts of chains of persistent actions. The second lock *(isConnected ?d1 ?d2)* states which two resources are used consecutively in a chain of actions. This is important if a new goal is to be fulfilled and this requires that an action sequence previously generated must be unlocked. We will get back to this in Section 5. The third lock *(isActive ?d1 ?d2)* always denotes the current end of the chain (the *tail*). During the planning process, this lock is propagated through the action sequence. Consider e.g. the *ShowDoc* operator: It has *(isActive ?d ?p)* as a precondition. Its effects include *(not (isActive ?d ?p))* and *(isActive ?d ?c)*. I.e., when *ShowDoc* is selected, the tail moves from *(isActive ?d ?p)* to *(isActive ?d ?c)*. Because every persistent action (apart from *Maximize* which is the head of the chain) has such a lock as a precondition, the chain can only be manipulated at its tail. Hence, the chain of actions cannot unintentionally be "destroyed" by a conflicting action. With this domain description, planners generate correct action sequences like the following:

```
(CanvasDown LW1)
(Maximize Doc2 NB2)
(MoveProjector LW4 LW1)
(CanvasDown LW3)
(Maximize Doc1 NB1)
```

```
(SwitchCrossbar NB2 NEC-MT1065 Doc2)
(SwitchCrossbar NB1 EPS3 Doc1)
(ShowDoc NEC-MT1065 LW1 Doc2)
(ShowDoc EPS3 LW3 Doc1)
```

Note that we added a corresponding *unlock* operator for every operator that locks a resource. This enables the planner to unlock the chain starting at its end if new goals are to be fulfilled.

4 Action Selection: The Guarding Approach

The persistent action problem can also be solved if we do not employ planning, but another approach that draws its principle from nature: action selection. In contrast to planning, action selection does not construct an explicit plan, but at every timepoint selects the action that is most likely to lead towards an open goal. This selection is based on the current world state and the goals to be fulfilled. The selected action is then executed right away. This resembles the way animals decide what to do next. Well-known action selection algorithms are those by Brooks [4] and Maes [9].

Particularly Maes' algorithm is feasible for smart environments as it can easily be distributed [13]: Each device carries descriptions of its possible actions. Each of those action descriptions is assigned a software agent that communicates with the agents of other actions. It takes part in the action selection and keeps track of the world state. The agents of all devices in an ensemble form a network at runtime. This network can easily be adapted if devices join or leave the ensemble, and it requires no central controlling component.

Action selection interleaves planning with execution: Every action selection step is followed by an execution step which changes the world state. This makes it possible to solve the persistent action problem in a fundamentally different way: One can employ the agent of a persistent action A as a "guard" for that action. Guarding means that as long as A is active, A's agent monitors whether any effect of a subsequent action B that is executed is the opposite of one of A's preconditions. In this case, it sends a message to all other agents stating that A is not executed anymore and its effects become false. Should one of A's effects be a precondition of another persistent action C which is currently executed, this process continues: C's agent will notice that C is not executed anymore, etc.

As an example, reconsider the erroneous action sequence generated by the planner in Section 1. This action sequence cannot be generated if we use the guarding approach: Consider the point in the action sequence when the action *(SwitchCrossbar NB2 NEC-MT1065)* is executed. This renders the condition *(Connected NB1 NEC-MT1065)* false. One precondition for the action *(ShowDoc NB1 Doc1 NEC-MT1065 LW3)*, which is currently active, requires the same condition to be true. This is noticed by this action's agent. It notifies the rest of the ensemble that the action is not active anymore and its effects become false. The goal *(DocShown Doc1 LW3)* is now open again and can be fulfilled once more. Hence, the action selection algorithm cannot generate an action sequence

that uses the steerable projector NEC-MT1065 to show Doc1 on LW3 and Doc2 on LW1 simultaneously. This means the world state in the model of the world does not become inconsistent to the actual world state in the real world. Thus, the guarding approach enables us to model the domain as in Section 1.

5 Comparing the Two Paradigms

In Sections 3 and 4 we introduced two paradigms which both solve the persistent action problem. In this section, we elaborate in more detail on the similarities and conceptual differences between the two paradigms. We also point out some of the implications those differences have on modelling and execution of devices' actions in smart environments.

Both paradigms have in common that actions can be modelled without global knowledge. Each action description can be written using only knowledge about conditions that must be fulfilled before the action can be executed and conditions that will be true after the action is executed. Of course, it is preferable that the developer of an action description has an idea e.g. which other actions might rely on the effects of the action. This guides the developer's decision which effects to consider for inclusion in the action description. However, the developer need not have knowledge about the complete domain.

One difference between the two paradigms is the cognitive model they resemble: The locks paradigm can be described with the concept of data flow. In the example action sequence shown in Section 3, the following *isActive* locks become active one after another:

```
(isActive Doc2 NB2)
(isActive Doc1 NB1)
(isActive Doc2 NEC-MT1065)
(isActive Doc1 EPS3)
(isActive Doc2 LW1)
(isActive Doc1 LW3)
```

Doc1 can be seen as data flowing from a source (notebook NB1) over an intermediate station (projector EPS3) to a sink (canvas LW3). Likewise for Doc2. In contrast, the agents in the guarding approach resemble the concept of guards that are positioned along a line to the goal, each monitoring whether its assigned persistent action is still being executed. This comes along with a fundamental difference in the approach to solving the persistent action problem: In the locks approach, we prevent conflicting actions from being executed. Thus, the developer has to be careful to add the appropriate locks to the action descriptions of persistent actions. Furthermore, for every operator describing a persistent action, a corresponding unlock operator must be added. This also implies that the overall number of operators is higher. In the guarding approach, on the other hand, we do not prevent conflicting actions from being selected. Instead, for every action A, we monitor whether a conflicting action B terminates

A's execution. The developer of an operator for a persistent action must only mark the action as a persistent action. The rest is managed by the action selection algorithm.

Another difference manifests itself if an action sequence has been generated and a new goal is to be fulfilled. Consider the point where the action sequence in Section 3 has been generated and the following new goal arises (the former goals are now not valid anymore): `(:goal (isActive Doc1 LW1))`.

This requires the locks approach to execute a number of unlock actions before the new goal can be fulfilled. A planner generates an action sequence such as:

```
(Unlock-ShowDoc NEC-MT1065 LW1 Doc2)
(Unlock-ShowDoc EPS3 LW3 Doc1)
(Unlock-ShowDoc NB2 NEC-MT1065 Doc2)
(Unlock-ShowDoc NB1 EPS3 Doc1)
(SwitchCrossbar NB1 NEC-MT1065 Doc1)
(ShowDoc NEC-MT1065 LW1 DOC1)
```

The existing chain of actions has to be unlocked backwards to the point where necessary actions to fulfill the new goal (*SwitchCrossbar*, *ShowDoc*) can be executed. This is the kind of scenario we need the *(isConnected ?d1 ?d2)* lock for: without it, the planner could not figure out which predecessor an action has and would thus not be able to unlock an existing sequence correctly. The guarding approach does not perform any unlock action in order to fulfill the new goal, it just executes *SwitchCrossbar* and *ShowDoc*. This is both a blessing and a curse. On the one hand, of course, less actions have to be executed. On the other hand, existing action sequences can unintentionally be "destroyed" by conflicting actions because there is no mechanism to protect them.

6 Conclusion

In this paper, we have motivated that it is feasible to model device actions in smart environments as planning operators. We have then introduced the *persistent action problem*: If a persistent action is active and another action is being executed that uses the same resources, the effects of the first action become invalid. We have suggested two ways to solve this problem: In planning, one can model actions using locks, thus explicitly preventing conflicting actions from being executed. Another approach which is applicable if one uses an action selection mechanism instead of planning is to assign an agent to each persistent action which monitors if any conflicting action destroys the preconditions of the persistent action when it is active. We have furthermore compared the characteristics of the two paradigms. Both have their benefits and shortcomings, yet both are suited for modelling device actions in smart environments because each has two key features: First, they solve the persistent action problem, and second, both allow to model the domain without global knowledge. As we have pointed out, the second feature is extremely important in highly dynamic environments.

Acknowledgement

Christiane Plociennik, Christoph Burghardt and Florian Marquardt are supported by a grant of the German National Research Foundation (DFG), Graduate School 1424 (MuSAMA).

References

1. Amigoni, F., Gatti, N., Pinciroli, C., Roveri, M.: What planner for ambient intelligent applications? IEEE Transactions on Systems, Man and Cybernetics - Part A 35(1), 7–21 (2005)
2. Bacchus, F., Kabanza, F.: Using Temporal Logic to Control Search in a Forward Chaining Planner. In: Proc. EWSP, pp. 141–153. Press (1995)
3. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence 22(1-2), 5–27 (1998)
4. Brooks, R.A.: A robust layered control system for a mobile robot. In: Artificial intelligence at MIT: expanding frontiers, pp. 2–27 (1990)
5. Chrpa, L.: Linear Logic in Planning. In: Proceedings of Doctoral Consortium of ICAPS, pp. 26–29 (2006)
6. Doherty, P., Kvarnström, J.: TALplanner: A Temporal Logic-Based Planner. AI Magazine 22(3) (2001)
7. Heider, T., Kirste, T.: Supporting goal based interaction with dynamic intelligent environments. In: Proc. ECAI, pp. 596–600 (2002)
8. Issarny, V., Sacchetti, D., Tartanoglu, F., Sailhan, F., Chibout, R., Levy, N., Talamona, A.: Developing ambient intelligence systems: A solution based on web services. Automated Software Engg. 12(1), 101–137 (2005)
9. Maes, P.: Situated Agents Can Have Goals. In: Maes, P. (ed.) Designing Autonomous Agents, pp. 49–70. MIT Press, Cambridge (1990)
10. McAllester, D., Rosenblitt, D.: Systematic Nonlinear Planning. In: Proceedings of the Ninth National Conference on Artificial Intelligence, pp. 634–639 (1991)
11. McDermott, D.: PDDL – The Planning Domain Definition Language. Draft (1998)
12. Mozer, M.C.: Lessons from an adaptive home. Smart Environments: Technology, Protocols, and Applications, 273–298 (2005)
13. Reisse, C., Kirste, T.: A Distributed Action Selection Mechanism for Device Cooperation in Smart Environments. In: Proc. IE (2008)
14. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice-Hall, Englewood Cliffs (2003)
15. Sussman, G.J.: A Computational Model of Skill Acquisition. Technical report, Cambridge, MA, USA (1973)
16. Weld, D., Etzioni, O.: The first law of robotics (a call to arms). In: Proc. AAAI, pp. 1042–1047 (1994)