Project Work

A Case-Study of Active Logic

---

submitted by
Thorben Ole Heins

supervised by
Jacek Malec

Department of Computer Science, Artificial Intelligence
Lund University
Sweden

Abstract

Active logic is an approach to overcome some limits which hinder classical logic on being sufficiently expressive for artificial intelligence agents in real world scenarios.
With the recent implementation of an automatic theorem prover in active logics an important step into the direction of actual usage of active logics in the real world was made.
The purpose of this work is to develop a non-trivial real-world scenario in order to test it using the prover. In order to do this we will give a short introduction into active logics and the theorem prover, before we specify a scenario and report on our experiments.

Contents

1 Introduction **3**
- 1.1 Problem . . . 3
- 1.2 Structure of this report . . . 4

2 Prerequisites **5**
- 2.1 Active Logic . . . 5
- 2.2 Automatic Theorem Prover in Active Logic . . . 6

3 Scenario & Architecture **7**
- 3.1 The test scenario . . . 7
  - 3.1.1 The Robot . . . 8
  - 3.1.2 The Building . . . 8
  - 3.1.3 Critical Scenario . . . 8
- 3.2 Architecture . . . 10
  - 3.2.1 Knowledge Base . . . 10
  - 3.2.2 Components . . . 12

4 Transferring the Problem to First-Order Logic **15**
- 4.1 Formalization in First-Order Logic . . . 15
  - 4.1.1 Basic formulae . . . 17
  - 4.1.2 Action formulae . . . 17
  - 4.1.3 Geometry . . . 18
  - 4.1.4 Verification . . . 18
- 4.2 Results . . . 18

5 Deploying the scenario to the architecture **20**
- 5.1 Formalization in Active Logic . . . 20
  - 5.1.1 Axioms . . . 20
  - 5.1.2 Observations . . . 21
- 5.2 Results . . . 21

6 Conclusion and Future Work **23**
- 6.1 Conclusion . . . 23
- 6.2 Future Work . . . 23

A Geometry of the Building in FOL **25**

B Prolog First-Order Logic Model Verifier **27**

C Active Logic Formalization **28**

D Perl Parser **29**

Chapter 1

Introduction

In times of steadily growing technization of the society, the development of robotic agents for the daily life will become more important. In order for these agents to perform the tasks in a way that is acceptable for humans, that rely on their performance, the agents have to be aware of time. This is because when interacting with humans, there might be critical tasks to be performed until a certain point in time, so called deadlines.
As the real world is not a constant structure there might also occur contradictions for the agent by observing its environment. An agent's knowledge needs not (and usually will not) be perfect: agent's current beliefs may contradict with the actual observable state of the world.
For about twenty years now, the computer science community has worked on a logic, which can help to support the development of time-aware agents, that could reason about time and deadlines and also handle contradictions. This logic is referred to as Active Logic. In his master's thesis Julien Delnaye developed a theorem prover at Lund University, which used active logics in three smaller examples. This theorem prover is the only available implementation in this project that gives the opportunity to figure out how active logic behaves dealing with real-world problems.
In this work we develop a bigger scenario and then try to implement it in the theorem prover mentioned before. The purpose of this is to try to show whether active logic, as implemented in the theorem prover, can be used. If this succeeds we also might show whether active logic are able to handle real life demands, as mentioned earlier.

1.1 Problem

During the last decades there was an ongoing research on several approaches, trying to overcome the limitations of classical logic formalisms regarding real-world agents. Active logic is one of these approaches, which also stands in the focus of this work. The automatic theorem prover, which was recently developed for active logic, is one opportunity to see active logic and its computational performance. As until now there were only some rather trivial tasks performed by the prover, in this work we will try to build up a new and more complex test scenario, which can be applied in the prover to test its performance on a

realistic scenario.
We therefore develop this scenario, taking into account the contradiction handling and time-awareness features of active logic. Then an architecture is developed, in which the prover can be used as active logic device in this scenario. In order to do this several steps have to be made before. At first the scenario has to be formulated in predicate logics. Then it has to be verified regarding its logical consistency. When this is done it has to be transferred to the active logic theorem prover language. Just when all this is done, the architecture mentioned earlier can be tested, meaning that the performance of the theorem prover can be analyzed.
Additionally we create a simple application that can be used transform the output of the theorem prover into a human readable form. This is very useful when it comes to analyzing the results of the reasoning process.

1.2 Structure of this report

This work is structured as follows:

- Chapter 1: contains this introduction.
- Chapter 2: contains a short introduction into active logics and the automatic theorem prover.
- Chapter 3: at first the test scenario is specified in natural language. Afterwards the architecture that is created to implement the scenario with active logic is presented.
- Chapter 4: the test scenario is transferred to first order logic and afterwards tested in basic prolog in order to verify its logical consistency.
- Chapter 5: we try to implement the architecture with the theorem prover as active logic reasoner.
- Chapter 6: conclusions & future work.
- Appendix A: First-Order Logic Geometrical model of a building.
- Appendix B: Prolog implementation of the scenario and documentation.
- Appendix C: Active Logic implementation of the scenario.
- Appendix D: Perl parser for prover output and the “two wise man problem”, making the prover output human readable.

Chapter 2

Prerequisites

In order to understand the next steps, some concepts have to be introduced. In this chapter we will explain the general ideas of active logic. Further we will present the automatic theorem prover for active logic.

2.1 Active Logic

In this section we present a general overview about active logic. As this work is not intended to discuss the theoretic cornerstones of active logic, we skip explaining active logic on a theoretic basis. We rather give a short overview of this field of research, so that the reader can understand why active logic is so useful and why we used it in this work. Theoretical discussions of active logic and its predecessor step-logic can be found in [6, 9, 2, 8].
Active logic in contrast to classical logic has some interesting features that enable it to model real-world problems. One thing is the fact that with active logic one can not only reason about time like other logics allow, but active logic allows to reason *in* time. We will refer to this fact as time-awareness in the following. It means that while reasoning, the reasoner can take the reasoning process and the time it takes into account. Additionally the reasoning is viewed as an ongoing process. This allows that the set of formulae that the reasoner considers can change from one time step to another, which also means that the set of beliefs might shrink over time.
Contradiction handling is another important feature of active logic, that is not present in classical logic. Active logic is able to handle contradiction because it has meta-reasoning abilities. This means, that during the reasoning process, the reasoning is "stopped" so that it can be decided what to do next [1]. If two contradicting formulae P and $\neg P$ are in the knowledge base at the same time, active logic can detect them, and mark them in some way, so that the knowledge base will remain consistent, in spite of the contradictive formulae. Later on one or both should be removed from the knowledge base.
Given these two features we now can think of real-world situations where they might be useful. The time-awareness naturally becomes important when we think of resource-bounded agents that perform tasks. For example a robot that has to perform several tasks, but does not know whether its battery power will suffice. If this robot then could reason with active logic about this question, it

would need to take into account the time it needs to reason about the problem, so it can estimate whether all the tasks plus the reasoning can be performed with the given battery level. This kind of reasoning is also referred to as reasoning about approaching deadlines.
There can also be found an example where contradiction handling is important. If a robot has to clean each room in a corridor, then it will somewhere have the knowledge that the ways between the rooms are free, so the robot can pass them. This might be represented e.g. by $free(wayX)$. If now during the robot's performance one of its sensors detects that one of the ways between two rooms is blocked, the knowledge, that this particular way is blocked, will be added to the knowledgebase (e.g. $\neg free(way6)$). This obviously leads to a contradiction because now $free(way6)$ and $\neg free(way6)$ are in the knowledgebase. In classical logic this would lead to any conclusion - an apparently undesirable result.

2.2 Automatic Theorem Prover in Active Logic

In order to test the behavior of active logic regarding its computational performance, Julien Delnaye has developed an automatic theorem prover in active logic (ATP) in his master's thesis [5]. The ATP is able to handle scenarios formulated in step-logic as well as scenarios formulated in LDS (Labelled Deductive Systems). These are two different kinds of active logic.
Step-logic was developed by Jennifer Elgot-Drapkin in her PhD thesis (see [6]). There are seven different kinds of step-logics and only the most expressive one is used in the ATP. Anyhow, step-logic is an oversimplification of the memory model that is used in active logic [5]. It can therefore lead to computational issues, because the set of beliefs continuously grows. As the memory of agents is naturally limited in size, this is a major issue. It also leads to a time problem, as the reasoning will naturally take longer after each reasoning step.
In order to overcome those problems, Michael Asker developed another kind of active logic based on Gabbay's *Labelled Deductive Systems* in his thesis. By using this formalism to model active logic and its memory model, the size of the belief set can be limited to a constant size, which will not be exceeded at any time.
The ATP can be used by providing three input sets respectively four sets when the LDS approach is used.
The first set consists of the inference rules that ATP will use to reason about its knowledge. When the LDS approach is used, two different kinds of inference rules are needed, which has its origin in the way the LDS handles the belief set. Then there is a set of axioms, which is used to tell the ATP what the world is like. Here things like "If you use the vacuum cleaner for one minute in room A, then in the next time step room A is clean" can be stated. The last set is the set of observations, which provides the sensory feedback that the agent using the ATP, provides. Here things like "I see that way 6 is blocked now" or "I am now at point 7 in the building" are told to the ATP.

Chapter 3

Scenario & Architecture

In this chapter we first specify a test scenario that is going to be tested in the ATP. After that the architecture of the system that is to be implemented is presented.

3.1 The test scenario

When creating a test scenario, that uses the advantages of active logic, one has to include time-awareness and contradiction handling. In the literature one can find several interesting scenarios. One of these is the story from "Seven days in the Life of a Robotic Agent" [4] by Chong et al., which was an inspiration while creating the scenario for this work.
Like in that story, in our scenario we have a robotic agent which has to perform tasks throughout the day. The robot works in a four story office building which is depicted in figures 3.1 and 3.2. We have created a schedule S which has to be performed by the robot every day, which can be seen in table 3.1. The schedule is rather imprecise: it is just a rough sketch of what the robot should do. When we want to let a real robot have this work performed it will naturally have to be much more detailed and expressive. We create such a version of the schedule in chapter 4. For now this kind of description is enough to get a first impression.

Table 3.1: Schedule S that has to be performed by the robot every day.

Time interval (hrs)	Task
0-3	clean the hallways
3-7	charge battery
6-7	load mail to be delivered
7-10	deliver & collect mail
10-11	load food to be delivered
11-14	deliver food
14-17	collect dishes
17-18	undergo maintenance
18-0	charge battery

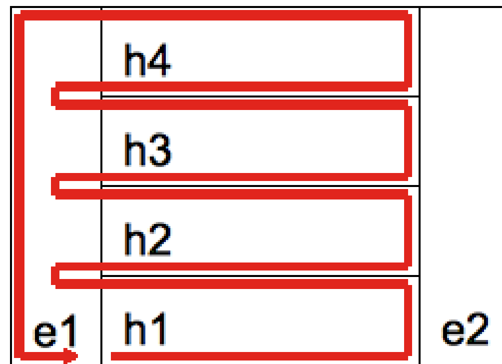

Figure 3.1: Four story building, in which the robot is performing its tasks. The bold line depicts the robots standard way through the building, elevator $e2$ is only used when elevator $e1$ cannot be used.

3.1.1 The Robot

The robot has to have certain sensors, which let it recognize obstacles, which block his way. It also has interfaces, which enable it to be packed with food, mail and dishes. As cleaning has to be performed, it also has to have cleaning facilities installed, e.g. a vacuum cleaner.
It is important to note that we did not use a real robot to perform the tasks while working on this project, as it was not the focus of this work. We rather built up a software, which acts like a robot environment would: it emulates a robotic environment for the purpose of analyzing the robots reasoning process.

3.1.2 The Building

The building has four stories. Each of the stories has only one hallway, called $h1$ to $h4$. On each corridor there are six rooms called $p2$, $p3$, $p5$, $p6$, $p8$ and $p9$. To enable robot movement between the hallways, the building has two elevators $e1$ and $e2$, which can both be used by the robot. In the normal case only $e1$ is used. The geometrical structure of the building (together with the location names used later) is depicted in Figure 3.2.

3.1.3 Critical Scenario

As the food delivery is crucial, the robot has to be able to determine times and deadlines. When everything is working as it is supposed to do, the robot will perform its tasks one after another and in the end of the day everything will be done and the robot will begin start all over at midnight. In such cases the ATP as active logic component does not need to perform any work in this project.
However, if there occur problems during the performance of the work, the ATP will have to perform some work. Such a case occurs, for example when the robot is currently delivering and collecting mail. Assume it wants to go from the third to the fourth hallway at 8:58 with one of the elevators but its sensors tells it, that the way is blocked.

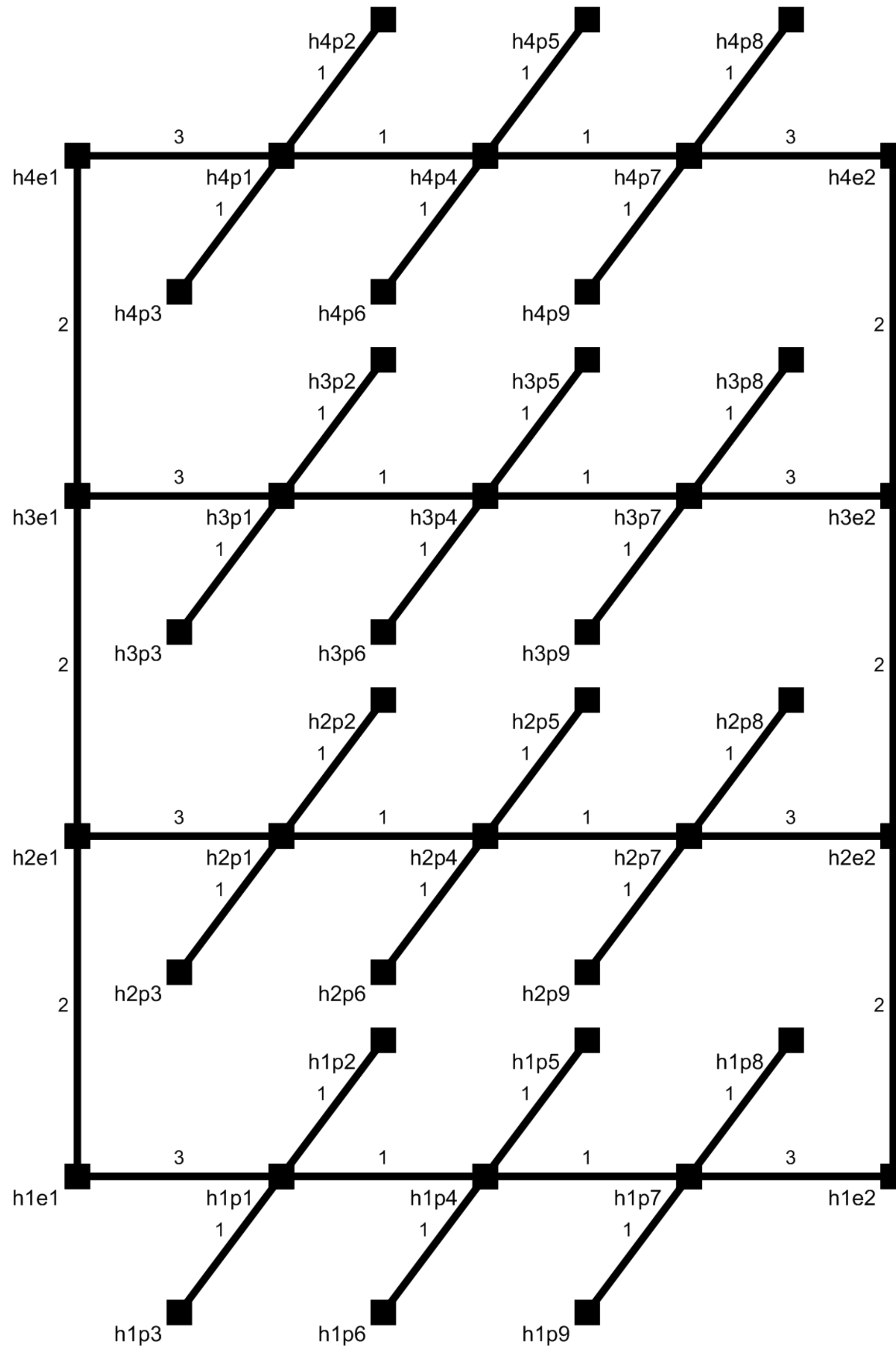

Figure 3.2: Grid representation of the four story building as the robot “sees” it.

This is the moment where active logic comes into play. The next actions have to be reconsidered or replanned in some way. Questions like these arise: Can the robot deliver and collect the mail to the fourth hallway $h4$ in time? Will it also be able to meet the other upcoming deadlines? Is food delivery more important than mail delivery? How long does it have to reason to find an alternative that satisfies the requirements? Is there such an alternative at all?
In the following we describe our approach of how to solve this problem and how to answer these questions.

3.2 Architecture

In order to be able to use some of the capabilities of active logic we decided to create an architecture that has three independent components which are connected by one Central Coordination Component (CCC):

- Planner
- Plan Verifier
- Automatic Theorem Prover in Active Logic (ATP)

It also handles the knowledge base which consists of four separate parts:

- Axioms
- Facts
- Plans
- Requirements

In the following we will have a closer look on how all these elements are related to each other and show where active logic actually will be of use for us. At first we will describe what the knowledge base contains, and which part is used where in the reasoning process.

3.2.1 Knowledge Base

The knowledge base is separated into four parts. It is not one central entity but rather is distributed all over the architecture, as not every component need to have knowledge of everything.

Axioms

There are three different kinds of axioms, which are: world model, robot action and logical laws. The world model axioms are those, which describe the static rules of the world. Things like "If there is no dust at one place it is clean" are captured here. With the robot action axioms the behavior of the robot and the side effects for the environment are expressed. Here one can find things like "When the robot is at one place and cleans it then afterwards this place is clean and it takes a certain amount of time and battery level to perform this action". These two kinds of axioms are very application-dependent, while the logical laws, including things like contradiction handling, belief inheritance and forgetting of beliefs, can be used in other scenarios, too.

Facts

Facts are the observations which are available to the active logic theorem prover. They represent basic percepts like the fact that the robot is at a certain position at some point in time or that mail has been delivered to a certain position in the building (or not). It can be viewed as a stream that corresponds to the time flow. Also the geometry of the building will be stored in the observations at time zero. This means that not only real observations from the robot is stored here, but also all the basic knowledge about the environment. Observations at time zero and world model axioms are often interchangeable.

Plans

A plan is a sequence of the tasks to be performed by the robot during the day. It is completely handled by the coordinator and shrinks as the time progresses. If there are contradictions detected in the output of the active logic theorem prover there is a planning process initiated by the coordinator, which returns a new plan that then has to be verified by the plan verifier.

Requirements

The requirements contain the deadlines that have to be reached. They may also be stored in the initial observations of the ATP. The requirements never change. There are soft and hard deadlines that have to be reached.

Relation between Plans & Requirements

As said before the requirements contain the deadlines for the tasks that the robot has to perform. When the coordinator at some point in time has to replan the robot actions, the plan verifier crosschecks the new plan with the deadlines in the requirements. So if there are deadlines that cannot be met, the coordinator initiates another replanning. In figure 3.3 this process is depicted.

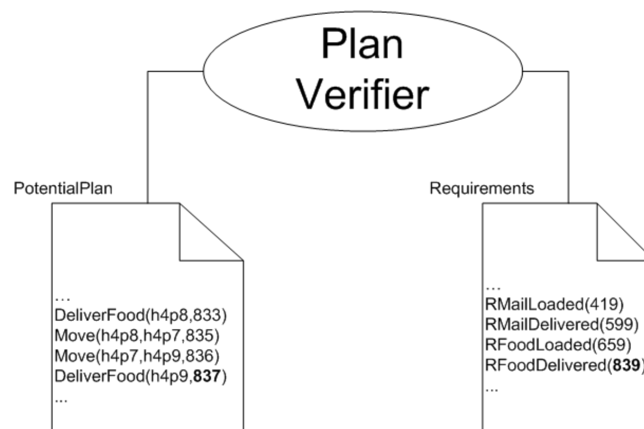

Figure 3.3: The plan verifier compares the potential plan to the static requirements and then determines whether the potential plan is ok. The one in this picture is ok, at least according to its food delivery.

3.2.2 Components

Planner

In the scope of this work no planning is performed directly, but the plans that are needed in the scenario are provided by an external module. Generally, the planner is the component that creates the plans for the robot's work. So its tasks are to create the initial plan as well as any replanning needed. Any suitable planner can be used [7].

Plan Verifier

After the plans are created they have to be verified by the plan verifier. Verification in this case is checking whether a plan meets the requirements. So the plan is executed virtually in classical logic in order to find out whether the requirements would be met in a real execution. If a plan can be positively verified it will afterwards be executed by the robot and the facts that represent the plan in the active logic theorem prover will be updated accordingly.

Automatic Theorem Prover

The ATP is the only component that actually handles active logic. In this project it is only used to handle contradiction. It is important to note, that the architecture allows to reason about deadlines as well, even if it was not done in this scenario.
Contradictions can occur when the robot's sensors deliver an observation that contradicts the beliefs that are currently in the theorem prover. One example for this is the case when the robot tries to move from one point a to another point b and it finds out that the way is blocked. Then the initially assumed fact $\neg Blocked(a, b)$ as well as the sensed fact $Blocked(a, b)$ lead to a contradiction, which will then trigger the system to replan the actions of the robot.
In general there are two cases in which replanning can occur: when the plan verifier does not verify the given plan replanning has to take place in order to meet the requirements. Also when the robot is currently executing a plan and then a contradiction is detected by the active logic theorem prover, replanning has to be done.

Central Coordination Component

The workflow between the three components mentioned before is handled in this module. Figure 3.4 gives an overview of the workflow.
In the case of a contradiction, the ATP detects it and creates a new predicate in its knowledge base. This predicate is then detected by the CCC, which initiates a replanning using the planner. The new plan is then verified before the robot executes it.
There are then three possible results of planning. It can be the case that there exists an alternative plan, which then has to be verified. It also can occur that it only exists a plan that only fulfills all hard requirements and not all soft requirements. In this case it might be useful to incorporate the actual reasoning time, but until now we are not prepared to do this. In the next paragraph we

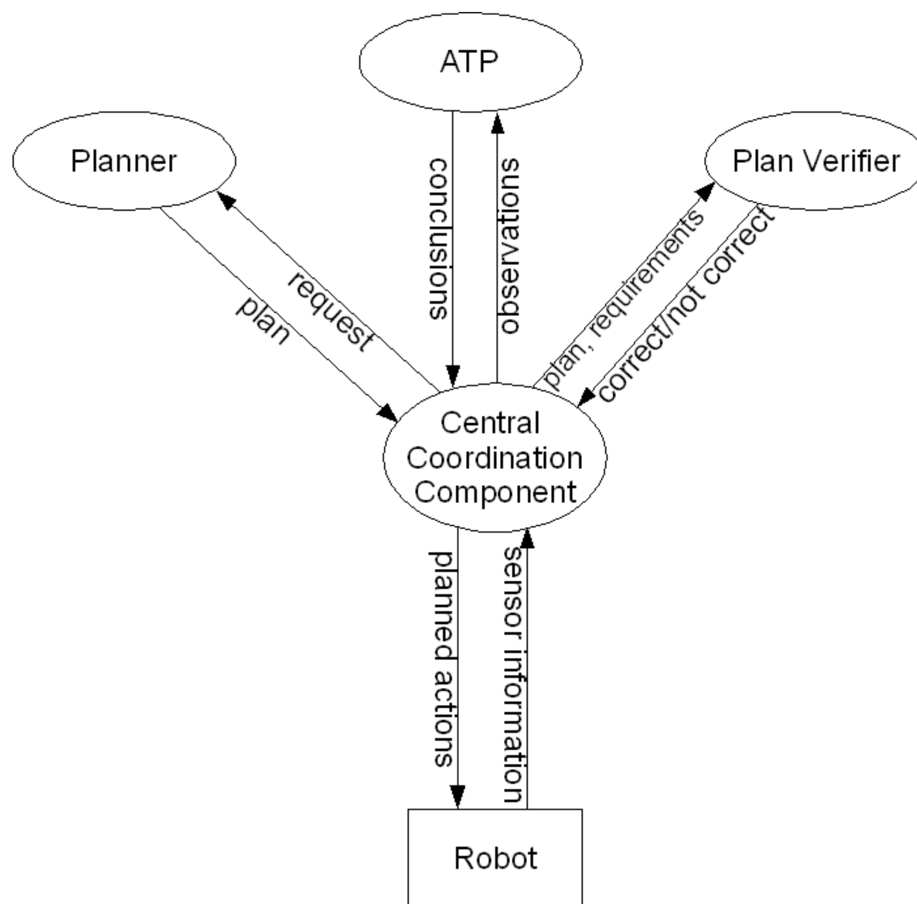

Figure 3.4: The architecture

give the reason for that. The third case is the obvious one, when no alternative plan exists. This has to lead to interaction with the robot operator.
This architecture takes away a lot of work from active logic. For example it would be very interesting to see how active logic would take into account the reasoning time itself when it comes to replanning. This would mean that all three components would have to be implemented in active logic, which is possible, but due to a limited project runtime, was not done in this work.
Another possibility would be to have some temporal characterization in active logic of an external planning module, so the reasoning about reasoning would still be possible within the active logic framework.

Chapter 4

Transferring the Problem to First-Order Logic

In the ATP first-order logic is used as the underlying language language. So in order to be able use the ATP for the robot's day scenario it is first of all formulated in first-order logic. The model that is created during this process is then implemented in standard Prolog in order to verify its logical consistency.

4.1 Formalization in First-Order Logic

In this section we introduce the formulae that are needed to model the robot's day scenario in first-order logic. At first we give a short overview of the predicates in table 4.1 below. Then we present the scenario captured in first order logic by the formulae 4.1 to 4.12.

Table 4.1: List of predicates used to model the scenario in first-order logic.

Predicate	Description
$AtRobot(l, t)$	Gives the location l of the robot at time t
$BatteryLevel(lvl, t)$	Battery level lvl of the robot at time t
$Blocked(l_1, l_2)$	The $GridEdge(l_1, l_2, t)$ between point l_1 an point l_2 in the building is blocked. The negation is the default case (see formula 4.2).
$CollectDishes(l, t)$	Represents the robot action of collecting dishes at location l at time t
$DeliverAndCollectMail(l, t)$	Represents the robot action of delivering and collecting mail at location l at time t
$DeliverFood(l, t)$	Represents the robot action of delivering food at location l at time t
	Continued on next page

Table 4.1 – continued from previous page

Predicate	Description
$Dirty(l,t)$	Indicates whether a location l is dirty at time t. By default all locations are dirty (see formula 4.1).
$DishesCollected(l,t)$	Indicates that the dishes have been collected at location l at time t
$DoCharge(t_1,t_2)$	Represents the charging of the robot's battery between time t_1 and time t_2
$DoClean(l,t)$	Represents the cleaning of location l at time t
$FoodAt(l,t)$	Indicates that food has been delivered to location l at time t
$FoodLoaded(t)$	Indicates that the food has been loaded on the robot at time t
$GridEdge(l_1,l_2,t)$	Represents an edge between the locations l_1 and l_2 in the house grid (see figure 3.2). Time t indicates how long it takes to move between those locations.
$GridPoint(l)$	Indicates that location l is a point in the grid depicted in figure 3.2
$Length(path)$	Is used as a function that returns the length of a $path$ provided. The length is here not a geometrical length but rather the time that is consumed moving along the $path$. This function has to be implemented separately.
$LoadFood(l_1,l_2)$	Represents the action of loading food on the robot between time t_1 and time t_2
$LoadMail(t_1,t_2)$	Represents the action of loading mail on the robot between time t_1 and time t_2
$MailAt(l,t)$	Indicates that mail has been delivered to location l at time t
$MailCollected(l,t)$	Indicates that mail has been collected at location l at time t
$MailLoaded(t)$	Indicates that the mail has been loaded on the robot at time t
$Move(l_1,l_2,t)$	Represents the action of the robot moving from location l_1 to l_2 at time t. This is only possible over edges. For moving over multiple edges $MoveAlongPath(path = [l_1,...,l_n])$ is used.
$MoveAlongPath(path = [l_1,...,l_n],t)$	Represents the action at timepoint t of the robot moving along a $path$, being represented as a set of $GridPoint$s.
$StationaryRobot(l,t_1,t_2)$	Indicates that the robot is at location l between the times t_1 and t_2

4.1.1 Basic formulae

$$\forall l(Dirty(l,0) \wedge \neg FoodAt(l,0) \wedge DishesCollected(l,0) \wedge \neg MailCollected(l,0) \wedge \neg MailAt(l,0) \wedge BatteryLevel(1000,0)) \quad (4.1)$$

$$\forall l_1, l_2, t GridEdge(l_1,l_2,t) \rightarrow \neg Blocked(l_1,l_2) \quad (4.2)$$

$$StationaryRobot(l,t_1,t_2) \rightarrow \forall t, t >= t_1, t <= t_2(AtRobot(l,t)) \quad (4.3)$$

In formula 4.1 the observations at time zero are set. Also formula 4.2 sets some initial knowledge, that all the $GridEdge$s are not blocked. Formula 4.3 is a predicate that just takes care that the robot is at a certain location l between two timepoints t_1 and t_2.

4.1.2 Action formulae

In the following the formulae are presented, that describe how the robot actions look like and what side effects they have. All free variables are assumed to be universally quantified.

$$AtRobot(l_1,t) \wedge Move(l_1,l_2,t) \wedge \exists u GridEdge(l_1,l_2,u) \wedge BatteryLevel(lvl,t) \rightarrow AtRobot(l_2,t+u) \wedge BatteryLevel(lvl-u,t+u) \quad (4.4)$$

$$AtRobot(l_1,t) \wedge MoveAlongPath(path=[l_1,l_2,...,l_n],t) \wedge \exists u GridEdge(l_1,l_2,u) \wedge Length(path) > 0 \rightarrow AtRobot(l_2,t+u) \wedge Move(l_1,l_2,t) \wedge MoveAlongPath([l_2,...,l_n],t+u) \quad (4.5)$$

$$AtRobot(l,t) \wedge Dirty(l,t) \wedge BatteryLevel(lvl,t) \wedge DoClean(l,t) \rightarrow \neg Dirty(l,t+1) \wedge AtRobot(l,t+1) \wedge BatteryLevel(lvl-1,t+1) \quad (4.6)$$

$$BatteryLevel(lvl,t_1) \wedge StationaryRobot(h1p3,t_1,t_2) \wedge DoCharge(t_1,t_2) \rightarrow BatteryLevel(lvl+((t_2-t_1)*10)) \quad (4.7)$$

$$StationaryRobot(h1p3,t_1,t_2) \wedge t_2 > t_1+20 \wedge LoadMail(t_1,t_2) \rightarrow MailLoaded(t_2) \quad (4.8)$$

$$AtRobot(l,t) \wedge BatteryLevel(lvl,t) \wedge \exists t_1 < t MailLoaded(t_1) \wedge \exists t_2 < t \neg MailCollected(l,t_2) \wedge \exists t_3 < t \neg MailAt(l,t_3) \wedge DeliverAndCollectMail(l,t) \rightarrow AtRobot(l,t+2) \wedge BatteryLevel(lvl-2,t+2) \wedge MailAt(l,t+2) \wedge MailCollected(l,t+2) \quad (4.9)$$

$$StationaryRobot(h1p3,t_1,t_2) \wedge t_2 > t_1+20 \wedge LoadFood(t_1,t_2) \rightarrow FoodLoaded(t_2) \quad (4.10)$$

$$AtRobot(l,t) \wedge BatteryLevel(lvl,t) \wedge \exists t_1 < tFoodLoaded(t_1) \wedge \neg FoodAt(l,t) \wedge DeliverFood(l,t) \rightarrow AtRobot(l,t+2) \wedge BatteryLevel(lvl-2,t+2) \wedge FoodAt(l,t+2) \wedge \neg DishesCollected(l,t+2) \tag{4.11}$$

$$AtRobot(l,t) \wedge BatteryLevel(lvl,t) \wedge \exists t_1 < t \neg DishesCollected(l,t_1) \wedge CollectDishes(l,t) \rightarrow AtRobot(l,t+1) \wedge BatteryLevel(lvl-1,t+1) \wedge DishesCollected(l,t+1) \tag{4.12}$$

For descriptions of the single predicates please consult table 4.1.

4.1.3 Geometry

The geometry of the building is also formulated in first-order logic. We just present here an excerpt of the whole list, in appendix A the complete set of predicates can be found:

$$\begin{aligned} &GridPoint(h1p1) \\ &\dots \\ &GridPoints(h4e2) \\ &GridEdge(h1p1, h1p2, 1) \\ &\dots \\ &GridEdge(h3e2, h4e2, 2) \end{aligned}$$

4.1.4 Verification

After the scenario was formalized in first-order logic, in order to verify its logical coherence, we implemented it in standard prolog. As there are some non-standard first order logic objects like paths and their lengths, we used prolog mechanisms to take care of them. During this implementation we made use of a routine to calculate paths in a graph taken from [3]. The listing of this program can be found in appendix B.

4.2 Results

The transfer to of the scenario to a model in first-order logic showed up some critical aspects regarding the scenario. Especially the whole path calculation and traveling is up to be solved. In this work we have not handled those things in first-order logic, but rather using an outer mechanism performing this work. By implementing the first-order logic model in Prolog, we were able to verify its logical coherence. The actions are executed in the order that is suggested by the plan, and at the end of the virtual day all the tasks have been performed fulfilling the hard and soft deadlines, which can be verified by consulting Prolog database. An additional feature is that path planning was implemented in Prolog, which can be used in future implementations. For further details see appendix B.
The Prolog implementation apart from verifying the logical model of the scenario, is also a possibility to implement the PlanVerifier in the architecture. In

order to verify arbitrary plans, an additional piece of software has to be written, which has as its output valid prolog code, which then can be run to verify the plan.

Chapter 5

Deploying the scenario to the architecture

After defining and verifying the robot's day scenario and creating an architecture we now try to deploy the scenario to the developed architecture. As already mentioned in the sections 3.2.2 and 4.2 the planner and plan verifier were not implemented in this work.
Additionally we found out that the automatic theorem prover in active logic does not have any interface, which would enable new observations to be streamed to the ATP dynamically. It is until now only possible to initially give a set of (temporally annotated) observations to the ATP, which then starts to reason about the knowledge regarding to the axioms and inference rules. This of course is a major obstacle when it comes to creating the developed architecture.
We therefore focused on using the automatic theorem prover as it is also used in [5] in two examples. This is using the ATP to work with step logic and a static set of observations. In the following we describe the creation of just such an experiment.

5.1 Formalization in Active Logic

As the ATP does not support dynamic updates of the observation set (OBS) as it is now, we have to create a set of observations in advance.

5.1.1 Axioms

The architecture splits the axioms into three different sets (see section 3.2.1). As only the logical laws are used in the ATP in this project, only they have to be transformed into the ATP syntax. In the ATP they are called inference rules. The rules that are used here, have already been used in other works like [5, 2].
As one can see there is no rule that would enable forgetting of contradictions. This has its reason in the fact, that forgetting is not possible in step logic.

$$\frac{i : \ldots}{i+1 : \ldots, \alpha} \quad if\ \alpha \in \text{OBS}(i+1) \tag{5.1}$$

$$\frac{i : \ldots, \alpha, (\alpha \rightarrow \beta)}{i+1 : \ldots, \beta} \quad \text{Modus Ponens} \tag{5.2}$$

$$\frac{i : \ldots, P_1 a, \ldots, P_n a, \forall x[(P_1 x \wedge \ldots \wedge P_n x) \rightarrow Qx]}{i+1 : \ldots, Qa} \quad \text{Extended Modus Ponens} \tag{5.3}$$

$$\frac{i : \ldots, \neg\beta, (\alpha \rightarrow \beta)}{i+1 : \ldots, \neg\alpha} \quad \text{Modus Tollens} \tag{5.4}$$

$$\frac{i : \ldots, \alpha}{i+1 : \ldots, \alpha} \quad \text{Inheritance} \tag{5.5}$$

$$\frac{i : \ldots, \alpha, \neg\alpha}{i+1 : \ldots, contra(\alpha, \neg\alpha)} \quad \text{Contradiction detection} \tag{5.6}$$

Figure 5.1: Inference Rule Set for the robot scenario

5.1.2 Observations

The observation set that is needed for this scenario now is limited to only statements of when the robot is where and which work has been performed at which point in time at which location.
Additionally to this list of observed actions or state changes, at some point in time (538) it is observed that one edge is blocked (i.e. not free). If everything works well the ATP discovers that there is a contradiction in its knowledge base at time point 541 and regarding to the rule in formula 5.6 adds a predicate to the knowledge base indicating a contradiction. This then would be detected by the Central Coordination Component, which would replan, reverify and finally let the robot execute the new plan avoiding the blocked edge. This process is assumed to take five minutes in the example we developed in this work. You can see an excerpt of the observation set showing this situation. The whole observation set can be found in appendix C.

```
...
[538,pred(atR,[h3p1,538])],
[538,[not,pred(free,[pred(edge,[h3p1,h3e1,538])])]],
[540,pred(batteryLevel,[880,540])],
[542,pred(batteryLevel,[879,542])],
[544,pred(batteryLevel,[878,544])],
[546,pred(batteryLevel,[876,546])],
[547,pred(batteryLevel,[875,547])],
[547,pred(atR,[h3p4,547])],
...
```

5.2 Results

Having created the inference rules and the observation set, we now let the ATP run using them as input. As one day has 1440 minutes, the ATP would have to perform at least 1440 iterations, to guarantee that each minute can be represented as one point in time. This already is a pretty coarse grained time

resolution but as we will see soon already this can lead to problems.
When executing the ATP using SWI-Prolog on a 32-bit workstation with 1GB main memory, the stack sizes of SWI-Prolog are limited to 128MB. This seems to be operating system independent, as we tested it on Windows XP, Linux, Mac OS X and the stacks always filled up after approximately 100 steps of reasoning.
If now the ATP's main function is invoked, the ATP starts to reason. At a certain point in time the execution stops, indicating that the stack size of the local stack was exceeded. The stack was always exceeded before time point 538 was reached. We have tested the contradiction detection by creating an observation set, that "produces" a contradiction much earlier. In this case the new contradiction predicate was successfully added to the knowledgebase of the ATP.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Active logic as a language to model real-world problems has interesting aspects and abilities. When developing a software that actually uses active logic and its facilities many problems can arise. Another aspect that has to be handled is to find an area in which active logics might be of use. The scenario developed in chapter 3 gives such an example of potential usage of active logic.
Integrating an existent active logic implementation as the automatic theorem prover into an software architecture also seems to be theoretically easy and was done in section 3.2 on the one hand. The practical implementation of such an architecture on the other hand is a task that is far from easy.
In chapter 5 we saw that the implementation of a PlanVerifier is possible, but could not be implemented during this short project. Anyhow as a proof of concept a Prolog implementation of one plan verification was developed.
As we then tried to deploy the modeled scenario to the active logic architecture, we on the one hand could find out that the ATP is not capable of dynamic manipulation of the observation set. And on the other hand that the stack size grows very fast using step logic with the ATP, leading to a stack overflow way before the scenario ends. As we did not use the LDS approach of the the ATP, we cannot say whether the stack overflows would have also occurred. But as LDS has a limited size for the memory of the agent, this problem should be avoidable using LDS.
These two facts just leave the conclusion, that the ATP as it is implemented now cannot be used in the architecture we developed.

6.2 Future Work

The future research in this area, especially concerning the scenario, the architecture presented presented in the report and the ATP can continue in many directions including some of the following:

- The ATP should be improved in a way that it allows dynamic manipulation of the observation set using a stream based approach.
- Another problem that should not be forgotten is the overflowing stack. One has to determine whether the memory management of the ATP or just the usage of step logic causes this.
- In order to test the scenario statically in the prover, it should be reformalized using the LDS approach of the ATP.
- The active logic implementation of the PlanVerifier and the Planner have to be done.
- With active logic being designed for real-life usage it would be very interesting to see how such a service robot as described in the scenario would work. This means that a deployment of the architecture to a real robot should be the ultimate goal.
- As it is theoretically possible, it would be very interesting to see whether an implementation of the architecture and the scenario completely in active logic actually can be done actually using existing hardware/software tools.

Appendix A

Geometry of the Building in FOL

This appendix only contains the full geometry representation of the building in first-order logic:

$$
\begin{array}{ll}
GridPoint(h1p1) & GridPoint(h1p2) \\
GridPoint(h1p3) & GridPoint(h1p4) \\
GridPoint(h1p5) & GridPoint(h1p6) \\
GridPoint(h1p7) & GridPoint(h1p8) \\
GridPoint(h1p9) & GridPoint(h1e1) \\
GridPoint(h1e2) & GridArc(h1p1, h1p2, 1) \\
GridArc(h1p1, h1p3, 1) & GridArc(h1p1, h1p4, 1) \\
GridArc(h1p1, h1e1, 3) & GridArc(h1p4, h1p5, 1) \\
GridArc(h1p4, h1p6, 1) & GridArc(h1p4, h1p7, 1) \\
GridArc(h1p7, h1p8, 1) & GridArc(h1p7, h1p9, 1) \\
GridArc(h1p7, h1e2, 3) & GridPoint(h2p1) \\
GridPoint(h2p2) & GridPoint(h2p3) \\
GridPoint(h2p4) & GridPoint(h2p5) \\
GridPoint(h2p6) & GridPoint(h2p7) \\
GridPoint(h2p8) & GridPoint(h2p9) \\
GridPoint(h2e1) & GridPoint(h2e2) \\
GridArc(h2p1, h2p2, 1) & GridArc(h2p1, h2p3, 1) \\
GridArc(h2p1, h2p4, 1) & GridArc(h2p1, h2e1, 3) \\
GridArc(h2p4, h2p5, 1) & GridArc(h2p4, h2p6, 1) \\
GridArc(h2p4, h2p7, 1) & GridArc(h2p7, h2p8, 1) \\
GridArc(h2p7, h2p9, 1) & GridArc(h2p7, h2e2, 3) \\
GridPoint(h3p1) & GridPoint(h3p2) \\
GridPoint(h3p3) & GridPoint(h3p4) \\
GridPoint(h3p5) & GridPoint(h3p6) \\
GridPoint(h3p7) & GridPoint(h3p8) \\
GridPoint(h3p9) & GridPoint(h3e1)
\end{array}
$$

$GridPoint(h3e2)$ $GridArc(h3p1, h3p2, 1)$
$GridArc(h3p1, h3p3, 1)$ $GridArc(h3p1, h3p4, 1)$
$GridArc(h3p1, h3e1, 3)$ $GridArc(h3p4, h3p5, 1)$
$GridArc(h3p4, h3p6, 1)$ $GridArc(h3p4, h3p7, 1)$
$GridArc(h3p7, h3p8, 1)$ $GridArc(h3p7, h3p9, 1)$
$GridArc(h3p7, h3e2, 3)$ $GridPoint(h4p1)$
$GridPoint(h4p2)$ $GridPoint(h4p3)$
$GridPoint(h4p4)$ $GridPoint(h4p5)$
$GridPoint(h4p6)$ $GridPoint(h4p7)$
$GridPoint(h4p8)$ $GridPoint(h4p9)$
$GridPoint(h4e1)$ $GridPoint(h4e2)$
$GridArc(h4p1, h4p2, 1)$ $GridArc(h4p1, h4p3, 1)$
$GridArc(h4p1, h4p4, 1)$ $GridArc(h4p1, h4e1, 3)$
$GridArc(h4p4, h4p5, 1)$ $GridArc(h4p4, h4p6, 1)$
$GridArc(h4p4, h4p7, 1)$ $GridArc(h4p7, h4p8, 1)$
$GridArc(h4p7, h4p9, 1)$ $GridArc(h4p7, h4e2, 3)$
$GridArc(h1e1, h2e1, 2)$ $GridArc(h2e1, h3e1, 2)$
$GridArc(h3e1, h4e1, 2)$ $GridArc(h1e2, h2e2, 2)$
$GridArc(h2e2, h3e2, 2)$ $GridArc(h3e2, h4e2, 2)$

Appendix B

Prolog First-Order Logic Model Verifier

Due to the vast amount of pages that would be here, if the whole listing was put here, we decided to just offer it as a download here: http://ai.cs.lth.se/xj/ThorbenHeins/prolog.tar.gz

Appendix C

Active Logic Formalization

Due to the vast amount of pages that would be here, if the whole listing was put here, we decided to just offer it as a download here: http://ai.cs.lth.se/xj/ThorbenHeins/al.tar.gz

Appendix D

Perl Parser

Due to the vast amount of pages that would be here, if the whole listing was put here, we decided to just offer it as a download here: http://ai.cs.lth.se/xj/ThorbenHeins/parser.tar.gz

Acknowledgements

I have to thank the supervisor of my work, Jacek Malec, for his steady and never ending support. The meetings and countless discussions always were very inspiring and productive. Apart from that it was very nice to experience that teaching and research can very well be motivating for both sides.

Thank you very much!

Bibliography

[1] M. L. Anderson and D. R. Perlis. Logic, self-awareness and self-improvement: The metacognitive loop and the problem of brittleness. *Journal of Logic and Computation 14*, 2004. 2.1

[2] M. Asker. Logical reasoning with temporal constraints, 2003. 2.1, 5.1.1

[3] I. Bratko. *Prolog Programming for Artificial Intelligence.* Pearson, 2000. 4.1.4

[4] W. Chong, M. ODonovan-Anderson, Y. Okamoto, and D. Perlis. Seven days in the life of a robotic agent. Technical report, University of Maryland and Microsoft, 2003. 3.1

[5] J. Delnaye. Automatic theorem proving in active logics. Master's thesis, Lund University, 2008. 2.2, 5, 5.1.1

[6] J. J. Elgot-Drapkin. *STEP-LOGIC: Reasoning Situated in Time.* PhD thesis, University of Maryland, 1988. 2.1, 2.2

[7] M. Ghallab, D. Nau, and P. Traverso. *Automated planning : theory and practice.* Morgan Kaufman Publ Inc, 2004. 3.2.2

[8] J. Hovold. On a semantic for active logic. Master's thesis, Lund University, 2005. 2.1

[9] M. Nirkhe, S. Kraus, and D. Perlis. Thinking takes time: A modal active-logic for reasoning in time. Technical report, Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, 1994. 2.1