VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

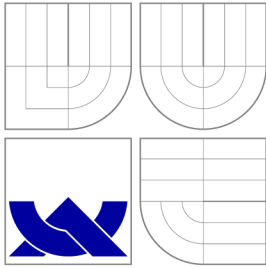# GRAPHIC DEVELOPMENT ENVIRONMENT OF AGENT LOW LEVEL LANGUAGE

DIPLOMOVÁ PRÁCE
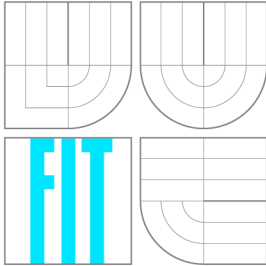MASTER'S THESIS

AUTOR PRÁCE   SZABOLCS KÜRTI
AUTHOR

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÉ VÝVOJOVÉ PROSTŘEDÍ AGENTNÍHO JAZYKA ALLL
GRAPHIC DEVELOPMENT ENVIRONMENT OF AGENT LOW LEVEL LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE   SZABOLCS KÜRTI
AUTHOR

VEDOUCÍ PRÁCE   Ing. JAN HORÁČEK
SUPERVISOR

BRNO 2012

Abstrakt

Cílem této práce je návrh a implementace grafického vývojového prostředí pro agentní jazyk ALLL. Jazyk ALLL bude představen detailně. Čtenář bude seznámen s frameworkem ANTLR pro generování nástrojů pro rozpoznávání jazyka. Teoretické základy multiagentních systémů a možnosti zvolené platformy budou taky rozebrány. Po popisu implementace komplexního grafického vývojového prostředí jsem zařadil popis testování a ohodnocení dosažených výsledků.

Abstract

The aim of this work is to design and implement a graphic development environment of agent language ALLL. Language ALLL going to be described in details, such as the ANTLR framework for generating language recognition tools. Theoretical basis of multi-agent systems, together with the features of the selected platform, will be discussed as well. Description of the implementation is followed by the presentation of testing. Closure deals with the discussion of the achieved results.

Klíčová slova

Keywords

Citace

Graphic development environment of agent low level language

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Horáčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

<div align="right">

. . . .

Szabolcs Kürti

May 21, 2012

</div>

Poděkování

Chtěl bych poděkovat Ing. Janu Horáčku za odbornou pomoc a vedení mé diplomové práce.

Contents

Chapter 1

Preface

This master's thesis deals with the problem of creating a graphic development environment of Agent Low Level Language (ALLL). This language has been created at the Brno University of Technology for programming agents in wireless sensor networks. This work tends to present a detailed analysis, implementation and testing of the graphic development environment.

This work is built on basics laid down by several others. I drew a lot of information from bachelor theses [6] and [11] regarding to the syntax of ALLL and the semantic meaning of individual language constructions.

First chapter contains a comprehensive theoretical analysis. The main problem is decomposed into subtasks and those are examined in detail. My conceptual guideline is to describe general concepts first, and place current problems and requirements into that context. A wide variety of problems is examined. Creation of a graphic development environment combines a number of different fields like language recognition, graphic user interface programming and simulation.

The theoretical analysis is followed by the description of the design. Chapter design tends to present proposed solutions to previously analysed requirements. Compilation and decompilation of abstract language elements are discussed in detail. Besides language recognition tasks graphic user interface issues are presented as well. Chapter contains a complete design for source code visualisation, just like the description of possible interactions between the user and the graphic interface.

Chapter implementation presents how the design of the graphic development environment was realized. In that chapter we move from general concepts to specific solutions: classes and code snippets solving explicit problems are listed there. The development environment was implemented in language Java.

Description of the implementation is followed by a chapter dealing with testing. Tests are coupled into several different groups. The last group is formed by tests intended for real life sensor networks. Those are presented in detail as it is required in the fourth point of the assignment.

Current state of the development, achieved results and possible improvements are discussed in the closure.

Chapter 2

Theoretical analysis

This chapter tends to present the theoretical analysis of the problem how to create a graphic development environment for the Agent Low Level Language (ALLL). The analysis can be decomposed into studying the following sub-problems:

- general concept of multi-agent systems and possibilities of the selected platform,

- compile and decompile ALLL source code,

- requirements set against this specific graphic development environment,

- how to represent and edit compiled code in a visual manner,

- model wireless sensor networks as a multi-agent system,

- simulate the behaviour of the modelled system.

The rest of this chapter deals with the detailed analysis of the above mentioned tasks. A good analysis is very important. Time is one of the key aspects in every development process. Good understanding of the problem can save us a lot of unnecessary extra work caused by bad decisions made in early stages.

2.1 Multi-agent systems and the WSageNt platform

This section gives an overview about how the general concept of multi-agent systems and the WSageNt platform are related to each other. Key features of WSageNt platform are presented and discussed briefly as well.

2.1.1 Multi-agent systems

A multi-agent system (MAS) is a system composed of multiple co-operating intelligent agents within an environment. Agents are designed to collect information about their surroundings. The acquired knowledge can be processed and shared with other agents in the environment. Multi-agent systems can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. Typical characteristics of multi-agent system according to [2] are the followings:

- agents are autonomous entities,

- no agent has a full global view of the system,

- there is no appointed central decision making agent which controls the others, otherwise the whole system would be reduced to a monolithic system.

2.1.2 WSageNt platform

The description of the WSageNt platform presented here is based on information available on the web page of the project [5]. The WSageNt platform is capable to run agents in wireless sensor nodes (MICAz and IRIS motes). Key features of this platform match the typical characteristics of a multi-agent system. The multi-agent system is represented by the network of wireless sensor nodes.

The core of the platform is programmed in TinyOS 2.x. Software is divided into two major parts: agent platform and the interpreter of ALLL code. Key features of the agent platform are:

- agent can collect data from the sensor attached to the currently seized node,

- communication of the agents is solved by sending messages,

- agents together with their knowledge can be cloned and loaded into another node.

The interpreter executes programs written in ALLL. Key features of this language are discussed briefly in the followings.

2.1.3 Agent Low Level Language

The Agent Low Level Language belongs to the long family of imperative languages. Programs written in ALLL are not compiled but interpreted. One of the main design goals was, that a source code written in this language should be very small in size because of the limited memory resources. ALLL is a low level language, it follows that there are no data types and instead of using variables programmers directly manipulate with registers.

Certain features of this agent language shows similarity with a variety of different other languages. For example ALLL is an imperative language like C but stores information in a belief base like logical language Prolog. The behaviour of the multi-agent system depends on a proper interaction of several agents. From this point of view programming in ALLL is very similar to using a message parsing interface like Open MPI. Parallel execution of several nodes must be kept in mind, and their proper synchronization has to be solved.

2.1.4 Structure of a ALLL agent

If we examine things with a top-down approach, a wireless sensor network forms a multi-agent system. Interacting agents can solve certain tasks according to their predefined behaviour written in ALLL. An agent can be decomposed into the following parts:

- identification,

- plan base,

- plan,

- registers,

- knowledge base:

  - belief base,
  - input base,

Execution of agents start at the plan. Plan is a list of actions dynamically changing during execution. However I decided to introduce two new terms namely *goal* and *action stack*. Goal remains the same during execution, dynamic changes are displayed by the action stack. It was necessary because of the source code editor. Goal is part of the source code, but the visualized code is not changing dynamically. The point of execution can be followed by a pointer, which jumps from action to action in the source code as it is executed. In the rest of this text term *plan* refers to an item of the plan base.

The goal and the plan base represent the behavioural description of the agent. Data measured by sensors and incoming messages are collected into the input base. The knowledge, acquired by the agent about its surroundings and its own state, can be stored in the belief base, while registers serve for data manipulation. Knowledge is represented in a form of tuples containing textual and/or decimal data.

Plan base is an unordered list of plans. Plan is an ordered list of actions labelled with a name. The label primarily serves for identification, so it has to be unique within the agent. Goal of the agent is special type of a plan without a name. It is always present, while the plan base can be empty.

During execution 3 registers can be used to store results returned by actions. Registers are addressed by register symbols &[1-3]. Certain actions accept tuples that can contain register references. In those cases content of the register is copied to replace the symbol, or the action fails if an empty register is referenced.

There is platform specific limitation for agents. The size of the whole agent can not exceed 2 KB. This is because of the limited memory available on motes. The size comprehends all parts of the agent: plan base, goal, action stack, input base, belief base and the registers.

2.1.5 Model of a WSageNt network

WSageNt networks can be modelled using the following items:

- sensor nodes,

- agents and

- links.

Model of a system is another system that emphasizes certain attributes of the original system while other features are abstracted. Communication between wireless nodes, in real life, is done through radio waves. Because of distance or different obstacles e.g. walls, some nodes can communicate while others can not. Not to mention that temporary signal fade outs can cause unforeseeable communication package loses. This project does not intend to deal with the radio communication. Nodes that can communicate are connected by links; if the link is missing, nodes are mutually unavailable for each other. The most important and widely used topologies of sensor networks are the followings according to [10]:
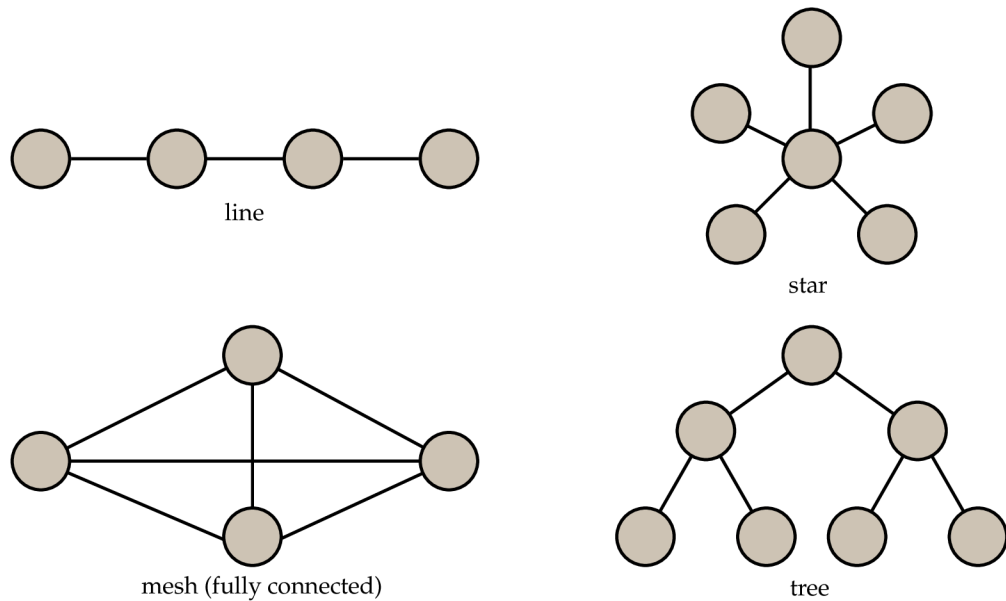
Figure 2.1: Frequently used network topologies

2.2 Compilation and decompilation

In order to be able to edit an ALLL source code it needs to be compiled into an inner representation. In our graphic development environment agents are represented by objects. Every part of the agent has its own object representation described by dedicated classes. Not only the behavioural description of the agent is translated. Initial knowledge: content of the registers, the input and the belief base are compiled to an inner representation too.

Decompilation means a reverse action, when a ALLL source code is created from the object representation, which is a relatively easy task in comparison with compilation. Every object knows how to represent its content in a textual form. Agents are exported into a ALLL source code by decompilation.

2.2.1 The ANTLR framework

Compilation of the source code into a inner representation requires language recognition tools. This subsection tends to present why the ANTLR (Another Tool for Language Recognition) framework has been selected for this task.

Building a lexer or a parser manually is a very demanding task. ANTLR is a framework for generating lexers, parsers and other language recognition tools. The main advantage of using such a framework is that we can focus on the primary problem and leave the monotonous work to the generator. ANTLR is written in Java and supports a variety of target languages, namely: Java, C, C++, C#, Objective-C, Python and Ruby. Terence Parr from the University of San Francisco stands behind the development since 1989. ANTLR Works is a grammar development environment with a number of features e.g. grammar visualisation. I have decided to use this framework because of my good previous experiences. I worked with the ANTLR plugin available for Eclipse. Description of the framework was adopted from [9].

Tools generated by ANTLR use LL(*) parsing strategy that supports more natural grammars then other approaches. It is called LL because it recognizes input from left to right using a leftmost derivation. The other big class of recognizers is called LR because they use rightmost derivations (for example YACC generates LR-based language tools). LR recognizers try to match lexemes of the input with the leaves of the parse tree and work their way up toward the starting symbol. While LL recognizers are goal-oriented and walk the parsing tree in a top-down way. With a starting rule in mind try to match the alternative rules. This kind of language processing is very similar to the way how humans understand complex sentences.

The strength of a LL recognizer depends on the amount of lookahead it uses. With a lookahead of 1 the recognizer can scan 1 symbol ahead if unable to decide which alternative rule to use based on the next symbol. Top-down recognizers with a fixed amount of lookahead k, are called LL(k). One of the biggest advantages of the ANTLR v3 is the LL(*) parsing strategy that allows lookahead to roam arbitrarily far ahead without a considerable reduction in recognition speed. This dramatically increases the number of acceptable grammars. However ANTLR has its own limitations too. Not all useful grammars are LL(*) for example nested structures with a recursive definition can not be accepted. ANTLR has built in strategies how to deal with this kind of problems but those are beyond our consideration.

The ANTLR has been selected mainly because of the above mentioned features of the framework. The following types of language recognition tools are generated by ANTLR for this project:

- lexer,

- parser and

- tree walkers.

Lexer decomposes input into a stream of lexemes. Parser builds from the lexemes an abstract syntax tree (AST). Tree walkers walk through abstract syntax trees to execute actions defined by the programmer.

2.2.2   Actions and services used in ALLL

Actions are the most fundamental building blocks of ALLL. Service call is a special type of a action that allows to call services of the host platform. A detailed analysis of actions and services is required in order to be able to compile source code into a proper inner representation. One of the most important things is to know the number and the type of parameters that individual actions require. Tables of this section were adopted from [6] and [11] with some minor changes.

code	parameter list	description
+	$[register, tuple]$	Adds content specified by the parameter to the belief base.
-	$[register, tuple]$	Removes tuples unifiable with the specified parameter from the belief base.
&	$([1-3])$	Changes and clears the active register.
^	$[(string), register]$	Calls the plan specified by the parameter
@	$(list\ of\ actions)$	Inserts the specified list of actions to the action stack with and additional „catch" (#) at the end.
!	$[integer, register], [tuple, register]$	Sends a message (second parameter) to the address specified by the first parameter.
*	$[tuple, register]$	Copies tuples unifiable with the specified parameter from the belief base to the active register.
?	$[integer, register, \_]$	Relocates tuples from the input base to the active register. Parameter specifies the source address, „_" selects the first message.
$	$(service)$	Calls the specified service.
#	$none$	If an error happens actions are deleted from the stack unit this action is encountered.

Table 2.1: Table of ALLL actions

2.2.3 High level abstractions

Programming in a low level language like ALLL gives a lot of control over how the code will be interpreted or executed; however, it is often hard to understand the logic behind such a code. High level programming languages use abstractions to hide the details of the computer, and allow to use natural language elements. There is a high level programming language for describing the behaviour of our agents. It is called Agent High Level Language (AHLL). This language has been created by Bc. Robert Kalmár in his bachelor's work: *Language of Higher Level of Abstraction for Programming Mobile Intelligent Agents*. The above mentioned high level language has a very similar syntax as C: global and local variables, if-then-else structures, program blocks, cycles and function calls are natively supported by AHLL.

The aim of this master's thesis is to create a graphic development environment for the Agent Low Level Language, nevertheless it is quite obvious, that implementing a selection of the previously mentioned higher level structures would have a great impact on the usability of the environment. In my opinion the most important abstraction is the if-then-else structure, because it enables to make a decision and continue the execution of the program on different paths.

In this project I am going to focus on the implementation of the if-then-else structure. Program codes, containing abstract language structures, needs to be translated into ALLL

code	parameter list	description
a	*none*	Activates monitoring of incoming messages.
f	$[list, register]$	Copies the first item of the specified list to the active register.
k	*none*	Kills the interpreter.
l	$[(colour\langle, state\rangle?), register]$	Controls LEDs. Colour „r", „g" or „y" specifies the red, green or the yellow diode respectively. State is optional; if it is omitted the selected diode is toggled between states on (1) and off (0).
m	$([integer, register]\langle, s\rangle?)$	Copies the agent to the node addressed by the parameter. Character „s" attached to the address stops execution of the current agent.
r	$[list, register]$	Copies the tail of the specified list to the active register.
s	*none*	Suspends execution until a message arrives.
w	$[(integer), register]$	Suspends execution for the specified amount of time (milliseconds).
d	$[none, (type, integer), register]$	Collects data from the sensor into the input base. If no parameter is specified the current value is used, otherwise the average (a), minimum (m) or maximum (M) of the last x (integer) measurements is returned.

Table 2.2: Table of supported services

code	meaning	explanation
mul	multiply	operand_1 $*$ operand_2
div	integer division	operand_1 $/$ operand_2
mod	remainder after integer division	operand_1 $\%$ operand_2
add	addition	operand_1 $+$ operand_2
sub	subtraction	operand_1 $-$ operand_2
les	less than	operand_1 $<$ operand_2
leq	less than or equal	operand_1 $\leq$ operand_2
mor	greater than	operand_1 $>$ operand_2
meq	greater than or equal	operand_1 $\geq$ operand_2
equ	equal	operand_1 $==$ operand_2
neq	not equal	operand_1 $!=$ operand_2
and	logic and (conjunction)	operand_1 operand_2
orr	logic or (disjunction)	operand_1 operand_2
not	logic not	$!$ operand_1
min	unary minus	$(-1)*$ operand_1
cpy	copy variables	operand_1 $=$ operand_2

Table 2.3: Mathematical operations supported by the platform

before we load them into sensor nodes. The development environment has to support export/import of abstract structures. There are two ways of doing this. The first option is, that we define a new language which supports these structures plus native ALLL constructions. We store our programs in this language and use an export to translate it into ALLL. The other option is that the development environment produce only clear ALLL source code and abstract structures are recognized in this code. The main advantage of the first solution is that recognition of abstract structures is a non-trivial problem, however to define a new language just to hold our structures seems to be redundant and needless. That is, why I decided to implement the second option - recognize if-then-else structures in the produced ALLL source code.

2.3   Semantic meaning of selected actions

For debugging purposes actions need to be executed. This demands not only to be aware of the syntactic structure of a ALLL program, but also to know the semantic meaning of individual actions. Some actions are quite straightforward, while others require some explanation. Actions with a complex meaning are described in this section.

2.3.1   Mathematical operations

Mathematical operations supported by the agent platform are listed in table 2.3. Description of mathematical services was adopted from [6]. Because ALLL is a low level language, something like variables do not exist. Despite this, tuples with the following structure can substitute variables:

$$((name),value),$$

where the first item, representing the name, is a tuple, and the second item is an integer value. From now on expression variable will refer to this structure instead of its general meaning.

Registers can contain multiple variables for mathematical operations:

$$((name),value\_1)((name),value\_2)...((name),value\_n),$$

or values can be present even without names:

$$(value\_1)(value\_2)...(value\_m).$$

Mathematical operations can be divided into binary and unary operations

- In case of a **binary operation** the result is computed by applying the operator on the Cartesian product of input variables.

- Result of a **unary operation** is computed by applying the operator on every variable of the operand.

Binary operations have the following form:

$$\$(o, type, op\_1, op\_2, name\_1, name\_2, name\_result),$$

where:

- **type** denotes the selected binary operation;

11

- **op_1** and **op_2** represent a single variable or a register symbol;

- **name_1** and **name_2** are tuples for filtering input variables. A variable is accepted by the operation only if its name corresponds with the filter. Variables without names are selected by empty tuples.

- **name_result** defines the name of the result.

Unary operations have the following form:

$$\$(o, \ \texttt{type}, \ \texttt{op\_1}, \ \texttt{name\_1}, \ \texttt{name\_result}),$$

where individual items of binary and unary operations are analogous.

2.3.2 Message sending

An agent can communicate with other agents within the environment by sending messages. Messages arrive to the input base and have the following structure:

$$(\texttt{source\_address, (content)}),$$

where `source_address` is an integer value and `(content)` is a tuple. Register references in the content are substituted before sending the message.

Messages are delivered from node to node. Address identifies the node, not the communicating agent. Messages are received by the agent at the targeted node. Messages can be sent only to nodes that are directly visible from the source. The source node tries to send the message in a cycle until it is successfully sent. If the target node never becomes available, then the sender stuck in an infinite loop.

2.3.3 Agent travelling

Agents can be moved between nodes by calling the „*move*" *(m)* service. If the target node is not available, the agent ends up in a loop, just like in message sending. If the targeted node is seized by another agent it is kicked out; new agents are always prioritized.

Depending on how this service was called the agent at the source node can be stopped or continued to run. If the execution on the source node continues, the agent is, practically speaking, cloned and the clone is sent to the target node. When an agent is cloned its whole inner structure is copied including even the content of registers and the point of execution. The new agent has the same goal, action stack, plan base, belief and input base as the original. The execution continues from the action that succeeds the recently called „*move*" *(m)* service.

2.4 Integrated development environment

An integrated development environment (IDE) is a software application that provides a comprehensive set of tools to programmers for software development. In general IDE consists of the following parts:

- source code editor,

- compiler and/or interpreter and a

- debugger.

In development environments programmers usually work with projects. Projects help to organize the work. In our graphic development environment project represents a wireless sensor network or generally speaking a multi-agent system. The following information need to be stored in a project file:

- the topology of the network,

- agent descriptions,

- initial position of agents,

- comments and

- additional information.

Commenting our source code is more than just a good habit. In lot of companies even financial bonuses are withheld until the produced source code is properly commented. Comments are stored in the project file, because they are not allowed in ALLL due to the previously mentioned size issues of the source code.

2.4.1 Source code editor

The source code editor serves for editing the plan base and the goal of an agent, and it has to support the following tasks:

- add a new plan,

- delete,

- rename or

- edit and existing plan.

We can think of a goal as a special type of a plan which is always present. It can not be deleted or renamed but its content is available for editing.

If we want to add a new plan its name has to be defined in advance, because its uniqueness is controlled first. Plan base can not contain multiple plans with the same name. Plan names also have to match the following regular expression:

$$([a-zA-Z0-9])*$$

The same limitations are applied to renaming as well. After the name is verified, a new plan is added to the belief base or the selected one is renamed.

Plans of the plan base are available for editing. Editing can be divided further into the following subtasks:

- add a new action,

- delete an existing action,

- change the order of actions within a plan,

- exchange actions between plans and

- edit parameters of actions.

A detailed description of these tasks is presented later, where their graphical realization is discussed as well.

2.4.2 Interpreter

In our graphic development environment (GDE) ALLL interpreter does not execute the textual form of the source code, but the compiled inner representation. Central part of the interpreter is the action stack. Execution continues until the action stack is emptied or the execution is suspended for some reason. Execution starts with loading actions of the goal to the action stack. From the interpreters point of view there are three different types of actions.

- The first type of actions changes only the inner state of the agent.

- Actions of the second type require cooperation from the network (send message, move agent).

- The third class modifies directly the state of the interpreter (wait, suspend, kill, load plan).

2.4.3 Debugger

Debugger is one of the most important parts of any development environment. The purpose of the debugger is first and foremost to reveal to potential flaws in the logic of the produced program. The user plays an active role in this process. There are different types of debugging:

- examine every action step by step,

- set up breakpoints or

- use conditional breakpoints where debugger stops only if specific conditions are encountered.

I have decided to implement the first solution. In my opinion breakpoints are useful for programs that are longer and more complex than a typical ALLL program.

To debug only one agent in a multi-agent system has a little meaning. Agents behave like different threads synchronized by messages. In a well written program the order, in which individual threads (agents) are executed, does not matter. Two different types of debugging is supported by our GDE:

- individual or

- system.

Individual debugging executes one agent step by step. It has the advantage that changes of the knowledge base and the action stack can be examined after every step. During system debugging every agent take one step and we can see how the whole system evolves.

2.5 Graphical development environment

The aim of this work is to create a graphic development environment that extends a simple IDE with visual code editing. In a graphic IDE source code is manipulated rather visually than textually. One of the key aspects of development environments is to make programming more convenient and effective to increase productivity. Visual information in general

is more acceptable for humans than a textual form. This is true in particular if we consider the Agent Low Level Language. Actions and services labelled with one single character, no white-spaces, nested tuples and language constructions can be very confusing. Humans do not necessarily understand statements accepted by machines. Programming languages have been born to bring closer machine languages to natural languages. Graphic development environments take this idea one step further as they form a bridge between programming languages and the human understanding.

2.5.1 Graphic User Interface

Graphic user interface (GUI) of our development environment can be divided into two major parts:

- static and

- dynamic graphic user interface.

Expression static GUI refers to the main frame of the application and its static subcomponents. Those are predefined panels and internal frames that do not change their inner structure. On the other hand, there is the dynamic graphic interface which is closely related to the visualisation of the network topology and the agents.

2.5.2 Model-view-controller

Model-view-controller (MVC) is a software architecture that separates the business logic of the application from the user interface. [3] The main advantage of this approach is that isolated domains can be developed, tested, and maintained independently. The user interface of the graphic IDE is rather complex, so, it is particularly important to divide the presentation layer from the logic domain. The control flow of MVC can be described as follows:

- The user interacts with the user interface.

- The controller handles the event and translates it into an appropriate logic action.

- The controller notifies the logic model of the action.

- The action can change the state of the model. The view queries the model whether update of the user interface is required or not.

- The user interface waits for further actions from the user. The control flow cycle is restarted.

Chapter 3

Design

This chapter deals with the design of the application based on the theoretical analysis presented in the previous chapter. Several different topics are discussed like language recognition, the logic program model and the graphic user interface.

3.1 Abstraction to ALLL compilation

Bachelor's work of Bc. Robert Kalmár, among other things, deals with translation of high level structures into ALLL source code. Some of his results are integrated into this work with some minor changes. Conditioned structures can be implemented due to a specific feature of the language. If a certain tuple can not be unified with the belief base, then the action stack is cleared until its totally emptied or the first „catch" (#) action is encountered.

The pseudo program code below explains how an if-then-else structure can be represented in ALLL. The following text is adopted from [6] with some minor changes.

```
macro if $condition $then $else
    @($condition_true -$condition_true $then)
    @($condition_false -$condition_false $else)
macro-end
```

After the condition is evaluated the result is added to the belief base as a tuple. After condition evaluation a „direct run" (@) action is executed. The first action inside tries to unify the true condition with the belief base for branch „then" while false condition represents the „else" branch. In case of a true condition unification succeeds and the execution continues. For the „else" branch unification fails, and actions of this branch are deleted.

Decisions are based on mathematical binary relations. Because of the low level language condition statements needs to be kept quite simple: one register or operand can figure on both sides of the condition and the two sides are compared by one mathematical relation. In the followings transition of an if-then-else structure to a sequence of ALLL actions will be presented step by step through an example.

Example 1. Express the following sentence in ALLL. *If the value in 1st register is less or equal to the value in the 2nd register, then insert into the belief base tuple (yes), otherwise insert tuple (no).*

The first thing we need to do is to identify which register can be used for condition evaluation. In this case the 3rd register is not used by the mathematical operation. Assume that the belief base (BB) is empty and registers `&1`, `&2` contain values (1), (2) respectively.

```
&(3)                              &3:   empty
$(o,leq,&1,&2,(),(),(cond))       &3:   (((cond),1))
+&3                               BB:   (((cond),1))
@(                                AS:   ..+(yes)#..
  *(((cond),1))                   &3:   ((((cond),1)))(((cond),1))
  -(((cond),1))                   BB:   empty
  &(3)                            &3:   empty
  +(yes)                          BB:   (yes)
)
@(                                AS:   ..+(no)#
  *(((cond),0))                   AS:   empty (because action failed)
  -(((cond),0))
  &(3)
  +(no)
)
```

This is the right place to mention, that to ensure correct behaviour, one minor restriction needs to be introduced. Action „catch" (#) can not be used within the abstraction, because this implementation assumes, that at time of calling action `*(((cond), 0/1))` the closest # symbol on the action stack (AS) defines the end of the current branch.

The difference between mine and the previously implemented solution is that action `-(((cond),0/1))` is called in both branches. This action removes the result of the condition evaluation from the belief base immediately after evaluation. The same if-then-else abstraction can be called recursively without reaching its end. If the result of the previous evaluation is not removed from the belief base, it can lead to an undesired behaviour.

3.2 Recognition of abstract language elements

The previous section presented how an if-then-else abstraction is translated into ALLL. This section deals with the reverse transition - how abstract structures can be recognized in a sequence of ALLL actions. First we need to decide what kind of mechanism is required for abstraction recognition. Language ALLL enables nested structures for example `@(*(1,_))`, so, it is not a regular language. A context-free grammar is required to match start and end brackets. Languages generated by context-free grammars are equivalent to languages accepted by pushdown automatons. A pushdown automaton is need to be created in order to recognize abstract structures. Design of the automaton respects that several abstractions can be nested into each other. I would like to emphasize that this automaton is not used to parse the ALLL language, but the sequence of ALLL actions. Language recognition tools generated by the ANTLR framework compile the source code into its inner representation, and abstract structures are searched in the compiled code. Textual and graphic representation of automaton M_{abs} was based on examples from [1]. The automaton is defined as:

$$M_{abs} = (Q, \Sigma, \Gamma, \delta, q_0, Z, F), \text{ where}$$

- $Q = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \text{then}, \text{else}, f_1, f_2, f_3\}$ is the set of states.

- Σ: input alphabet is equal to the set of actions supported by ALLL.

- $\Gamma = \{\ \$, @(,)\ \}$ is the stack alphabet.

- δ: transition relations are described in the subsequent picture.

- Starting state $q_0 = 1$.

- $Z = \$$ is the initial stack symbol.

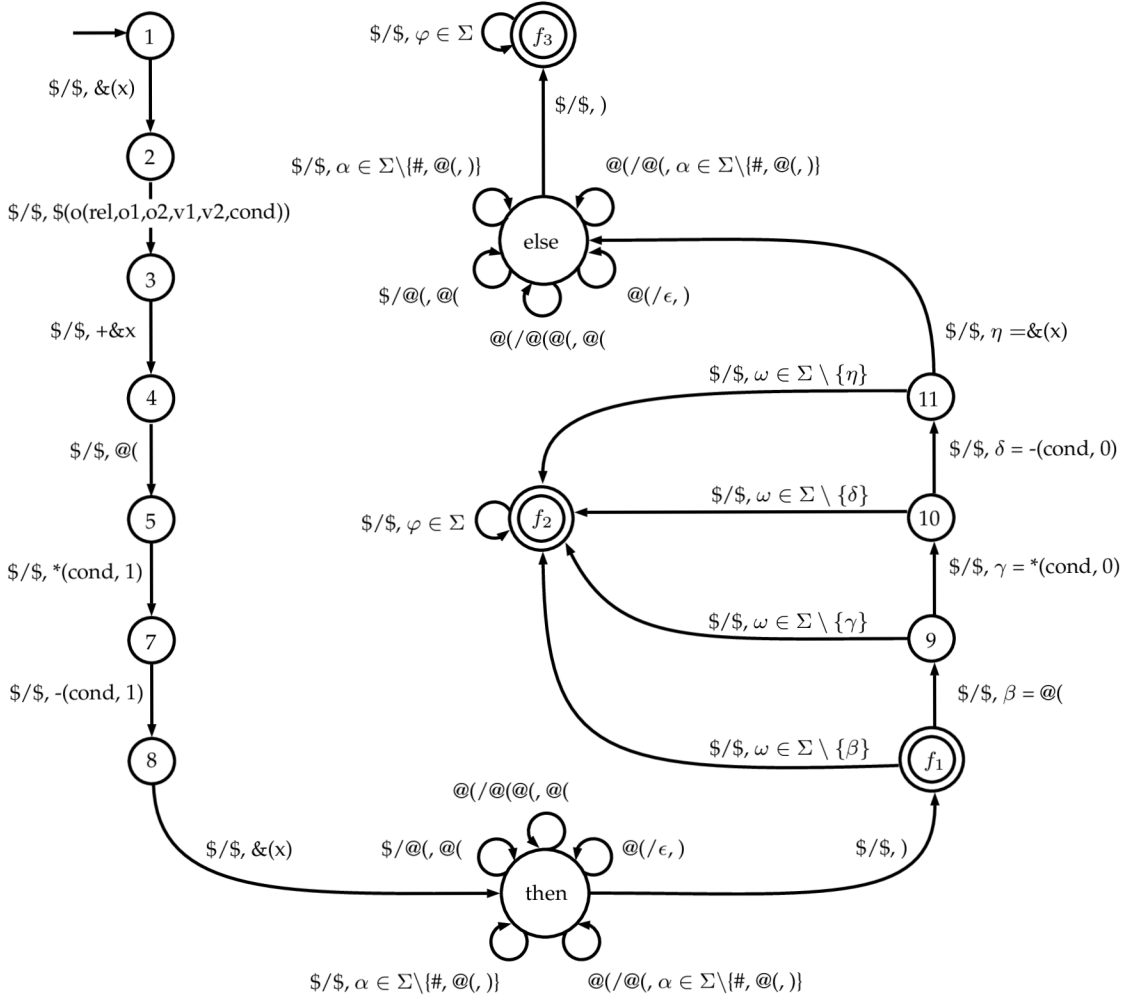- $F = \{\ f_1, f_2, f_3\ \}$ is the set of accepting states.

Figure 3.1: Pushdown automaton M_{abs}

Pushdown automaton M_{abs} accepts the input sequence of actions if an if-then-else abstraction starts with the first action. States 1-3 match condition evaluation, states 4-8

18

accept actions that control whether „then" branch should be invoked or not. State *then* accepts the content of the „then" branch, including nested if-then-else abstractions. The machine accepts input by final state f_1 if there are no more actions after the „then" branch. States 9 - 10 accept actions that control whether „else" branch should be invoked or not. Acceptance by final state f_2 means that the „then" branch is followed by at least one other action but it does not represent an „else" branch. State *else*, just like state *then*, accepts the content of the current branch. The input is accepted by final state f_3 if the first action of the sequence starts a complete if-the-else abstract structure. States f_2 and f_3 accept any action supported by ALLL, because we need to accept a sequence which ends at the end of the program. Practically speaking the machine consumes everything what follows the abstract structure. Bracket matching is not controlled in these states, the input has to be a valid and correctly encapsulated sequence of ALLL actions.

3.3   Static graphic user interface

Static graphic user interface comprehends the layout of the main window of the application and its subcomponents. The main frame is structured as follows.
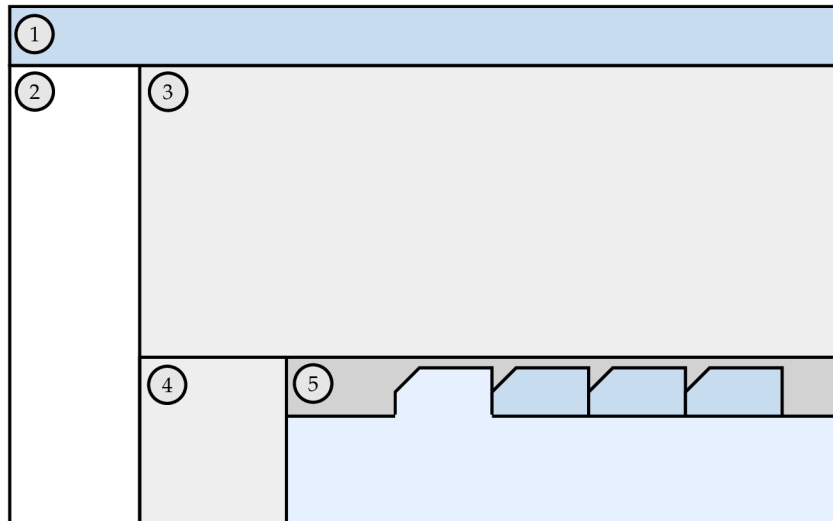

Figure 3.2: Sketch of the main frame

Menu bar is at the top of the main frame (1). Column on the left (2) is reserved for language elements (actions, services, abstractions). The right side is divided horizontally. The upper part (3) deals with the visualization of the topology and the behavioural description of agents. The bottom part is further divided into a control panel (4) and a list of other panels (5). Control panel is for debugging purposes, it contains the following information about the currently displayed agent:

- content of registers,

- id of the currently active register,

- state of the agent described by the following flags:

- monitoring incoming messages,
- waiting for a message,
- interpreter is killed.

The control panel also displays details about the node seized by the displayed agent. Sensor node related information are the following:

- id of the node,

- actual value measured by the sensor.

The sensor field displays editable data. It helps to debug programs with decisions, where agents can behave differently according to the value returned by the sensor.

Tabbed pane labelled with number 5 contains the following panels:

- code view & import,

- knowledge base,

- project properties and

- console.

Code view & import panel displays the ALLL source code of the agent. Whatsoever change is made to the agent through the graphic interface, the effect on the textual representation is instantly displayed on this panel. Another task that has been placed here is the agent import. A file chooser dialogue is displayed after a click on the „import agent" button. If the selected file represents a valid agent, the actually displayed is replaced by the new definition.

Knowledge base panel enables to initialize the content of the belief and to follow their changes during debugging. The third field displayed by this panel is not editable. It represents the current state of the action stack. Actions waiting for execution are listed there.

Project properties panel enables to view and edit basic properties (title and description) of the project. The result of the last executed task, for example: open project, import agent, initialize knowledge base, start debugging, stop debugging, save project, etc. is displayed on the console panel. In case of an error a detailed error description helps the user to find out what the problem might be.

3.4 Dynamic graphic user interface

Dynamic graphic user interface can be divided into the topology editor and the source code editor. Topology editor serves for modelling wireless sensor networks while the source code editor visually represents the source code of agents in an editable way. Dynamic graphic user interface is called dynamic because it can restructure itself according to the users interactions with interface. Dynamic GUI is very similar to a painters canvas. User can add new elements, remove or edit existing content.

3.4.1 Topology editor

The desired topology can be created with just a few clicks in the topology editor of the development environment. Agents can be inserted to the created network. Every node can hold exactly 1 or 0 agent. Nodes contain sensors for collecting data and a set of red, green and yellow diodes for visual signalization. Agents can be dragged into a different node or
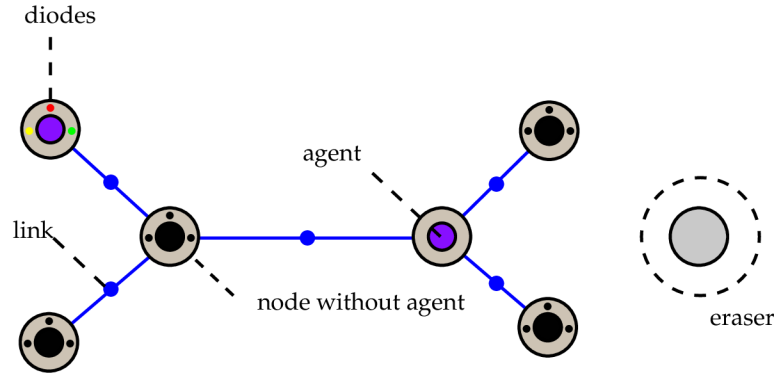

Figure 3.3: Topology editor

deleted by dropping them outside a node. Links are editable too. Unneeded nodes can be deleted by dragging them over the eraser object. If a node is deleted, the attached links are deleted as well. If the node contained an agent it is removed from the network too.

3.4.2 Plan base visualization

This section deals with the visual representation of programs written in ALLL. The focus is on the behavioural description. The main task is to represent graphically the goal and the plan base of agents in a editable form.

Plan base is an unordered collection of plans. Plan contains actions in a specified order. In the rest of this section visual representation of these structures will be explained in a bottom-up way. First actions and their parameters are examined, after that plans are presented, and last but not least the graphic representation of a plan base is described.

Actions consists of the following parts:

- name,

- parameter list and

- alternative parameters.

Actions are identified by their names. Parameter list of an action contains 0 to n parameters. In some cases the the actual value of the parameter can be replaced with a register reference. This is solved in our graphic IDE by using alternative parameters. The user can select whether he/she wants to use an explicit value or a register symbol. In order to make the best use of the available space, only the currently selected parameter is visible by default. The list of alternatives becomes visible only if the user wants to replace the currently active parameter. For example action „add to belief base" (+) can be defined in the following ways:
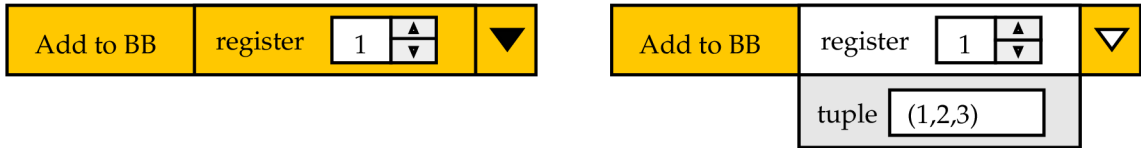
- `+&1`

- `+(1,2,3).`

Figure 3.4: Graphical representation of the „add to belief base“ (+) action

Visual representation of a plan contains the following major components:

- header,

- left column,
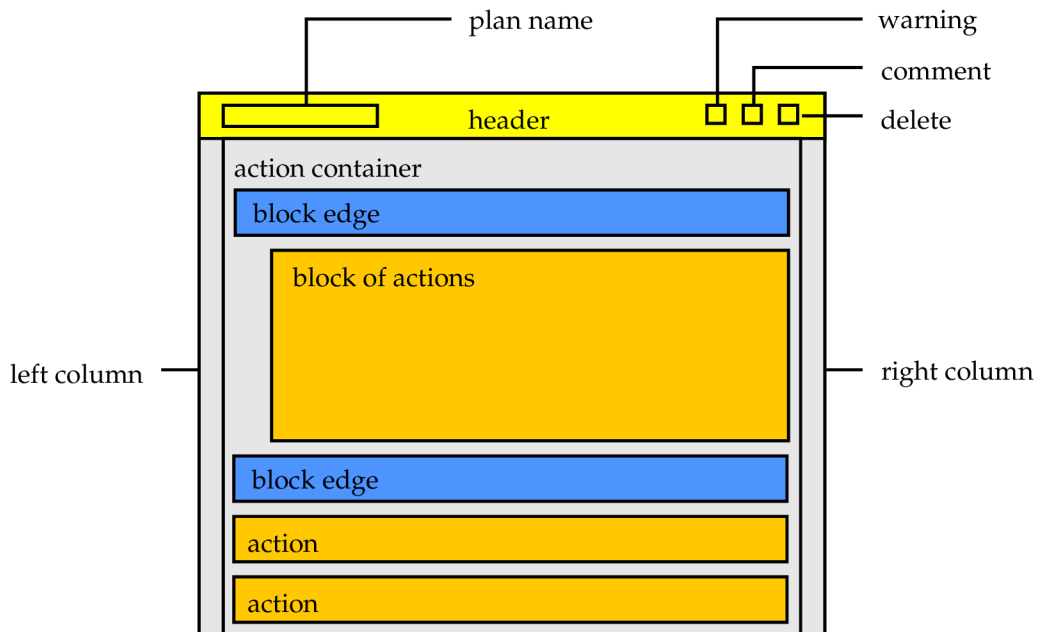
- right column and

- action container.

Figure 3.5: Graphical representation of a plan

Header contains further subcomponents e.g the name of the plan is displayed on the left side. Name has to be displayed in an editable way, because plans can be renamed. Buttons on the right side of the header intend to:

- display warning messages,

- display a comment box and

- delete the selected plan.

Left and right columns are used to display information specific to individual actions. The action container arranges actions of the plan. Every action is displayed on a separate row in order to keep the visual code clean. One of the main tasks of the action container is to display structured actions correctly. Typical structured actions are the „*direct run*“ *(@)* action and high level abstractions. Structured actions defines blocks. Actions within a block are displayed with indent for better readability.

There is an additional visual description layer above the individual plans, namely the plan base. Plans do not stand on their own with no connections to the others. Action „*indirect run*“ *(ˆ)* is used to load actions to the action stack from the plan specified by the passed parameter. Visually it can be described as a jump to the target plan. Plans of the plan base are interconnected by jumps leading from „*indirect run*“ actions to plans. If the target plan is the same as the source, we speak about recursive jumps. If the passed parameter is a register symbol, no connection is drawn because the content of the addressed register is not available during editing. There is no way to tell without debugging on which plan the action points.
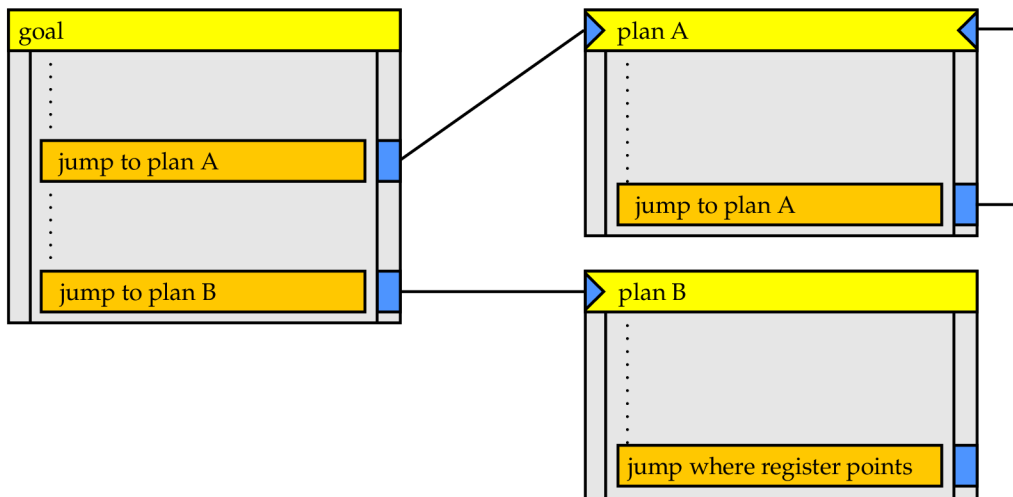

Figure 3.6: Graphical representation of a plan base

Connections are drawn to a layer that is below the layer of plans. It helps to keep the graphic representation manageable and perspicacious, because the connection lines do not interfere with the content of plans.

3.4.3 Graphical source code editing

The graphic source code editor supports not only visualization but editing as well. Editing composes from the following tasks:

- move the selected action within the plan base and

- edit parameters.

Parameters are represented by predefined components available in Java. Combo boxes, check boxes, text fields and other items are used. Their content is initialized according to

the inner representation of the source code. The inner representation of the source code is updated whenever user interacts with these components and changes the value they hold. If incorrect data are entered, the update is not executed, and the graphic user interface displays a warning to let the user know that something is wrong.

New actions are added by using a so called empty action. Every plan including the goal contains an empty action. The newly added action appears in place of the selected empty action, while it is pushed one position further. Even if there is a number of plans displayed on the screen only one empty action can be selected at a time. There are some other restrictions too e.g. empty actions can not be exchanged between plans; every plan contains exactly one. Of course empty action has no textual representation, because it does not exist in ALLL; serves only editing purposes.
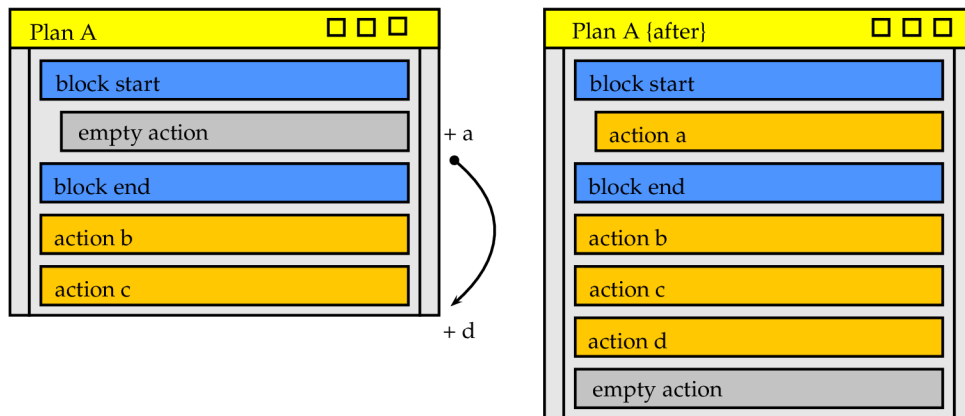

Figure 3.7: Adding new actions

One of the most natural way of modifying things on a computer screen is to drag and drop them. With this in mind actions can be repositioned, exchanged between plans or deleted by dragging and dropping. The position of the dragged action at the time of release defines how to respond to the drop event. If the action is dropped above the empty space between the plans it means deletion, otherwise the action is placed to its new position. Inner representation of the source code is re-generated according to the new graphic definition.
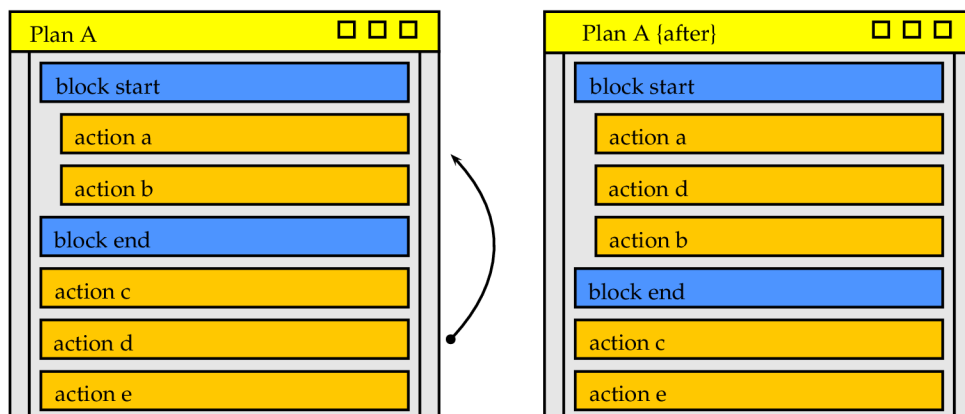

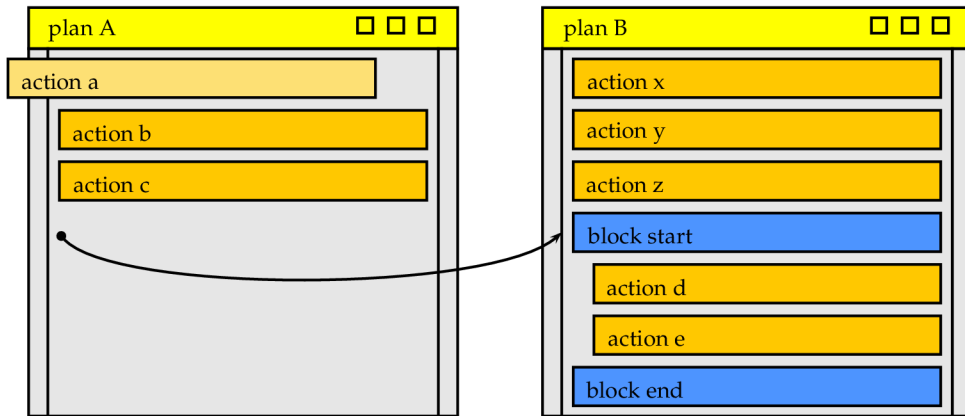Figure 3.8: Change the order of actions

Figure 3.9: Drag and drop actions

Plans are not represented internally as a one-dimensional array of actions. A structured action can contain other actions, because of this a tree like structure is required. A structured action is displayed as a block of actions enclosed by edges. Actions can be moved into or move out from a block in two ways. The first option is to drag and drop the action directly. The other one is to move one of the edges. It is important to control that a block edge can not be moved into an incorrect position in which distinct blocks are overlapping. The starting edge of a block represents the whole structured action. If it is deleted or dropped into another plan, the whole block of actions is moved. It is important to update parent-children relations of structured actions after a change.
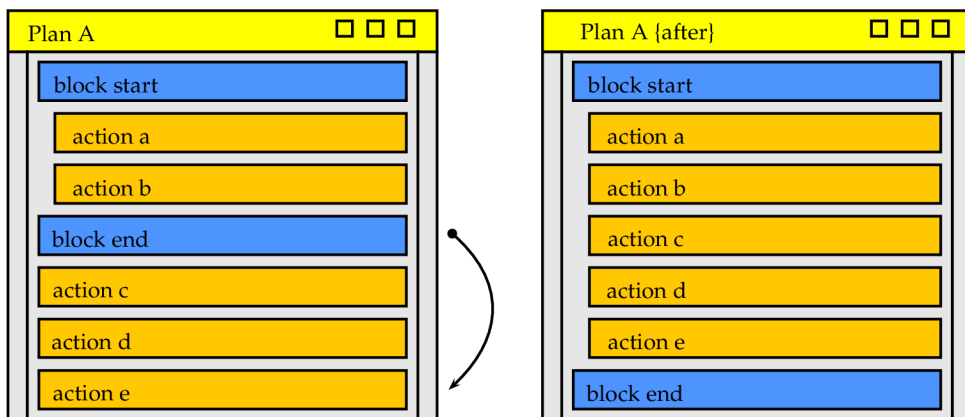

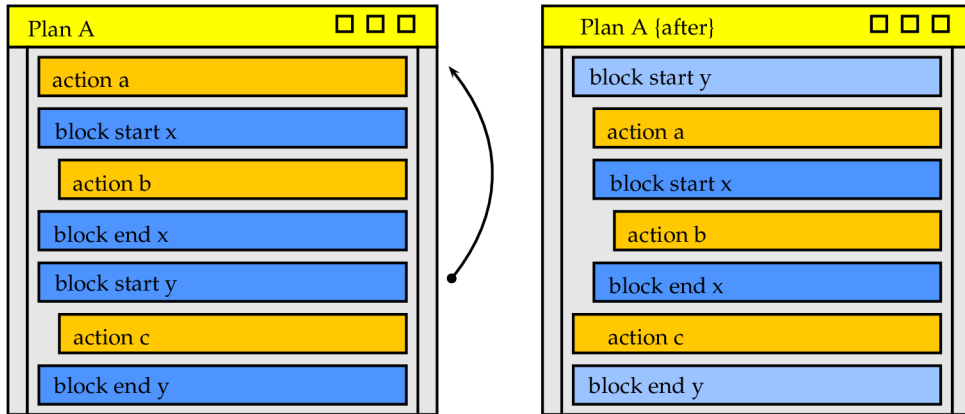Figure 3.10: Re-structure actions by moving block edges

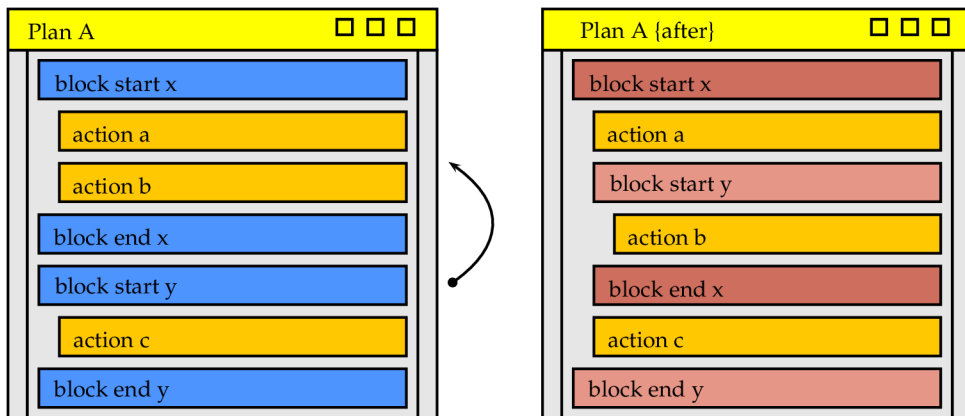Figure 3.11: Example, correct position of a block edge

Figure 3.12: Example, incorrect position of a block edge

3.4.4 Colour codes

Using colour codes to express differences between elements is are very useful approach, especially in a graphic environment. Agents during debugging can change their state. Following colour codes represent possible states of an agent.
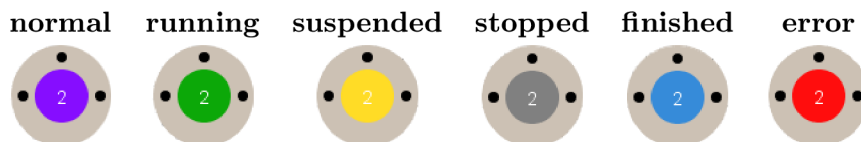

Table 3.1: Colour codes of agent states

A purple colour is used by default in the topology editor (no debugging). Green tone represents a running agent. Yellow color is used to express, that the execution of the agent

is suspended until a message arrives. Agents painted in gray are stopped, for example because of calling the move service with the „stop" parameter. A blue tint notifies that the action stack of the agent has emptied normally while red tells us that is has emptied because of a failed action.

Colour codes are also used in the source code editor. Actions are painted using the following colour set.

default	structured	selected	succeeded	failed	suspended	auxiliary

Table 3.2: Colour codes of actions

Simple actions and services have orange color. Edges of structured actions are painted using a blue tone. If the user clicks on one of the edges, they turn into a lighter colour to visually express the selection and the boundaries of the structure.

The rest of the colour palette is for debugging. Successfully executed actions turn to green (and back into the their original colour as the execution moves forward). Actions that failed for some reason, for example because of invalid input parameters, are repainted in red. Service that suspends execution of the agent until a message arrives turns to yellow. Auxiliary color code is used at conditions. Abstractions are decompiled into a sequence of ALLL actions. Those are executed one by one. Action „if" turns to green in case of a true condition. Red color is used to express a false condition or that the mathematical operation inside the abstraction got invalid parameters. A tone of brown color is used when an auxiliary action of the actual abstraction is executed.

3.4.5 Dynamic GUI and the logic program model

In our implementation there are two different kinds of user-graphic interface interactions. Some of them can cause a change in the model state while effects of the second type are restricted to the graphic domain. Removing an action from a plan for example, changes the model state, but re-arranging GUI plans on the canvas does not.

The logic domain and the graphic interface are sharply separated. Classes of the logic domain are collected into the package called `logic` while elements of the graphic user interface are in a separate package called `gui`. Few selected GUI objects play the role of a controller.

Visualization of language elements is solved by GUI objects while the logic model is defined by different objects. There is a mirroring between objects of the logic and the graphic domain. GUI objects realize only the graphic representation while objects of the logic model store the information which is presented. It helps to reduce information redundancy since the same information is not stored twice There is a 1:1 relationship between the two kinds of objects. Their hierarchical structure is described on the picture below.
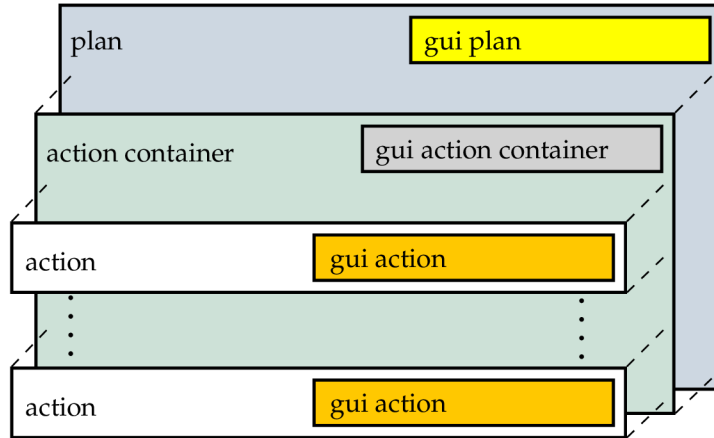
Figure 3.13: Object hierarchy example

Graphic interface objects belong to their logic model equivalents. They are created and disposed together. When a GUI object wants to display some information about the model, for example the actual value of a parameter, it can ask the logic action to return it. It was a design principle to define object interactions in a way, that the logic model does not event now that it is being visualized. Only the graphic representation is aware about both of the domains. One of the main advantages of this solution is that there is no need to constantly synchronize the logic model with its view. In most of the cases the graphical representation is simply rebuilt according to the current state of the logic model.

Chapter 4

Implementation

This chapter tends to present the implementation of the ALLL graphic development environment. This part describes solutions to problems presented in the previous chapter. The following topics are discussed:

- language recognition,

- model-view-controller design pattern,

- logic program model and the

- graphic user interface (GUI).

4.1 Language recognition

Source code written in ALLL is compiled into an inner representation using the following language recognition tools generated by the ANTLR framework:

- lexer (class `ALLLLexer`),

- parser (class `ALLLParser`) and

- abstract syntax tree walkers.

Two distinct tree walkers solve the following tasks:

- semantic control (class `SemanticTreeWalker`) and

- building inner representation of a source code (class `ReaderTreeWalker`).

4.1.1 Recognition exception handling

The lexer and parser tools are generated according to the rules defined in the `ALLL.g` grammar file, and perform the first two phases of the language recognition process. During lexical and syntactical analysis several errors can occur. ANTLR has a very sophisticated error handling and reporting system. Basically there are two kinds of a language recognition error in this framework:

- The first type is from which a recognizer can recover by guessing what the problem might be. The recognition continues only a warning is displayed to the user.

- In the second case recognizer fails to recover and recognition stops.

In order to use custom error handling, defined in our class **Error**, some parts of the generated lexer and parser need to be replaced with our code. The following code snippet is added to **ALLL.g** to override the function displaying recoverable errors.

```
@lexer::members{
    public void displayRecognitionError(
        String[] tokenNames,
        RecognitionException e){
            Error.addLexicalException(e);
    }
}
```

Unrecoverable errors can be overridden by adding the following code to the grammar file.

```
@lexer::rulecatch{
    catch (RecognitionException e){
        Error.addLexicalException(e);
    }
}
```

Prefix **lexer::** in the previous examples notifies that the code is intended for the lexer. Without the prefix functions of the parser are overridden by default.

Output of the parser is the abstract syntax tree (AST) of the program. Tree walkers take the AST generated by the parser and walk through it. By adding code snippets to the tree walker it can become a translator an interpreter or can implement any kind of a desired functionality. Tree walkers are defined by tree grammars. In this project tree walkers are used for semantic error handling and creating the inner representation of the source code.

4.1.2 Semantic control

Some of the control mechanisms is moved from the parser to the semantic control defined in a separate class called **Semantics**. The idea behind that is to keep the parser flexible and more general. Lets take an example. The service which controls the LEDs can accept only certain values as the color parameter. The parser controls only whether the parameter is a string and the supported values („r"„g"„y") are controlled subsequently by the semantic tree walker. It ensures, that support for additional LED colors can be added only by redefining the adequate semantic control, without making any change to the parser. On top of that semantic control is inevitable because of „jumps" within the plan base. Controlling the existence of the targeted plan is beyond the power of the parser.

The following semantic controls are executed by the semantic tree walker:

- Every „indirect run" (ˆ) action refers to an existing plan.

- Valid register id (1-3) is used when a register is referenced.

- Valid state (0,1 or none) and color („r" „g" „y") parameters are passed to LED control services.

- A supported calculation mode („a" „m" „M") parameter is passed to data collecting services.

- Tuples in the input base have valid source addresses (integer - message, „s" „a" „m" „M" for sensor data) .

Semantic control was designed to be expansible. New control mechanisms can be defined easily in three steps:

1. Create a new exception class derived from `ALLLSemanticException` to describe the new semantic error.

2. Implement the logic in a new static method of the `Semantics` class. In case of an error the method throws the exception defined in point 1.

3. Add code snippet to the semantic tree walker to call the new method of the `Semantics` class.

4.1.3 Compiling source code

Inner representation of the source code is built by the reader tree walker. Object representation of agents is assembled by calling static functions of the `Reader` class in a specific order. The reader tree walker is responsible for the following actions:

- For each recognized language element create an object for inner representation.

- Define connections between the created objects: actions belong to plans, plans to a plan base and plan base to an agent.

- Recognize high level language structures in a sequence of ALLL actions.

4.2 Project files

Definition of the modelled multi agent system can be saved as a project. Project attributes are stored in a XML document. Behavioural and state descriptions of individual agents are stored in separate files with an `.alll` extension. It is beneficial, because agents can be easily imported into another project. An agent description has the following structure:

```
(plan base)(goal)
reg:(initial content of the 1st register)
reg:(initial content of the 2st register)
reg:(initial content of the 3st register)
bb:(belief base)
ib:(input base)
```

The XML document stores, among others, information about the network topology, details about the plan base visualization, comments and agent source code files are linked too. It is important to use CDATA sections where it is required. Start of a character data section indicates to the XML parser that the following content do not need to be parsed. A property which can contain arbitrary characters, e.g. project description, is stored as character data.

4.2.1 Structure of the project info file

The project info document is structured as follows.

```
<project>
    <properties>
        <title>[!CDATA[{project title}]]</title>
        <description>[!CDATA[{project description}]]<description/>
        <nodecounter>{node counter}</nodecounter>
        <agentcounter>{agent counter}</agentcounter>
        <sink x="{x-coordinate}" y="{y-coordinate}"/>
    </properties>
    <nodes>
        <node ord="{node number}" x="{y-coordinate}" y="{y-coordinate}"/>
        ...
    </nodes>
    <links>
        <link dst="{destination node}" src="{source node}"/>
        ...
    </links>
    <agents>
        <agent file="{source file}"
               name="{agent name}"
               class="{agent class}"
               ord="{agent number}"
               node="{initial position}">
            <plan isgoal="true"
                  name="_goal"
                  x="{x-coordinate}"
                  y="{y-coordinate}"/>
            <plan isgoal="false"
                  name="blick"
                  x="{x-coordinate}"
                  y="{y-coordinate}">
                <comment>[!CDATA[{comment}]]</comment>
            </plan>
            ...
        </agent>
        ...
    </agents>
</project>
```

4.2.2 Reading the project info file

The first thing we need to solve, in order to load projects, is parsing XML documents. Language Java supports several application programming interfaces (API) for XML processing. I have decided to use the xpath package. XPath uses path expressions to select nodes or node-sets in an XML document. It is a more convenient than sequential processing for example.

The validity of the loaded information needs to be verified. The following controls are executed:

- In network topology sink (eraser) can not be positioned initially over any of the nodes.

- Link items can not connect non-existing nodes.

- Agents can not be added to non-existing nodes in the network.

- Agent definition source files have to exist.

- Plans referenced in the project file have to exist in the model.

If the requested information is not found in the document or any of the previous controls fails, an `CorruptedProjectException` is thrown and loading is interrupted.

4.3   Logic program model

The logical program model defines the inner representation of a real life sensor network including the agents. The reality is transformed into an abstract model defining which attributes of real life entities are the most important for us, and how we work with them. This section tends to present the structural units of the logic program model using a bottom-up approach.

4.3.1   Hierarchy of action classes

Actions and services supported by ALLL are internally represented by separate classes. These classes are organized into a hierarchical tree structure. This hierarchy expresses which actions share certain attributes and functionality. Some action classes are abstract, they can not be instantiated.

Abstract class `ActionToR` groups actions that have exactly one parameter which can be a tuple or a register (abbreviation „ToR" stands for Tuple or Register). Classes derived from the abstract class `Service` are transformed into their textual representation with one extra step: services have to be wrapped into a „service call" ($) action. Next to simple actions there are structured actions as well. Actions „direct run" (@) and condition „if-then-else" group actions into blocks. Abstract class `Condition` allows to extend the currently supported if-then-else structure with additional structures e.g. „while" or „for" cycles. Class `ServiceLoR` play a very similar role as class `ActionToR`, except that services „first" (f) and „remainder" (r) accept as a parameter a register symbol or a tuple, which can not contain register symbols only integers, string values and unificators.

4.3.2   Inner representation of actions

Action objects represent ALLL actions or services. Services are called using a special action „service call" ($), but from our point of view it is irrelevant. Terms action and service are used interchangeably in this text, as they both refer to an executional unit of a ALLL program.

The most important part of an action is its parameter list. Major class of ALLL actions has only one parameter, but several actions require more parameters. ALLL allows in most of the cases to pass parameters through a register reference. So, we need to be able to define whether we want to use as parameter a register or a specific value. Another example

to alternative parameters can be the *„test input base" (?)* action, which accepts three different kinds of a parameter: integer value, register or a unificator. These alternatives are modelled as components. Parameter has a list of components which represents the selection of available alternatives. Exactly one component is active at a time and it defines the currently used parameter type. Some components have predefined values while others accept input from the user. For example register components allow to switch between registers 1-3, but tuple component waits for the user to enter a value. These components are reusable in different actions.
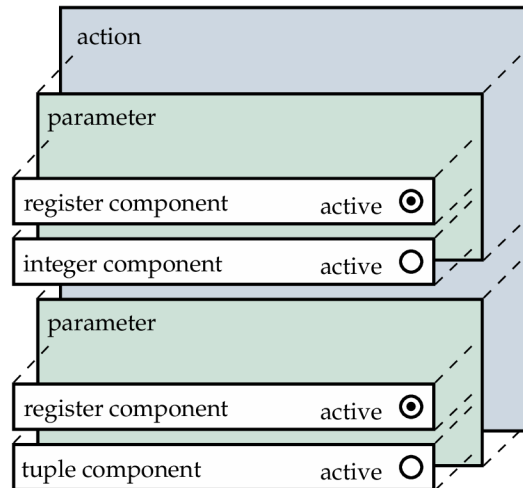

Figure 4.1: Inner representation of an action

Parameters and components play an important role in decompiling the inner representation of an action to ALLL code snippet. The textual representation of an action is defined by the type of the action and the parameter list. The textual representation of a parameter is defined by the currently active component and the value what holds.

4.3.3 Inner representation of agents and nodes

The inner representation of agents copies the general structure of agents. It has a belief, input and plan base and a goal, however it has an additional unit as well. Virtual machine is used to execute actions. Virtual machines can be divided into two major parts: to a set of registers and an action stack. State of the action stack defines which actions in which order going to be executed. Debugging starts with loading actions of the plan to the action stack. The top of the stack points to the currently executed action, it is like a program pointer. I have decided to move the virtual machine from the sensor node to the agent, because when an agent is cloned the state of the action stack needs to be cloned as well. Actions are loaded to the action stack from the plan base, so cloning of the plan base is closely related to the action stack. That is the main reason why machines executing agent source code are part of the agents.

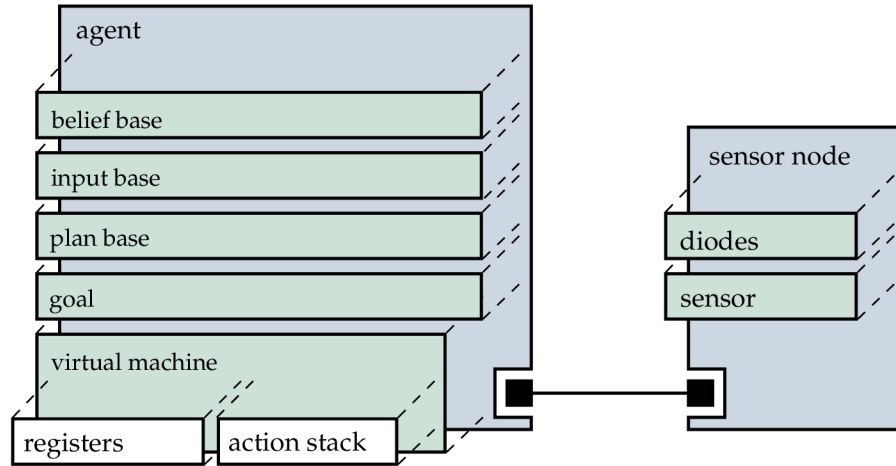Figure 4.2: Hierarchy of action classes
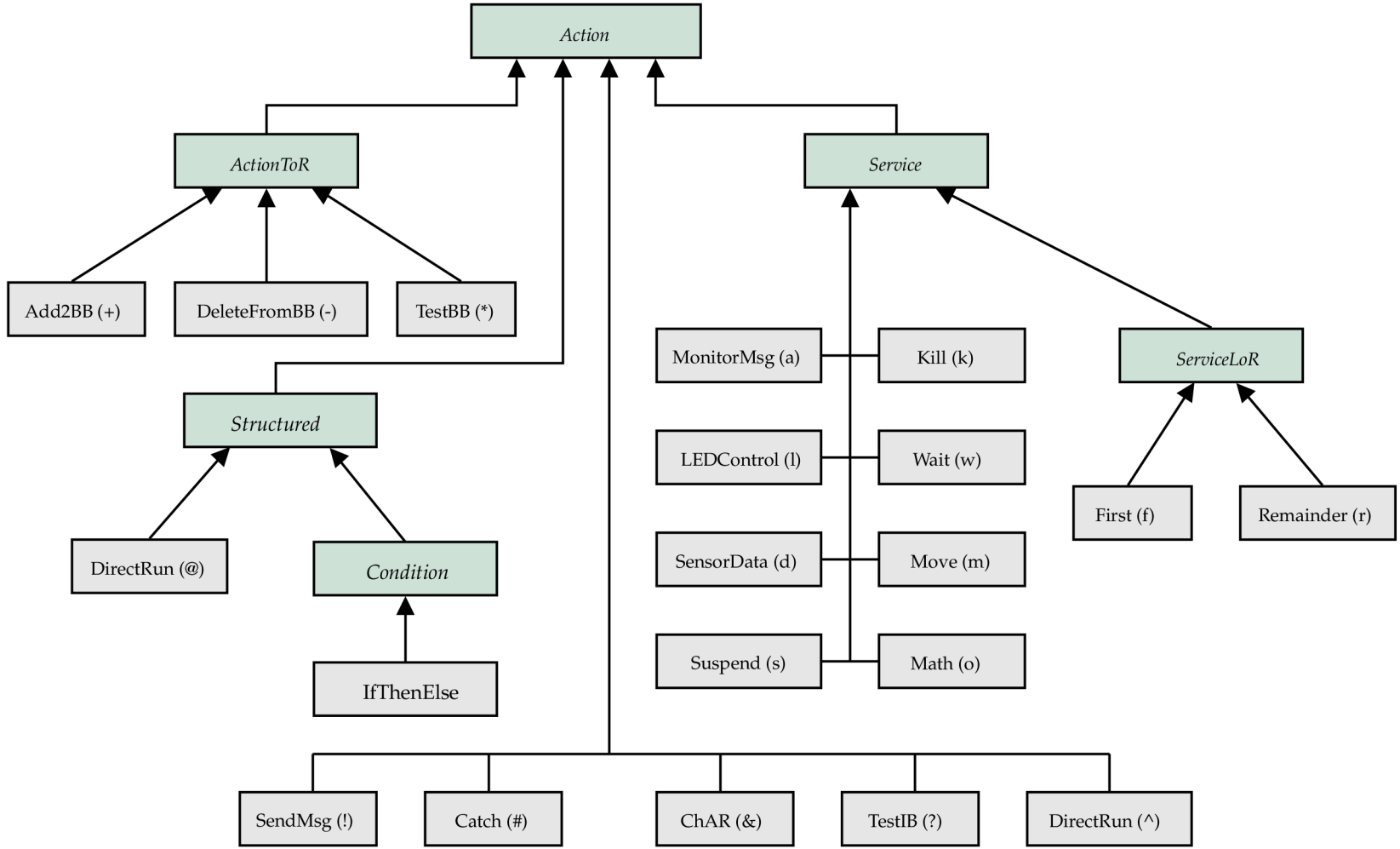
Figure 4.3: Structure of an agent - sensor node object pair

Sensor nodes have relatively simple structure, they contain LEDs for signalization and a sensor for data collecting. The context of the currently seized node needs to be passed to those actions that manipulate with them. It is done trough the connection that exists between the agent and its host node. This connection is bracken up when agent moves to another node.

4.3.4 Inner representation of a multi-agent system

The inner representation of a multi-agent system can be divided into the following parts:

- topology,

- list of agents,

- auxiliary variables for debugging.

The topology defines how many nodes the system contains, and how are those node connected. The initial position of agents is also part of the topology. The auxiliary variables are used to restore pre-debug state of the system. The state of the multi-agent system before and after the debugging can differ. Number and position of agents can change. New agents can born by cloning and working agents can be kicked-out from their node by other agents. Auxiliary variables keep track of these changes. A new record is created for every new agent, and kicked-out agents are stored in a recycle bin. By using these informations the initial state can be restored without re-generating the whole system.

The multi-agent system also implements an interface towards actions. Message sending and agent move require knowledge about the topology. The multi-agent system can tell whether the agent can be moved or the message can be delivered to the target node.

4.4 Dynamic graphic user interface

In the followings some of the implementation details of the dynamic graphic user interface will be presented. The most important classes of source code visualization and editing are described below.

4.4.1 Class GuiAction

Class GuiAction is derived from class `JPanel` and serves as a container for action parameters. `GuiAction` is an abstract class. The differences in presentation of individual actions are handled by derived classes. Basically the content is displayed by collecting components from the parameters to a list and items of this list are organized using a `FlowLayout`. Some actions, e.g. binary mathematical operation or message sending, require more sophisticated arrangement of items, primarily because there is to many of them. In this case the `FlowLayout` is replaced by a `SpringLayout`.

The minimal width of an action is a sum calculated from the minimum width of components plus padding. Naturally it can be overridden in derived classes where specific arrangement of components is used. Even one action can have different minimum width depending on the currently selected parameter. Alternative parameters are displayed using my custom `ParameterSelectPane` which is very similar to a combo box, but instead a textual value it holds components. It behaves as an abstract component, and its minimal width is based on the current selection.

4.4.2 Class GuiPlan

The graphic representation of plans was implemented according to the design presented in the previous chapter. Instances of the `GuiPlan` class serve as graphic user interface objects for logic domain plans. Different subcomponents e.g. header, side columns, action container are derived from the `JPanel` class as well as the whole graphic plan. Subcomponents are organized using a `SpringLayout`, which allows to define positions using relations instead of constant values. The action container uses a `BoxLayout` to display individual actions on top of each other. Indent used to display blocks of actions is solved by creating custom empty borders around the actions.

Figure 4.4: Snapshot of a GUI plan

Plans are always displayed on the smallest possible surface. Height is primarily defined by the number of actions within the plan. When a new action is added or an existing

37

is deleted, size of the plan is re-calculated. Calculating the minimum width of a plan is a bit more complicated, because editing action parameters can change it too. Displaying plans with correct dimensions is ensured by overriding the `getMinimumSize()` method of the `Component` class (class `JPanel` is derived from `Component`).

4.4.3   Class Canvas

Class `Canvas` extends JLayeredPanel and displays the plan base and the goal of one selected agent. Distinct layers are used to display different groups of objects. From the bottom to the top layer they are the followings:

- background,

- connections,

- plans and

- pop-up boxes.

Layer plan is actually a group of layers, because each plan is assigned to a different layer. This organization allows to the order them completely freely. Pop-up boxes are internal frames added to the `POPUP_LAYER` of the canvas. Two different boxes are used, one for displaying user comments, another to create a new plan.

Class `Canvas` plays an important role in dragging and dropping actions. On the one hand canvas ensures the dragged action to be above the layer of plans, on the other hand canvas can tell above which plan the dragged action was released. It is important to place the dragged action above the layer of plans, otherwise actions from lower layers could be hidden by plans of higher layers during dragging.

4.4.4   GUI during debugging

To prevent code editing during debugging graphic user interface has to react differently to user interactions during execution. Precautions are twofold. On the one hand mouse handler methods do not react during debugging. On the other hand components of GUI actions are disabled.

State of the multi-agent system, whether it is being debugged or not, is provided by the `isDebugging()` static method of the `Project` class. All `JComponents` (text boxes, combo boxes, etc.) used for visualization can be disabled in one step by calling the `setEnabled(false)` overridden method of the `GuiAgent` class. Disabled `JComponents` in Java are repainted using tones of gray with low contrast by default. Information displayed by those components is hard to read during debugging. This problem can be solved by initializing the native `UIManager` Java class. User specified colors can override the default ones.

4.5   Agent cloning

Cloning an agent is one of the problems that had to be solved effectively. It is possible to clone objects in language Java, but the returned clone still shares its inner structure with the original one. Class `AgentFactory` implements deep cloning of agents. The most

demanding subtask of agent cloning is to clone actions. Actions are cloned by creating new actions using parameters of the original action in the constructor.

The problem is that, because of better maintainability, every action is defined in a different class. First we have to find out which constructor we want to call. Reflection is used to find the correct one. Reflection is a feature in the Java programming language. It allows an executing Java program to examine or „introspect" upon itself, and manipulate internal properties of the program [7]. Objects in Java can have several constructors. Reflection can return the desired one if we specify the list of parameter types. To simplify things every action can have only one constructor, and parameters are defined in the most general way, by using the `Object` class. The following source code sample demonstrates constructor creation.

```
Constructor ct = null;
Class runtimetype[] = new Class[1];
/* Get the class */
Class cls = action.getClass();
/* CONSTRUCTOR WITHOUT PARAMETERS */
if (action.getParameters().size() < 1){
    Constructor cts[] = cls.getConstructors();
    if (cts.length > 0){
        /* Default constructor e.g. ServiceKill() */
        ct = cts[0];
    }
/* CONSTRUCTOR WITH PARAMETERS */
}else{
    /* Get parameter types */
    ArrayList<Class> partypes = new ArrayList<Class>();
    for(int i = 0; i < action.getParameters().size(); i++){
            partypes.add(Object.class);
    }
    /* For example ActionSendMsg(Object, Object) */
    ct = cls.getConstructor(partypes.toArray(runtimetype));
}
```

If a structured action is cloned its children have to be cloned as well. During the process of agent cloning the same action can be met several times, for example because of child-parent relationships. The same action has to be mapped to the same clone. Agent factory saves every cloned action to a `HashMap<Action,Action>`, where the key represents the original action and the value the cloned one. When a new action is passed for cloning, this map is controlled first. If the action has been previously cloned the value from this map is returned. Otherwise the process of cloning continues and the result is saved.

The cloned agent is a perfect and autonomous copy of the original agent. Right after the cloning they are in the same state, have the same behavioural description, but because they both continue to run on different nodes it can change very quickly. The behavioural description remains the same, but the knowledge they collect about their environment will probably be different.

Chapter 5

Tests

Software Testing is the process of executing a program or system with the intent of finding errors. [8] Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. [4] Testing phase of our graphic development environment (GDE) can be divided into 3 different steps.

The first phase have run parallel with the implementation. In this phase the program have been tested on its own. The aim of these tests was to verify that the GDE conformances design requirements. A typical examples can be:

- Export and subsequent import of the same source code in order to verify correct functionality of compiling and decompiling.

- Export and subsequent import of if-then-else abstractions in different forms (with / without else branch, nested conditions) to test abstraction recognition.

- Test of the dynamic graphic graphic user interface: moving, deleting actions, creating new plans, etc.

The second phase includes comparison of outputs from the GDE and from the WSageNt interpreter. Typical example can be the test named „Blink", which is presented below. Of course, it is only one of the many tests of the second phase. Implementation of the semantic meaning of individual actions have been tested exhaustively.

The third phase represents testing on real life sensor networks. Tests „Remote sensor" and „Travelling agent" (presented below) have been created in order to verify correct behaviour of the GDE debugger.

During tests a number of bugs and errors have been revealed. All of them have been fixed. Nevertheless it is fairly improbable to find all of the problems only by running a few tests. Although I am assured that tests presented in this chapter helped to improve correctness and reliability of the created development environment.

5.1 Test no. 1 - Blink

Test „Blink" is a simple test for verifying correct functionality of the source code editor and the debugger of GDE. Working with the belief base and the LED control services are tested primarily. The simplest topology is used for this test, only one agent in one node is required. The source code of the agent has been adopted, with some minor changes, from [11] (page 27).

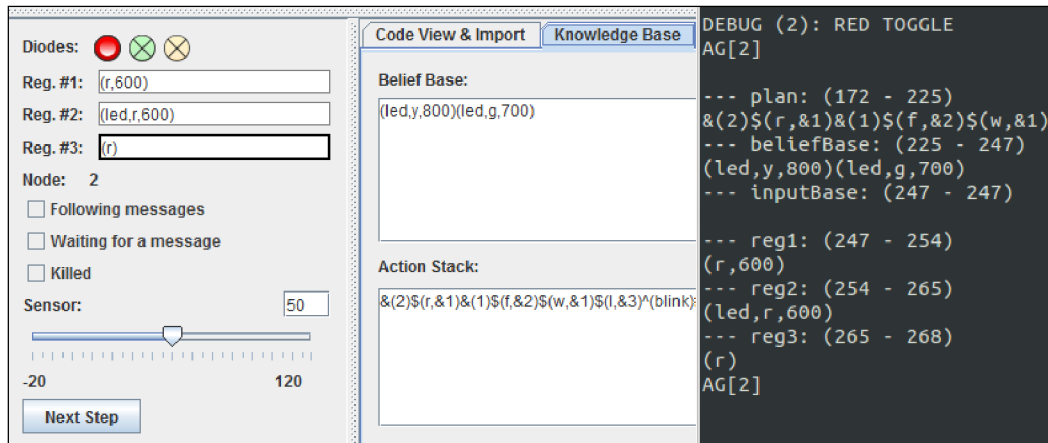| Agent	`blink`

Plan Base	`(fill,(+(led,r,600)+(led,g,700)+(led,y,800)`
	 `^(blink)#^(fill)))`
	 `(blink,(&(1)*(led,_,_)&(2)$(f,&1)-&2&(1)$(r,&2)&(3)`
	 `$(f,&1)$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)`
	 `^(blink)))`

Goal	`^(fill)`

Table 5.1: Agent „blink"

Figure 5.1: Outputs produced by the debugger and the WSageNt interpreter

This program fills the belief base of the agent with tuples that define, which diode and for how long (in ms) should be turned on. After executing all the LED control services the belief base remains empty, but the cycle is restarted from the beginning, and the execution continues.

This test is based on comparison. Model state changes caused by debugging were compared to the output of the WSageNt interpreter running the same code. State trajectories matched, so, the debugger has passed this test.

5.2 Test no. 2 - Remote sensor

Collection of programs called „Remote sensor" serves for testing data collection, message sending, synchronization and correct functionality of the if-then-else abstraction. A very simple topology, line of two nodes, is used. There are two agents in this system: the „sensor" and the „display". One of the agents („sensor") measures a value. The input received from the sensor is sent to the address defined in the belief base. After that execution of the agent is suspended. In this state „sensor" is waiting for an acknowledgement from the „display". Agent „display" receives the message and processes the input value. If the received value is less then a predefined threshold, the green diode, otherwise the red diode is turned on. After that an acknowledgement is sent back. This signalizes to the „sensor" that agent „display" has finished and is waiting for the next value to process.

Agent	sensor
Plan Base	`(address,(&(1)*(address,_)&(3)$(f,&1)&(2)$(r,&3)))`
	`(cycle,(^(measure)^(send)^(cycle)))`
	`(measure,(&(1)$(d)?(_)))`
	`(send,(!(&2,&1)^(ack)))`
	`(ack,(&(1)$(s)?(_)))`
Goal	`$(a)^(address)^(cycle)`
Belief Base	`(address,2)`
Node	`3`

Table 5.2: Agent „sensor"

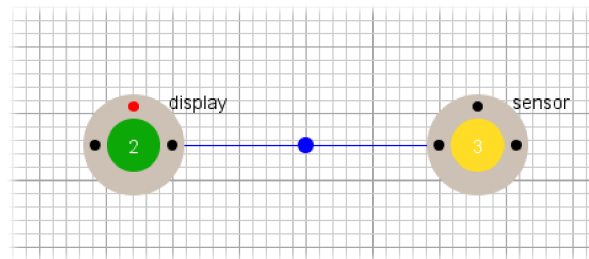| Agent	display
Plan Base	`(cycle,(&(1)$(s)?(_)^(blink)^(cycle)))`
	`(blink,($(l,(g,0))$(l,(r,0))&(2)`
	`$(o,les,&1,(560),(),(),(cond))+&2`
	`@(*(((cond),1))-(((cond),_))&(2)$(l,(g,1)))`
	`@(*(((cond),0))-(((cond),_))&(2)$(l,(r,1)))!(3,(next))))`
Goal	`$(a)^(cycle)`
Node	`2`

Table 5.3: Agent „display"

Figure 5.2: Test no. 2 - snapshot from the GDE

Testing on a real wireless sensor network took place as follows. Sensors were set to measure temperature. The two IRIS motes were turned on. Agent „display" had been loaded into node number 2 first, after that, agent „sensor" to node number 3. Almost immediately, as the base station has finished to load the second agent, green diode on the 2nd node turned on. Blinking of the diode showed that there is a continuous communication between the two nodes. The first phase of the test finished successfully. In the second phase the „sensor" had been several times drawn close to a heat source (ventilation-blower of a notebook) and put away. When the sensor node was close to the heat source, the red diode turned on at the remote node. If the sensor has been away, after a short time (needed by the sensor to cool down), „display" switched to green. The pair of „sensor" and „display" agents worked as it was expected.

The value used as a threshold between „green“ and „red“ states had been selected before the actual test, and its actual value (560) was based on several test measurements.

5.3   Test no. 3 - Travelling agent

The aim of the third example is to test agent moving. A mesh topology is used. Tester network consists 4 fully connected nodes. Initial position `(pos,(2))` of the agent is loaded the the belief base before the start. Traveller agent „jumps“ from one node to another based on the predefined path specified by action `+(path, (3,4,5,2))`. Plan `savepath` saves the remaining path to the belief base, while plan `loadpath` loads it as a list of addresses to the 1st register. Head of this list is transferred to the 2nd register and service „*move*“ *(m)* is called. Move service is invoked with the additional „stop“ parameter. This ensures, that there is always only one active agent in the network. Agent informs us about its current position by blinking the green diode once. Agent „traveller“ moves along a closed circuit. When the starting position is reached again, plan `savepath` saves the previously travelled path to belief base, and the cycle is re-started.

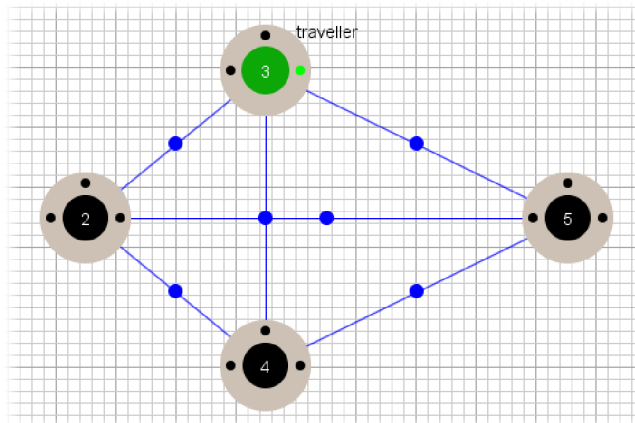| Agent	traveller
Plan Base	`(cycle,(^(savepath)^(loadpath)^(jump)^(cycle)))`
	`(savepath,(@(*(pos,(2))+(path,3,4,5,2))-(pos,_))`
	`@(&(3)*(pos,_)&(3)$(r,&1)+(path,&3)-(pos,_))))`
	`(loadpath,(&(1)*(path,_)&(2)$(f,&1)&(3)$(r,&2)&(1)`
	`$(f,&3)-(path,_)))`
	`(jump,(&(2)$(f,&1)$(m,(&2,s))$(l,(g,1))$(w,(300))`
	`$(l,(g))+(pos,&2)))`
Goal	`^(cycle)`
Belief Base	`(pos,(2))`
Node	2

Table 5.4: Agent „traveller“

Figure 5.3: Test no. 2 - snapshot from the GDE

43

The actual test started by turning on the four IRIS motes and arranging them in a line according the predefined path programmed into the agent. Agent started at node no. 2. Blinking green diodes showed the path of the agent. Agent moved as it was expected, when it reached the last node, agent jumped right back the the first one. After a few cycles the test had been considered as successful and had been finished.

Chapter 6

Closure

The aim of this thesis was to design and implement a fully functional graphic development environment (GDE) of Agent Low Level Language. First of all I had to study this specific agent language, after that I could start to implement the solution.

The created environment supports the following tasks. Any source code written in ALLL can be imported into the environment, where the visualized form of the program can be edited and exported back into ALLL. The environment supports not only native ALLL instructions but the „if-then-else" abstraction can be used as well. Projects help to organize our work. A project wraps a number of agents and the specified network topology into a multi-agent system. Projects can be saved and opened just like in any other development environment. Topology editor of the GDE serves for modelling real life sensor networks. Last but not least the designed system can be examined during execution using the debugger tool.

Tests presented in the previous chapter prove that this implementation is usable. However a number of possible improvements could be added e.g. support for additional services: neighbour discovery, footprints of agents, etc. The number of supported abstraction needs expansion as well. Structures like „while", „for" and „switch" could be integrated next to „if-then-else". Examining possibilities of ALLL source code optimalization would be an interesting topic too.

After experimenting with the created environment, creating and debugging a few multi-agent systems, I found the program very useful. Especially when I compared the graphically displayed source code to its textual representation. It is a lot easier to understand the logic of the program from its visualized form. Debugger helps us to understand not only how the whole system works, but the specific semantic meaning of individual actions as well. I think this program could be used for educational purposes too. I would strongly recommend, for those who have just started to learn agent programming and ALLL, to start with this graphic development environment. I hope that with my work I contributed to this promising field of development.

Bibliography

[1] A. Meduna, R. Lukáš. Formálne jazyky a překladače. materials for course IFJ, 2008.

[2] Wikipedia article. Multi-agent system [online].
http://en.wikipedia.org/wiki/Multi_agent_system, [cit. 2011-11-21].

[3] Wikipedia article. Model-view-controller [online].
http://en.wikipedia.org/wiki/Model_view_controller, [cit. 2011-11-28].

[4] Hetzel, William C. *The Complete Guide to Software Testing 2nd ed.* Wellesley, Mass, 1988. ISBN: 0894352423.

[5] Horáček, J. Wsagent: Multiagent platform for wireless sensor networks [online].
http://www.fit.vutbr.cz/~ihoracek/WSageNt/, 2010 [cit. 2011-11-21].

[6] Kalmár, R. *Jazyk Vyšší Úrovně Abstrakce pro Programování Mobilních Inteligentních Agentů.* bachelor's thesis, Brno, FIT VUT v Brně, 2010.

[7] G. McCluskey. Using java reflection [online].
http://java.sun.com/developer/technicalArticles/ALT/Reflection/, [cit. 2012-03-15].

[8] Myers, Glenford J. *The art of software testing.* New York : Wiley, 1979. ISBN: 0471043281.

[9] Parr, T. *The Definitive ANTLR Reference.* USA, 2010. ISBN-10: 0-9787392-5-6.

[10] S. Buettrich, A. E. Pascual. Basic wirelles infrastructure and topologies [online].
http://www.itrainonline.org/itrainonline/mmtk/wireless_en/
04_Infrastructure_Topology/
04_en_mmtk_wireless_basic-infrastructure-topology_slides.pdf, 2006 [cit. 2011-11-23].

[11] Spáčil, P. *Mobilní Agenti v Bezdrátových Senzorových Sítích.* bachelor's thesis, Brno, FIT VUT v Brně, 2009.

Appendix A

Content of the CD

The enclosed CD contains the following structure of folders:

- `text:`
    - contains the text of this thesis in a portable document format (.pdf), and as a
    - LaTeX project (including graphic files),

- `program:`
    - contains Java source files of the graphic development environment,
    - ANT makefile and
    - users manual,

- `doc`
    - contains program documentation,

- `poster:`
    - contains the poster in a portable document format (.pdf).

Appendix B

Manual

Requirements:

- Java compiler.

- Apache Ant builder tool.

Required libraries are enclosed in directory „libs" next to the source files. To run the application follow the instructions. Copy the „program" directory into your hard disk, open the directory with a command line tool and type: `ant compile`. If the compilation finished successfully, type: `ant run`, to run the graphic development environment.

Appendix C

Poster

The poster was handed over in a paper form to the supervisor; the .pdf version is on the enclosed optical disk.