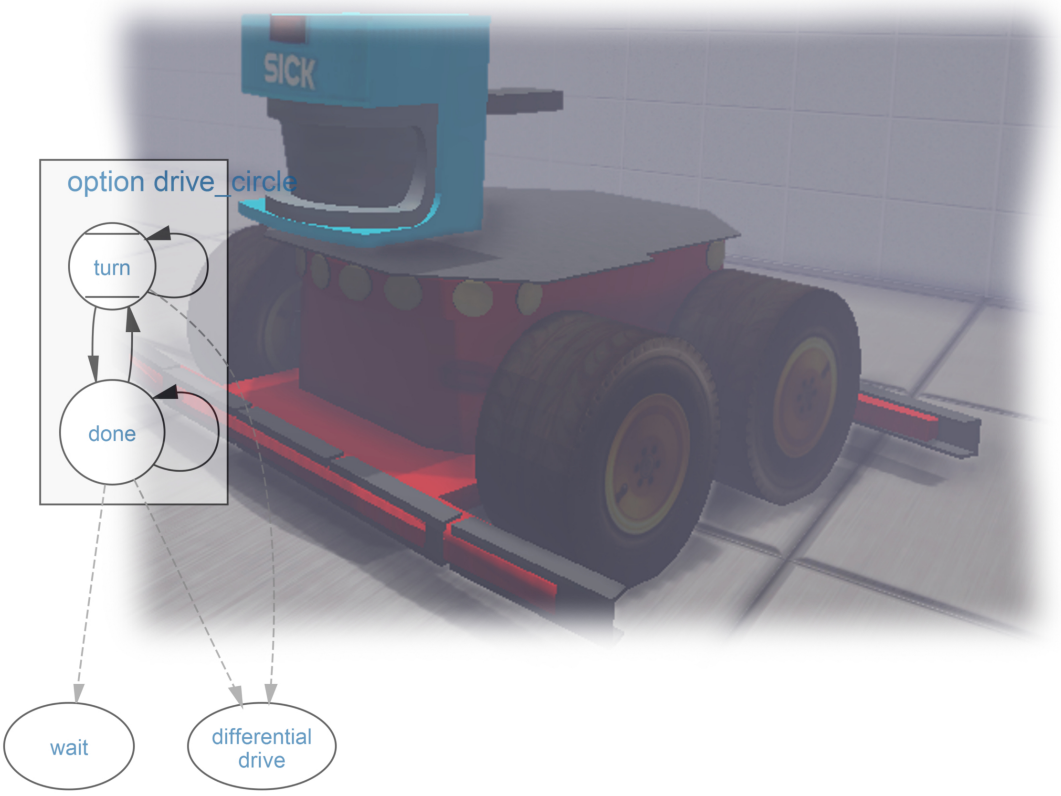
Combining Robocup Rescue and XABSL

Maarten P. de Waard



UNIVERSITEIT VAN AMSTERDAM

Bachelor project: Final Thesis

Combining Robocup Rescue and XABSL

Maarten P. de Waard
5894883

Bachelor thesis
Credits: 6 EC

Bsc. Artificial Intelligence

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
dr. A. Visser

Informatics Institute
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 26th, 2012

Contents

1 Introduction **1**

2 Behavior Based Artificial Intelligence **1**

3 BBAI Implementations and Alternatives **3**
3.1 XABSL . . . 3
3.2 POSH . . . 4
3.3 Petri Net Plans . . . 5
3.4 COLBERT . . . 7
3.5 Other alternatives . . . 7

4 Language of choice **8**
4.1 Advantages of Using XABSL . . . 8
4.2 Disadvantages of Using XABSL . . . 9
4.3 Creating an XABSL-specification . . . 9
4.4 Traditionally Combining XABSL With Any Existing Program . . . 11

5 RoboCup Rescue **13**
5.1 Description . . . 13
5.2 Alternative possibilities . . . 13

6 Approach **13**
6.1 Information flow . . . 13
6.1.1 Messages . . . 13
6.2 World Class . . . 15
6.3 Other Added Simplicity . . . 15

7 Results **15**
7.1 UsarCommander . . . 15
7.1.1 Driving a circle . . . 15
7.1.2 Walking a corridor . . . 15

8 Conclusion **16**

9 Discussion and future work **18**

Abstract

In this research, a product will be introduced, that combines the Extensible Agent Behavior Specification Language (XABSL) with any program capable of having a socket connection. A use of this product is shown, by combining it to the rescue project on the University of Amsterdam, using *UsarCommander*, a program designed to control one or more robots, in a virtual rescue operation in the simulator USARSim.
Keywords: XABSL, Behavior Based Artificial Intelligence, RoboCup, UsarCommander

1 Introduction

The research will be focussing on combining a specially designed Behavior Specification Language with any program capable of making a socket connection. In particular, the focus will lay on virtual rescue operations, otherwise known as the *RoboCup Rescue League*. Using a behavior specification language will make it possible to separate specification of behaviors from implementation.

At the end of the research, a product will be presented that can be inserted into any other program to take over the behavior specification. The method will be proven with UsarCommander, which is used by the Rescue team on the University of Amsterdam.

Currently, the focus of research in the rescue missions is mainly on creating smart implementations of sensors. Much of the actual controlling of robots is done by hand, using programs that forward the camera images of the robots to a human operator controlling them. Some of these operators use simple behaviors to help them, like for example making the robot automatically traverse a path to a specified point. This kind of simple task can be called a behavior.

An improvement that can be made in these behavior controlled robots, is in the specification of which behavior should be selected in a certain situation and how the behavior is executed. This can be done by creating behavior-controlled robots, that can autonomously select the best behavior to activate on a certain moment, and using their sensors as input, can choose the right way to navigate.

Currently, not many behavior-controlled exploration algorithms exist. An exception is path finding on challenging terrain [9] . This research will result in a method to easily adjust and improve the behavior of any robot in any robot commanding program, especially focussing on UsarCommander, the program used by the UvA Rescue team[1].

There has however been research in Behavior Based Artificial Intelligence since 1986.

2 Behavior Based Artificial Intelligence

This was first researched by brooks [3], who laid the foundations of looking at intelligence in different layers. Brooks proposed that the following four elements were key requirements in a robot controlling system:

1. **Multiple goals:** A robot should be able to chase multiple goals at the same time, for example reaching a place in minimal time, while conserving power reserves. There should be an ability to prioritize goals, so that dangerous situations can be evaded while the main goals are still executed when the robot is able to. A simple example is being able to evade obstacles while reaching the place it wants to reach.

2. **Multiple sensors:** Most robots have more than one sensor, each having its own error measure. Some sensors have a bigger error in certain situations then others. For example while traversing inside a building, a robot should not be trusting its GPS sensor (Global Positioning Satelite), while being outside this would be a good option. A robot should be able to cope with these different errors, and use the right sensors at the right time with the right amount of trust.

3. **Robustness:** A robot's artificial intelligence should be robust. This means that when certain sensors fail, or unexpected deviations from it's normal environment occur (for example when a robot meant for inside-use comes outside a building, where there are less walls, but more small obstacles), the robot should still be able to act in a sensible way, in stead of just stop and stay still, or act randomly.

[1] Team description and more at: http://www.jointrescueforces.eu/wiki/tiki-index.php

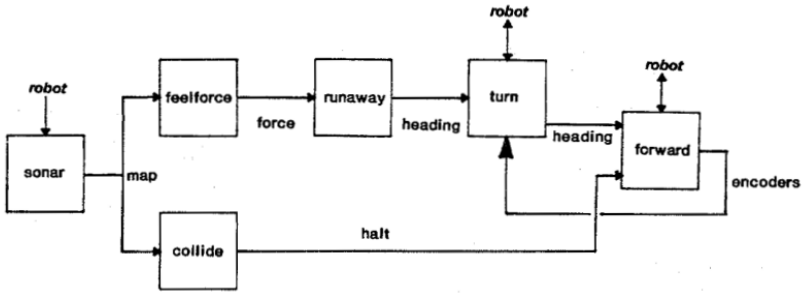


Figure 1: A level 0 control system, as proposed by Brooks

4. **Extensibility:** Brooks only speaks of being able to enlarge the processing power of the robot, when more sensors are loaded on the robot. I would like to add to this, that the intelligent system should have some kind of modularity in its software, making extending the system to work with a new kind of sensor, or even a totally different robot or environment (for example virtual vs the real world) easy, without having to rewrite big parts of code, or search through the program to find where a sensor should be added and where the activation of the sensor occurs, etc.

Brooks explains that typically, robot intelligences slice problems up in the following order: Sense, map sensor data in a world representation, plan, execute task and at last: control motors to do so. He then offers a new implementation of problem-decomposing, in the following order, and calls these 'Levels of competence'

0. "Avoid contact with objects (whether the objects move or are stationary)."
1. "Wander aimlessly around without hitting things."
2. "'Explore' the world by seeing places in the distance that look reachable and heading for them."
3. "Build a map of the environment and plan routes from one place to another."
4. "Notice changes in the 'static' environment."
5. "Reason about the world in terms of identifiable objects and perform tasks related to certain objects."
6. "Formulate and execute plans that involve changing the state of the world in some desirable way."
7. "Reason about the behavior of objects in the world and modify plans accordingly."

Each level of competence adds complexity to the entire system, thereby creating a layered implementation of behavior in an (in that time) untraditional way. Brooks proposes that each of these layers can be implemented in a finite state automaton, resulting in figure 1 as a representation for the zeroth level, and, by augmenting this with an FSA for level one and two, in figure 2.

In the level 0 representation, the robot will 'run away' when it is standing still and a moving object closes in. Alternatively, it will halt when a probable collision is detected. This is enough for simple obstacle avoidance.

This representation is augmented by inserting the avoidance and wander states above it parallel to the runaway state, in figure 2, This results in level 1 behavior: a robot capable of wandering around aimlessly, without hitting any objects. The direction outputted by the level 0 FSA is, when possible, overridden by the direction of the level 1 output.

As can be seen, this method a very large FSA, when we add the second level of control. This has the advantage of being capable of more complex behavior, in this case exploring an area, thus no more simply wandering around, but reaching places it has not yet explored. A disadvantage of this method, however, is that these big FSA's are quite complex to understand Adding more and more complexity to the system results in bigger and bigger images, resulting in more representation complexity and, in the worst case, in a system that only the creator can understand fully, but cannot anymore be represented in a clear way. Of course it needs to be considered that this system was created in 1986,

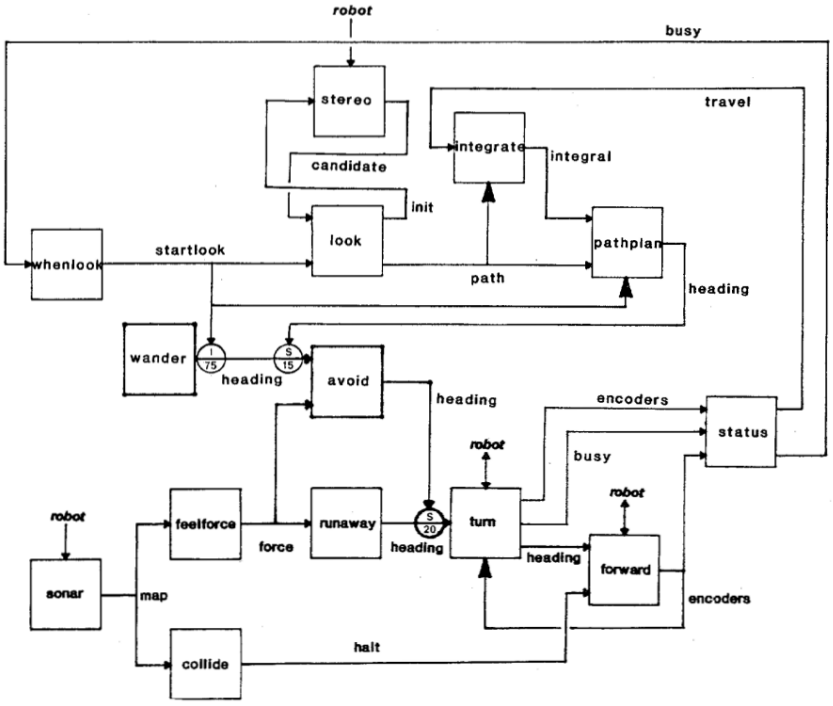


Figure 2: A level 2 control system, as proposed by Brooks

when computers were many times slower and capabilities were limited. Brooks managed to get the level 2 version working on a real robot in the time, by distributing the system over many cores.

This is the main advantage Brooks proposes, of this kind of implementation: The processes needed for the in- and output of the states, can be done with the least possible interaction between processes, making Brooks able to distribute the implementation over different processors and thereby able to run this, for that time, complex program.

Nowadays, this implementation is a bit outdated, mainly because of the complex structure of the representation. The behavior based approach, however, has been used in several solutions to controlling robots. These solutions will be discussed in the following section.

3 BBAI Implementations and Alternatives

This section will cover most of the BBAI implementations that can be chosen from when deciding to create an application that should be capable of specifying a Behavior Based artificial intelligence.

3.1 XABSL

One of the implementations of behavior based software is *XABSL* [7] [6]: a programming language, created to easily describe behaviors for autonomous agents based on hierarchical finite state machines. It is the software that has been used by the German robotic soccer team to specify their robots' behaviors. The team won in 2008, and the years after that.

The language is used to specify a finite state automaton hierarchy. This means that the user defines several finite state automata, which can activate each other. Each state makes decisions on certain variables, and as an output activate another state or another FSA. The hierarchies are built up from the following components:

A XABSL-specification is built up from the following components:

- **Agents:** A rooted acyclic graph, containing all the behaviors for one agent. Several of these agents can be created, all having their own graph and thus their own behavior.

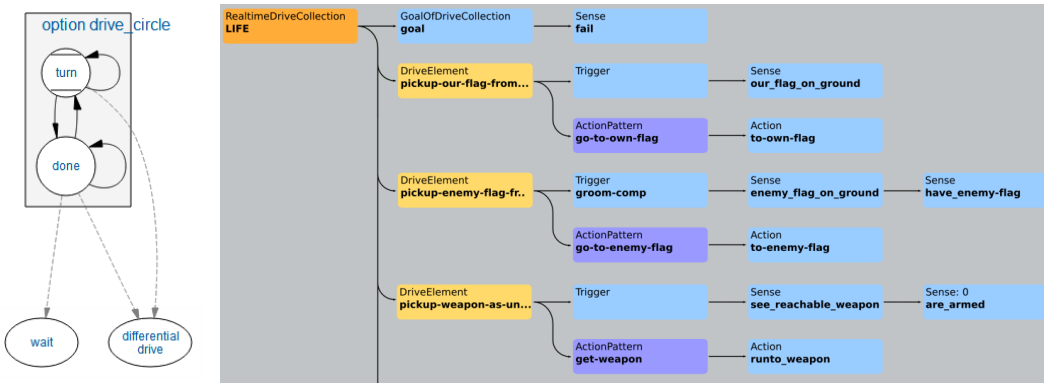


Figure 3: Left: An example of a figure generated by the XABSL compiler, from XABSL code. Right: a POSH hierarchy

- **Options:** Complex agent behavior. Each option is on itself a finite state machine, containing several states. When creating an agent, the start option can be specified, which makes the user able to create different agents from one Option hierarchy. Options can also have parameters, enabling an option to have different outputs for different agents. Figure 3 contains an example of a simple option, that makes the agent turn 360 degrees and then stop and wait for a certain amount of time.

- **States:** Options are bounded to each other by states, each state has a decision tree, and an action. The decision tree decides whether to stay at this state, or to go to another state. These decisions are based on variables that can either be internal, or inputs.

  When a decision tree decides to stay at its current state, an action is performed. Actions can be activation of a basic behavior, or of another option. Several actions per state are permitted.

- **Basic behavior:** At every leaf of an option (so, every state with no other states to reference to) a basic behavior is activated. This is a small piece of native code (C++ or Java), that influences the actions of the agent in its world.

This is an improvement over Brooks' BBAI, because the FSA's are now no longer directly connected to each other through state connections, but are connected via the actions of certain states, in that way improving the comprehensibility of the representation, thereby also improving the modularity of the system, because modules can be better recognized and then expanded.

By using basic behaviors, that can be written in C++ or Java, XABSL also enables distribution of the system: Each basic behavior can run its own module. This way basic behavior can be a module that simply makes a robot move, but also a module that finds a ball in a soccer field, using libraries like OpenCV. This makes an XABSL application capable of the same things as any native C++ or Java application, which is almost everything one or more computers can do.

Section 4 will explain more clearly the advantages of XABSL, and the possibilities of agents, options, states and basic behaviors.

3.2 POSH

POSH [2] is a very similar alternative implementation of a Behavior Specification Language. Posh is defined as a *Behavior Oriented Design*, which is a combination of *Object Oriented Design* (OOD, used by object oriented programming languages like Java and C++) and *Behavior Based Artificial Intelligence* (BBAI).

From OOD the language takes the object hierarchy that it is known for. In object oriented languages a person is capable of creating an object based on another object. These can be *Abstract classes*, or *Interface classes*. When using an abstract class to define an object, this means the class can be extended by another object: The new object automatically has all the properties its *Base class*

(the original, abstract, class) has, but can override some of them, or add new ones. An interface class can define what its subclass should have, for example when an interface class specifies a method that searches for a doorway, using laser sensors, its subclass should implement this method. The interface class itself does not have any actual implementations. BOD objects are literally built in an object oriented language, thereby having all its advantages

The BBAI-part of it is the decomposition of intelligence as subtasks called *acts*. Examples of acts are knowing your position and planning a route. There is no implementation of prioritizing certain acts above others, other than that they come earlier in the POSH diagram.

Behaviors in BOD are thus specified as a *Behavior object*, written in an Object Oriented language. They are split up in actuators and senses. The actuators are used to act on the world, for example move in a certain direction, or pick something up, whereas the senses are used to inform the planner the current context. Context can be any piece of information about the world, or the agents internal state. The whereabouts of an object, or the data from a laser scanner can be context, but the agents current battery level is also context.. All the specified behaviors together form the *Behavior Library*, which can be used by the action planning system to select the right behavior on the right time.

Furthermore POSH uses three aggregates: simple sequences, competences and drive collections.

1. **Simple sequences:** The sequence is simply a sequence in which order a diagram should be traversed

2. **Competences:** A competence is a prioritised set of condition-action pairs. These condition-action pairs are based on the current context (described above). Because of the hierarchical structure of the system, only small pieces of context have to be processed at a time. When a certain part of competence is reached, competences have been passed higher up in the hierarchy, meaning that this information needs no more checking. In this part there is assumed that in the time it took to traverse the tree, the world has not changed significantly.

3. **Drive Collections:** The drive collection is a special competence, that is executed before each program cycle. The collection contains all vital condition-action pairs to be able to survive. For example when an enemy is close (given that the environment has enemies that can seriously harm the agent), the agent should hide from the enemy, or take other actions not to get harmed. The drive collection can also contain routines that have to be executed every once in a while, like checking the environment for safety

This is actually quite similar to XABSL, because selects actions based on decisions based on its findings. The actions are always executed by an external program. There are some important differences though:

- POSH is designed to be used by non-programmers. This means the interface is easy, colorful and simple, whereas XABSL prioritizes complex capabilities, ignoring the fact that non-programmers then couldn't use it. This improves the adaptability of XABSL far above the capabilities of POSH, resulting in ability to create more complex behaviors.

- Where XABSL has a close coupling with the perception stream of the robot, POSH has no variable management, enabling the system to be a lot easier to use, but also maximizing the complexity of the specified behaviors to a lower maximum than XABSL offers.

3.3 Petri Net Plans

Another possible method to represent robot behavior as plans, is by using *Petri Net Plans* (PNP), a language based on *Petri Nets* (Also known as Place/Transition Nets or P/T Nets).

A Petri net is a mathematical modelling language, making it an alternative to the use of Finite State Automata. The formal definition of a *net*, as given in [4], is the following:

"A *net* N is a triple (S, T, F), where S and T are two disjoint, finite sets, and F is a relation on $S \cup T$ such that $F \cap (S \times S) = F \cap (T \times T) = \emptyset$."

In this definition, the elements of S and T are respectively called *places* and *transitions*. The elements of F are called *arcs*. A transition t can be made, when a marking M marks all of its input places. If t is enabled at M, it will occur, and thus lead to the successor of marking M, which we call M'. This is denoted by $M \xrightarrow{t} M'$.

A *Petri Net* is a pair (N, M_0), with N representing a net, and M_0 representing the *initial* marking of N, which means that M_0 is the first marking of N. A sequence of transitions that enables the graph to go from marking M_0 to M_n, is called a *finite occurrence sequence*. Infinite occurrence sequences are possible too, when there is no clear end marking, but transitions keep on happening. If a state is reached where transitions can no longer occur, this is not called an infinite occurrence sequence.

The main advantage of Petri Nets, is that due to their definition, they are very analyzable. Three attributes exist, on which analysis can be done more easily than most comparable mathematical models.

1. **Reachability:** For most graphs it is hard to exactly calculate if every node in the graph can possibly be reached. For petri nets, it is possible to calculate this automatically, and with great certainty. An algorithm for this calculation is given in [8].

2. **Liveness:** Degrees of liveness can be assigned to Petri Nets. Loosely speaking, a petri net is live when every transition can always occur again. More precisely, the following *levels* of liveness are possible[2]:

   - $dead$: It can never fire, i.e. it is not in any firing sequence in $L(N, M_0)$
   - $L_1 - live$: (potentially fireable) if and only if it may fire, i.e. it is in some firing sequences in $L(N, M_0)$.
   - $L_2 - live$: if and only if it can fire arbitrarily often, i.e. if for every positive integer k, it occurs at least k times in some firing sequence in $L(N, M_0)$.
   - $L_3 - live$: if and only if it can fire infinitely often, i.e. if for every positive integer k, it occurs at least k times in V, for some prefix-closed set of firing sequences $V \subseteq L(N, M_0)$
   - $L_4 - live$ (live) if and only if it may always fire, i.e. it is L_1-live in every reachable marking in $R(N, M_0)$.

3. **Boundedness:** A petri net is *bounded* if its set of reachable markings is finite. For petri nets, it has been shown that this is also decidable, because an unbounded Petri net, is easily characterized in the following matter:

   An unbounded petri net is characterized by a reachable marking M, and a sequence of transitions σ, so that $M \xrightarrow{t} M + L$, where L is some non-zero marking and the sum of these markings is defined place-wise. The sequence σ can be called a *token generator* when it, starting at M leads to $M + L$. This generator makes the petri net unbounded.

For this section, it is extra important to know that Petri Nets have the ability of forking and joining their tokens. This means that from one state, two tokens can be output, going to two different states. On the other hand two tokens from different states can be asked as input for one state, making the state wait for output from both input transitions, before continuing by outputting one token of itself, in the next marking.

Petri net plans [11], is a language using these petri nets, to create special *plans*. Because of the petri nets fork and join abilities, Petri Net Plans is explicitly suitable for controlling several robots. The clear advantage of this approach, is that there is only one petri net plan needed, to control all the robots in the field, whereas most other methods have one plan per agent.

An example of a petri net plan can be seen in figure 4. It can be seen that the h_sync method is used to halt the robots until both have synced, and only then continue to the next state.

Other advantages of Petri Net Plans are closely related to the advantages of Petri Nets above Finite State Automata. A consideration has be done though: It is harder to specify a completely sufficient petri net plan, than to specify several simple Finite State Automata.

[2] From wikipedia

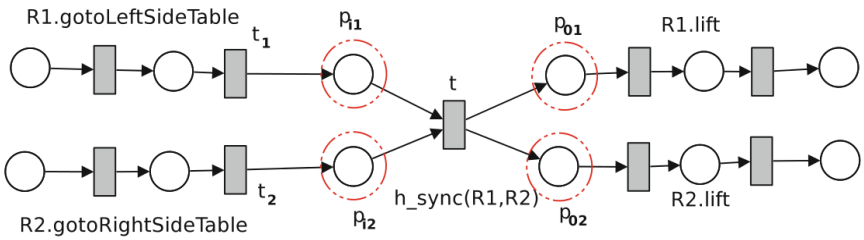


Figure 4: A petri net describing the actions of two robots, for a simple task execution. The robots use h_sync to synchronize. Due to the join capabilities of a petri net, none of the robots will continue to state p_{0n} before the sync is done.

```
1  act patrol2{int a}
2      start:
3          while(a != 0)
4          {
5              a = a-1;
6              Turnto(180);
7              Move(1000);
8              Turnto(0);
9              Move(1000);
10         }
11         succeed;
12     onInterrupt:
13         waitfor(sfDonePosition());
14         suspend;
15     onResume:
16         a = a+1;
17         goto start;
```

Figure 5: A simple specification of a patrolling behavior in COLBERT

3.4 COLBERT

COLBERT is a programming language using the Saphira Architecture. The abstract of [5] states that: "The design criteria for Colbert are:"

1. "To have a simple language with standard iterative, sequential and conditional constructs."
2. "To have a clear and understandable semantics based on FSAs."
3. "To have a debugging environment in which the user can check the state of the system and redefine Colbert activities."
4. "To have an small, fast, and portable executive."

The main advantage of COLBERT is that it is small and fast. And as we all know: quick reflexes are important in a dangerous world, so having small and fast behavior based AI has it advantages. COLBERT is based on a subset of ANSI C and it is even possible to compile it to native C code, to make the program even easier to run.

Figure 5 shows a small sample of specification in the COLBERT language.

3.5 Other alternatives

These are the main alternatives. There are others, like using constraint logic, with aid of If-then-else statements directly in the main program. One can also use machine learning techniques, for example *Reincorcement Learning* methods are used by *Sarsa* and *Q-learning*. This is, however a complete

different way of looking at the problem, with its own set of results, which are not comparable to the ones reachable with human-specified behaviors, because the current learning techniques are not yet sufficient.

4 Language of choice

In this research, of the above specified languages, XABSL has been chosen. There are a few advantages of XABSL above the others, but also a few disadvantages. The following part will discuss why XABSL is, in this case, a better choice than de aforementioned alternatives. Then, this section will explain more about how XABSL hierarchies are specified, and how they are combined with other programs, traditionally.

4.1 Advantages of Using XABSL

The first thing that needs to be considered when choosing a behavior specification type and language, is what characteristics are the most important. Here is a list of things that the ideal behavior specification language for multiple robot control should have.

- The points mentioned by Brooks:
  - *Multiple goals*
  - *Multiple sensors*
  - *Robustness*
  - *Extensibility*
- **Modularity:** It should be possible to build up a specification in small pieces, thereby keeping track of what is specified where, and not getting entangled in pages of code, of which no body remembers the function
- **Documentation:** Most programs, especially the ones used in the RoboCup, are not only used by one person. For a team to be able to work on a project together, it is good to easily understand the work somebody else has done. This is achieved by keeping documentation in your code, but it is achieved even more by documentations on websites or in other documents, that can be consulted without having to actually dig into the code.

All of the mentioned possibilities enable having **multiple goals**. POSH has the possibility of adding the set of extra important goals, which the others do not, but POSH has too many disadvantages in relation to XABSL to make this point count.

Of all the possibilities discussed, XABSL has the clearest solution for dealing with **Multiple sensors**. XABSL has input variables, which, together, are a world representation that is constantly updated by the Engine, that runs the decision-making program. Petri Net Plans has of course the added possibility of keeping track of several robots at the same time, in the same graph. This could be seen as an advantage above XABSL, but since XABSL offers us the possibility of adding extra variables from the running program, the relevant information about other robots could also be added to the reasoning engine via that method.

Robustness is the ability of functioning in another setting than usual. A lot of the robustness of a robots behavior is not in the language in which it is specified, but mainly in the specification itself. A way to force programs to be at least a bit robust, is by error-checking the specified behaviors before running the program. This can be done very easily in PNP, because Petri Nets have so many algorithms for complete checking of reachability, liveness and boundedness. XABSL however also offers a compiler that compiles your code and finds the necessary problems. A complete survey of if every node in the graph is reachable is not done, but to be able to do that, more knowledge of the input variables should be present.

Extensibility was a bigger problem in 1986 then now. Computer systems were slower, forcing programmers to create the fastest possible programs, when creating something heavy, like a behavior

```
1 include "my-basic-behaviors.xabsl";
2 include "my-symbols.xabsl";
3 include "Options/drive_circle.xabsl";
4
5 /** Start an agent with a name and a specified option. */
6 agent robot1("Robot-1", drive_circle);
```

Figure 6: A very basic agent file, which includes only the explicitly necessary

control system. Brooks' system had problems with their *level 2* specification, whereas nowadays the cheaper computers could run that. Inherently, almost all possible options of creating a behavior specification would allow for a system to be expanded with a new sensor, or a new kind of processor. In XABSL, adding a new sensor, would simply mean adding a new option, that accounts for dealing with that sensor. Another possibility would be to process the sensor to a world representation, in whatever program is used in combination with XABSL.

XABSL, together with POSH, are the most **modular** languages of the ones proposed. This is because of the modular way the behavior specifications are built up. My preference goes to XABSL in this case, because its finite state automata enable a bit more complex implementations than the acts used in POSH hierarchies.

In contrast to the other possibilities, the XABSL compiler offers a method of directly translating the code to svg images, and the comments to html context, and automatically combining those to a full web page, with the entire documentation. Figure 3 shows an example of an automatically generated image. These images, together with the extracted comments, make for a complete documentation, which provides overview to any one who's interested, in less time and effort than when some body would have to look through the code.

4.2 Disadvantages of Using XABSL

Especially PNP offers some great possibilities that XABSL does not. Using Petri nets in stead of the FSA-hierarchy XABSL provides, has the following advantages:

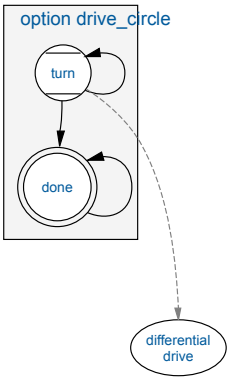
- Mathematical proof of reachability: It can be useful to be able to show that every node in a behavior specification can certainly be reached. Unreachable nodes mean bad specification, which mostly comes from, or results in bugs, which in their turn results in a program that could have worked better. Some of these differences, however, are resolved in [1]
- Multiple robots in one plan: PNP offers a possibility of using one plan with multiple robots, at the same time. Of course, when using XABSL two robots can use the same option tree, but when using PNP, the robots actually share the Petri Net Plan, enabling the system to work more efficiently. A part of this problem is solved by creating a shared world-representation in the program that is used in coherence with XABSL, but some functions, like for example the h_sync in figure 4, are difficult, if not impossible, to recreate using XABSL.

4.3 Creating an XABSL-specification

When using XABSL in coherence with any application, the *XABSL engine* is used to control the behavior. This engine uses a user specified XABSL graph and the internal variables of the program it is running in in coherence, to make decisions about what *basic behavior*, or *option* to run next. More about the engine can be read in section 4.4

This engine runs on *intermediate code* that is automatically generated from several XABSL files, by the XABSL compiler. This intermediate code is a file containing all the information from the different files, which is read by the engine. The following files are always used by the compiler:

- **Agents:** One of the files, specifies all the agents. This is the root file of the XABSL hierarchy. In this file, all the other files are included (similar to the include in the C programming language).



(a) output option of the code

```
1  /** Drives a circle */
2  option drive_circle{
3      initial state turn{
4          decision{
5              if(amount_turned < 360){
6                  stay;
7              }
8              else{
9                  goto done;
10             }
11         }
12         action{
13             differential_drive(speed=0,
14                 turning_speed=-0.3);
15         }
16     }
17     target state done{
18         action {}
19     }
20 }
```

(b) code in `drive_circle.xabsl`

Figure 7: A simple option (FSA), capable of making a robot turn a circle. In this option `amount_turned` is an input variable and `differential_drive` is a basic behavior.

When that is done, agents can be specified by typing `agent robotName(robotIdentifier, startOption)`. An example agent file is given in figure 6

- **my-basic-behaviors** and **my-symbols:** As you can see, the agent file includes not only one option, but also two files with basic behaviors and symbols. The first of these two, contains the names of the basic behaviors. These basic behaviors need to be specified in C++ or Java, depending on which of the engines is used. The file simply names them, for the compiler to be sure that these behaviors exist, and are called in the right manner.

  The symbols file defines all the variables that are used by XABSL. Three kind of variables exist:

  1. *Input variables:* variables that are put into the engine, from the external program. For example, the x and y position of the robot.
  2. *Internal variables:* these are the variables that XABSL keeps track of itself. An internal variable does not have to be implemented by the engine running program, but can be completely modified by the engine itself. Internal variables can also be constants, to enable very simple modification of constants like, for example, driving speed, without having to recompile the main program.
  3. *Output variables:* If XABSL should be able to change a variable in the main program, this is an output variable.

  These three together, enable complete control of XABSL in the main application, and the other way around. The variables can be monitored at all time, by simply asking the engine for the current value.

- **Option definition files:** Each option defines a finite state machine in its own file. These options are conveniently placed in a separate folder, and included by the agents file. As can be seen,

the robot in figure 6 has option drive_circle as argument. The definition of this simple option can be seen in figure 7. An option always contains one or several states, one of which is the *initial state*. This concept is known in Finite State Automata as the start of the automaton. Optionally, a *target state* can be provided, which means that the option stops there.

As can be read on the XABSL website[3] and in [7], every state has the following layout:

```
<state> ::=
  [initial] [target] state <name> {
    decision
    {
      [ else ] <decision tree>
    }
    action
    {
      { <action definition> }
    }
  }
```

The decision tree makes decisions based on symbols specified in my-symbols.xabsl. All common boolean operators are supported, and because the trees are built up of merely if and else statements, the tree is very simple to create and understand.

An action definition can contain several options or basic behaviors to be called. These will be called in the order in which they are specified, so not all at once.

An automatically generated documentation web page of an option always shows the code of the decision tree (including comments), a graph of the finite state automaton, graphs of every state and links to the options or behaviors in the action definition.

4.4 Traditionally Combining XABSL With Any Existing Program

Before the existence of this research, XABSL could only be combined with other C++ programs, by using the C++ XABSL engine, or later with Java programs, by using the Java engine. Of course the third possibility of parsing the intermediate code with an own implementation is present, but that would require actually rebuilding the XABSL engine in the programming language of your choice, which, to do it without faults, would require a great understanding of XABSL, and a **lot** of time. This section will cover how the engine is used, and what it does. The focus will lay on the Java XABSL engine, since that is the one used in the rest of the research.

Each robot will get its own reasoning engine. The engine is simply created by creating a Java Engine object, which only takes an output stream (for debug messages) and a time function as an input.

Next, all the symbols and basic behaviors present in the `my-symbols` and `my-basic-behaviors` files have to be registered to the engine. This means that for every input variable, a variable of the Java program running the engine needs to registered, together with the name it has in XABSL, and the getter function you have created for the variable in Java. This enables the engine to access the variable whenever it needs to.

Behaviors are registered almost the same way, but it uses a Java behavior class, which extends the basic behavior class provided by the XABSL framework. This enables the engine to run the `execute()` method of that class, which can be any type of native Java code. The behavior class of a XABSL using program can do anything another Java class can, thus not limiting the possibilities of the behavior.

Once everything is registered, the intermediate code can be read into the XABSL engine. This results in the engine linking everything together, after which the engine can be used to select the right behavior on the right time. This is done by calling the engine's `execute` method in a loop. This enables users to choose when the engine should be executed, making it possible to wait until every sensor has been updated, and then execute again.

[3] www.xabsl.de

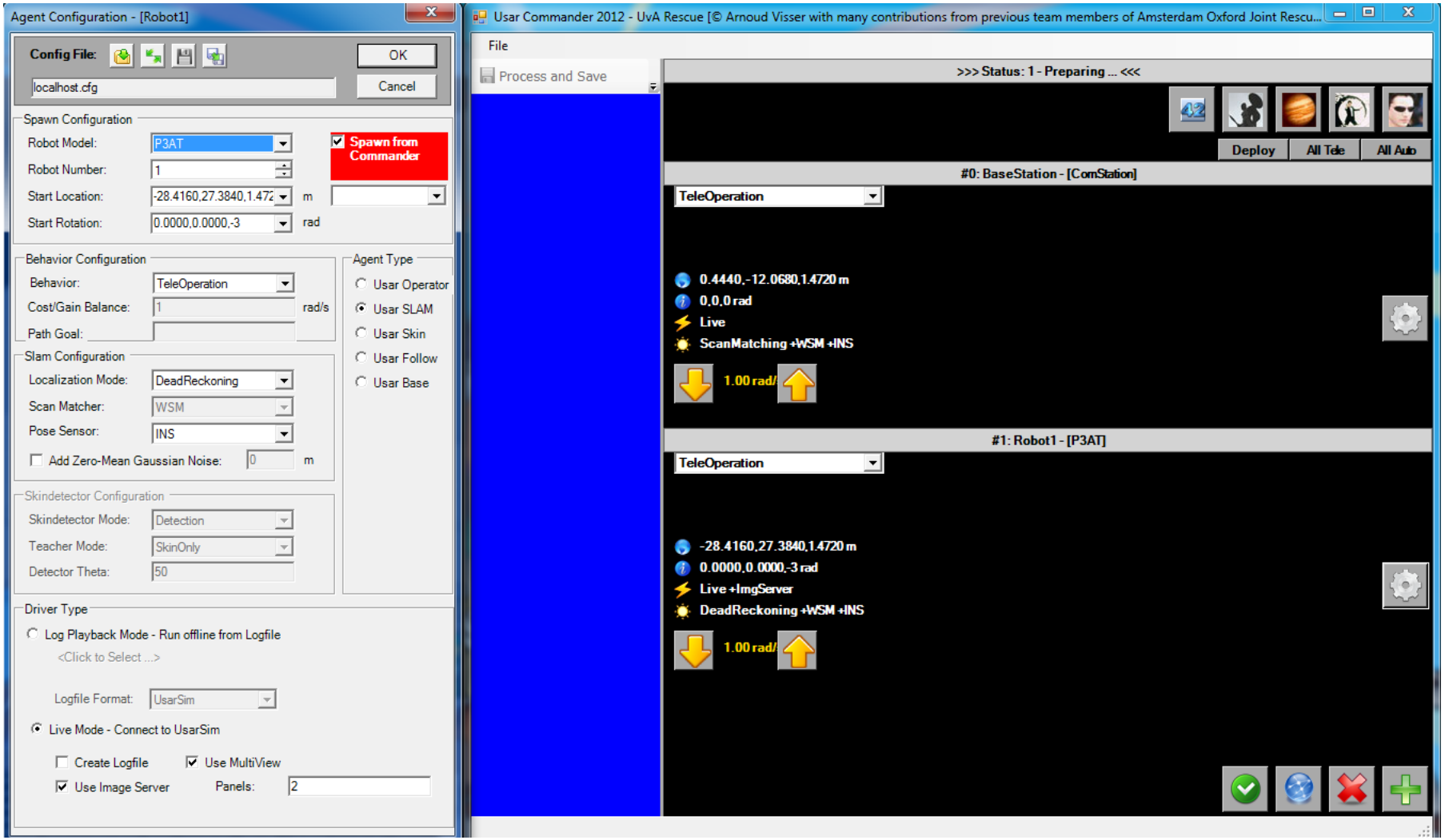


Figure 8: The start screen of UsarCommander, alongside the screen where several options can be selected for a robot, including SLAM method and

5 RoboCup Rescue

Before continuing with the research done, one last component has to be introduced. The Rescue program used to test the framework on. This program is called UsarCommander. The program is written in Visual basic and provides many tools to get information from a session in USARSim.

5.1 Description

UsarCommander[4] was originally developed by Bayu Slamet, and extended by Arnoud Visser and many others. The program takes care of connecting to USARSim (the simulator used in the RoboCup) and makes the user able to easily get sensor data from the simulated robots. It is also possible to control the robots with several types of behavior, like corridor-following, obstacle-avoidance, or tele-operation, the last of which enables the operator to manually control the robots, using an interactive human interface.

Over time, the system has been expanded with many subprojects, for example one implementing a SLAM (Simultaneous Localisation And Mapping) algorithm, to make an accurate map from the sensor data of several robots [10]. All the information used and produced by these subprojects can be accessed by other subprojects, resulting in an ideal environment for creating new robot-controlling applications. An example of the UsarCommander interface and its possibilities, is shown in figure 8.

5.2 Alternative possibilities

Since the university of Amsterdam is not the only university that enrolls in the Rescue simulation league of the RoboCup, alternatives to UsarCommander are available. It is however not a trend over all of the world to create open source programs for the RoboCup, which limits the list of possibilities.

One alternative that is widely known is Iridium. A program created to provide easy access to USARSim for programmers with little experience. This is a nice framework to test your programs on, but it lacks the possibilities and features that UsarCommander offers.

6 Approach

There is one problem left to solve: UsarCommander is written in Visual Basic, whereas XABSL is written in C++ or Java. The solution is socket programming.

Sockets programming is possible in almost every common programming language, and simple means 'writing strings to, and reading strings from a network socket'. It enables computers to write and read to and from not only their own network sockets, but any one that has been opened to *listen*.

This research offers a framework, written in Java, that enables using XABSL on any program that has been written in a language supporting Socket programming.

6.1 Information flow

This section will describe what information is contained in which part of the program, and how information is shared between both of them. A representation of the information flow of the program can be seen in figure 9. In this figure, an S represents a socket. The commanding program is, in this case, UsarCommander, but can be any program. JXI stands for *JavaXabslImplementation*, the name of the project this research is about.

6.1.1 Messages

The socket connection enable sending messages from one module to the other. These messages are always strings, so an encoding of other values has to be made when sending, for example, arrays.

Several methods exist for automatically encoding the messages that are sent over sockets. The best, widely used example is JavaScript Object Notation (also known as JSON)[5]. It offers very easy encoding of Java objects, when using a Java-like language (Java or JavaScript). Several libraries exist

[4] Available at http://www.jointrescueforces.eu/wiki/tiki-index.php
[5] Exact description and implementations can be found at http://www.json.org/

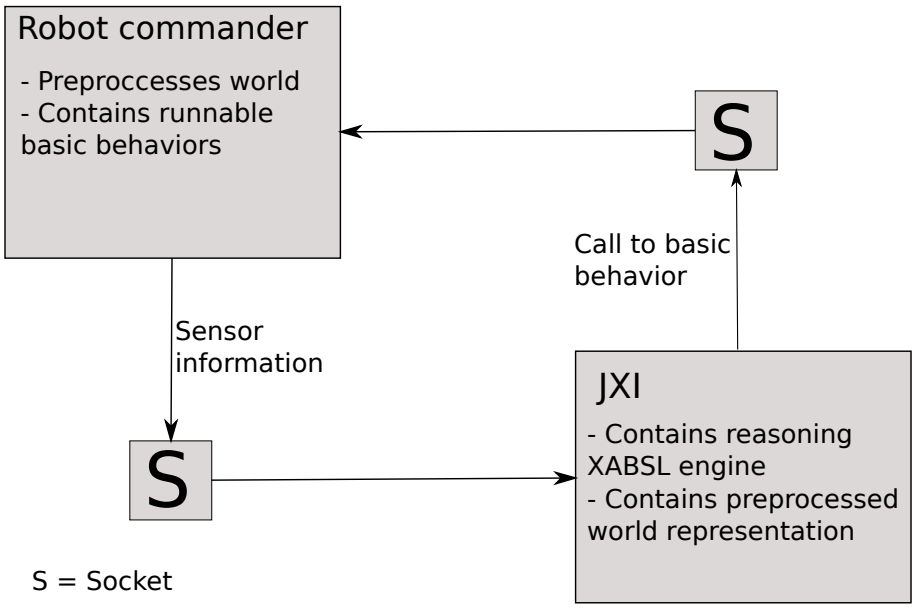


Figure 9: The information flow from any robot running program to JavaXabslImplementation (JXI). A world representation is made in JXI, enabling the commander program, as well as JXI to preprocess sensor data to variables in the world representation. For example `ammount_turned` can be created in JXI, when executing the option `drive_circle`, from the information then ins sensor sends. S represents a socket connection.

to use JSON with other languages, but some of them lack the simplicity JSON offers to others and many of the JSON libraries are created by third parties, which makes them less trustworthy.

For example when using JSON in Visual Basic, a solution is offered by MSDN[6], but it comes down to the programmer having to do most of the formatting himself. For this particular reason, this research uses its own, simpler notation for messages.

Every message is built up in the following structure:

```
<target>:<parameter1>,<parameter2>,...,<parameterN>
```

When messages are sent from the robot commander unit, the messages mostly consist of sensor data. For example, a laser range scanner sensor would send the following data:

```
LASER:1.14,2.55,3.54,5.54,2.99,1.44
```

This can then easily be parsed in JXI, by splitting the entire string on the colon. The first element of the array that comes out, is the name of the sensor, and the second element contains the sensor data, in a string that can be split on the commas. Splitting is done by invoking the Java method `split()`, which is a standard String operation.

When the XABSL engine invokes a Basic Behavior, JXI will send a message containing the behavior and its arguments, in the same fashion. For example:

```
DIFFERENTIALDRIVE:5,1
```

will invoke the differential drive method in the robot commander. Currently this is the only method that has to be implemented in the commander program. The first argument resembles the driving speed, the second argument resembles the turning speed. A positive turning speed makes the robot turn left, and a negative value makes it turn right. Together, these values can invoke every movement

[6]The solution is described here: http://msdn.microsoft.com/en-us/library/bb299886.aspx

a simple robot can make. Of course, more behaviors can be added for flying robots, or robots with more moving parts.

6.2 World Class

The JXI package contains a world object. This object contains all the input variables XABSL requires. In the future, this object can be shared with other robots, enabling them to know each others location.

The world class also provides a simple interface for users of the system to add their own, new, variables to the JXI. More information on this class and adding variables, can be found in the code documentation.

6.3 Other Added Simplicity

The connection is not the only thing JXI adds to the standard XABSL engine. The engine tends to require some objects that are always the same, but not included in the standard package. An example of these objects, is that before being able to register a decimal parameter (symbol) to the engine, the interface of `DecimalParameter` should be implemented. This only has to be done once, to have a runnable class in Java, but for some reason is not delivered with the XABSL engine. Of course JXI does deliver a `MyDecimalParameter` class that does this.

Also an abstract class of a basic behavior (`StandardBehavior`) is shipped with the JXI package. This is an extension of the `BasicBehavior` class in the XABSL engine, that also handles the socket connection, making every basic behavior capable of sending messages to the robot commander unit.

7 Results

This section contains the results of the experiments that have been done with the system.

7.1 UsarCommander

A few simple adjustments had to be done to be able to run JXI alongside with UsarCommander. This mainly involved receiving and sending messages over a socket connection, and creating a new behavior in the choice menu, to be able to start a connection to XABSL when starting an USARSim session.

When these adjustments were done, both programs were capable of connecting to each other, and the first XABSL hierarchy could run.

7.1.1 Driving a circle

The first XABSL hierarchy used, was one capable of driving a circle. The code and option graph can be seen in figure 7, which was explained in section 4.3. The agent directly started in this option, so no other files were needed.

This resulted in the robot turning slightly more than 360 degrees, which is Okay, because the XABSL engine waits until it's over 360 degrees, and then sends back a message for the robot to stop spinning.

Of course, driving a circle is not the only possible behavior we would want our robot to do. In fact, it is no desired behavior at all, we could make it drive circles all day, but it would never reach any victims that way.

7.1.2 Walking a corridor

That's why the next step is to make the robot walk a corridor. The algorithm for walking a corridor was already placed inside the code of UsarCommander, so would not have to be reinvented.

The original walk-corridor relied on laser range scanner data. To minimize data traffic between the two units, the laser range scanner data had to be pre processed. This is done by taking the minimum value of 7 regions of the scanner: from the 'western' end of the robot to the 'eastern' end. The resulting option and code can be seen in figures 10 and 11. The code only shows one of the

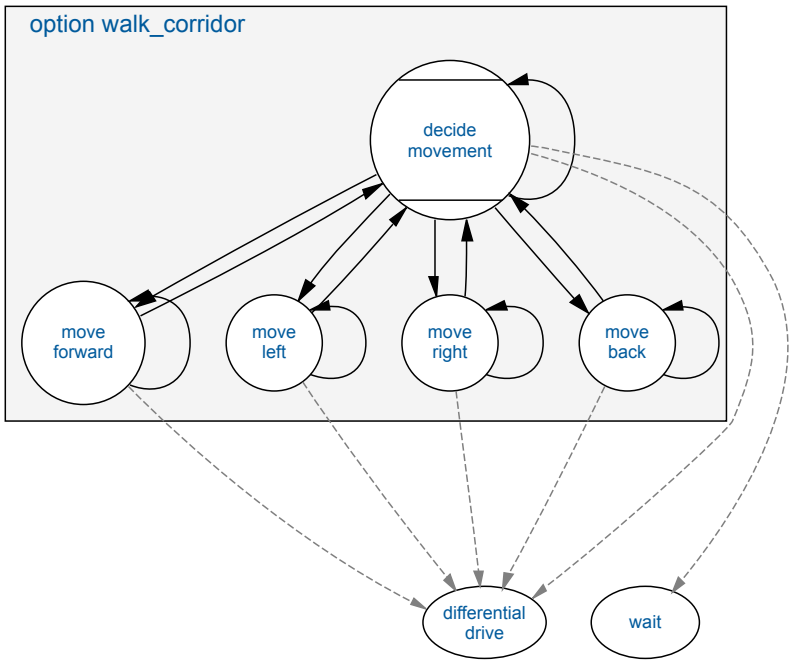


Figure 10: The graph for autonomously traversing through a corridor. The principle is simple: the robot always turns to the place where the longest laser ray comes from.

movement states, since the principle is the same for all of them. Creating the XABSL code for this corridor walk algorithm took very little time.

The corridor traversing had a bit of problems. The XABSL engine reacted good on everything, just as expected, but the UsarCommander was not reacting fast enough to send all the sensor data in time. This meant that after running the program for a minute, the sensor data of 30 seconds in was only just coming in! Since the (distributed) system was running on 2 different systems (UsarCommander and UDK on one, JXI on the other), the problem could not be in JXI, because it handled every piece of information as soon as it came in. The problem thus had to be in UsarCommander, but couldn't be solved in time anymore.

8 Conclusion

In its current state, JXI is a proof of concept. The framework works, as far as it has been used. There is not yet any information of how the framework reacts in case of enormous agent specifications, or with a big amount of variables. Still, everything that has been tested worked great on the JXI-side, and XABSL has proven itself as capable of running a soccer league with 6 robots.

The JXI package offers great possibilities for every one who want to add Behavior Based Artificial Intelligence to his project. I would recommend the usage of JXI above the ordinary Java Xabsl engine, due to its poor documentation. The engine does not provide any documents about how to run it, other than their unit test, whereas JXI has a code documentation online, which explains how it works and how it is supposed to be ran.

Furthermore having a distributed system is often a good option. Especially when more robots are supported, one command center should be consulted on what to do next. This command center could very well be using JXI to choose the right behavior.

As an added note, a socket connected program is easily integrated in any kind of system. The only thing needed is a socket connection to JXI, something that combines the right variables to a simple message and something that reads the incoming messages, and runs the corresponding basic behavior.

```
/** Uses the data from the laser sensor to walk through a
 * corridor as good as possible, without bumping into the walls */
option walk_corridor{
  initial state decide_movement{
    decision{
      if(laser_max < maximum_laser_value){
        goto move_back;
      }
      else{
        if(laser_max == laser_min_n){
          goto move_forward;
        }
        else{
          if(laser_max == laser_min_nne || laser_max ==laser_min_ne
                        || laser_max == laser_min_ene)
          {
            goto move_right;
          }
          else{
            if(laser_max == laser_min_nnw || laser_max ==laser_min_nw
                          || laser_max == laser_min_wnw)
            {
              goto move_left;
            }
            else{
              stay;
            }
          }
        }
      }
    }
    action{
      differential_drive(speed=0, turning_speed=0);
      wait(time=2);
    }
  }

  state move_left{
    decision{
      /** go on till threat is over */
      if(laser_max == laser_min_nne || laser_max ==laser_min_ne
                || laser_max == laser_min_ene || laser_max == laser_min_n){
        goto decide_movement;
      }
      else{
        stay;
      }
    }
    action{
      differential_drive(speed=forward_speed, turning_speed=turning_speed);
    }
  }
}
```

Figure 11: The XABSL code for autonomously traversing through a corridor. Only one of the movement states is shown, since the principle is simple.

9 Discussion and future work

Of course, before combining this project to every system thinkable, some improvements can be made to the JavaXabslImplementation. This section will sum up the main inconveniences and the work that should be done to solve them.

- **Multiple robots:** Currently the program only supports using one XABSL engine with one robot and one connection to a robot commander system. To enable using several connections, the world class should be edited to support several robot information variables. Most of this editing involves creating arrays in stead of single variables, which is not hard to bring into practice, but for which more time was needed.
- **Messages:** Right now, the messages are plain and simple, but limited to simple data types. JSON offers an improvement over this limited complexity, but adds poor support for some languages. When complex variables are being used, including JSON should be considered. For now, the simplicity of the used messages outweighs the complexity of JSON.
- **Testing:** The JXI framework has not yet been fully tested. The fact that UsarCommander seemed to be too slow in sending sensor information, made the testing state a hard one. When the bug in UsarCommander is fixed, more testing can be done. An alternative is to test the framework on other systems, like Iridium or a new framework.

References

[1] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 175–188. ACM, 1998.

[2] C. Brom, J. Gemrot, M. Bida, O. Burkert, S.J. Partington, and J.J. Bryson. Posh tools for game agent development by students and non-programmers. In *The Nineth International Computer Games Conference: AI, Mobile, Educational and Serious Games*, pages 126–133, 2006.

[3] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, 1986.

[4] J. Esparza and M. Nielsen. Decidability issues for petri nets. *Petri nets newsletter*, 94:5–23, 1994.

[5] K. Konolige. Colbert: A language for reactive control in sapphira. In *KI-97: Advances in Artificial Intelligence*, pages 31–52. Springer, 1997.

[6] M. Lötzsch, J. Bach, H.D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language xabsl. *RoboCup 2003: Robot Soccer World Cup VII*, pages 114–124, 2004.

[7] M. Lötzsch, M. Lötzsch, J. Bach, H.D. Burkhard, M. Jüngel, M. Lötzsch, M. Risler, M. Jüngel, M. Risler, O. von Stryk, et al. *XABSL-a behavior engineering system for autonomous agents*. PhD thesis, Diploma thesis. Humboldt-Universität zu Berlin, 2004. Available online: http://www. martin-loetzsch. de/papers/diploma-thesis. pdf, 2004.

[8] E.W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246. ACM, 1981.

[9] H. Seraji and A. Howard. Behavior-based robot navigation on challenging terrain: A fuzzy logic approach. *Robotics and Automation, IEEE Transactions on*, 18(3):308–321, 2002.

[10] B. Slamet and M. Pfingsthorn. Manifoldslam: a multi-agent simultaneous localization and mapping system for the robocup rescue virtual robots competition. *Master thesis in Artificial Intelligence at the Universiteit van Amsterdam*, 11, 2006.

[11] V.A. Ziparo, L. Iocchi, P.U. Lima, D. Nardi, and PF Palamara. Petri net plans. *Autonomous Agents and Multi-Agent Systems*, 23(3):344–383, 2011.