A DSL for Behavior Trees in Lingua Franca

Akash Ahmad

Bachelor's Thesis
March 2023

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
M. Sc. Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Behavior Trees are hierarchical models used to describe behavior of *agents*, which are autonomous computer systems performing tasks depending on the environment [WJ95]. Their primary use is for industrial robots or entities in video games, where reactivity, modularity, and maintenance are important properties. Reactivity is crucial in these areas, as agents are often used in environments that cannot be accurately predicted. Modularity is another essential property enabling an easy transfer of sub-behavior, as certain sub-behaviors are reused in other agents. Lastly, maintenance is critical, as agents often need to be modified. These Behavior Trees are very similar to the Hierarchical Finite State Machines (HFSMs). However, they offer several advantages over them. The Behavior Trees are much more flexible in maintenance due to their top-down view. Furthermore, Behavior Trees ensure high modularity and code reusability, as nodes within the tree are implemented independently, and the tree structure defines the execution flow. Code reusability is challenging in HFSMs, where transitions must be explicitly specified, requiring prior knowledge of the overall behavior [CÖ17].

Lingua Franca is a polyglot coordination language[1]. The language focuses on retaining determinacy and handling explicit time management. That makes the language particularly well-suited for designing agents since it focuses on time-sensitive, concurrent, and reactive applications. The benefits of both approaches could be combined by introducing an explicit domain-specific language (DSL) for defining Behavior Trees in Lingua Franca.

This thesis aims to enrich Lingua Franca by extending the current Lingua Franca DSL with an explicit DSL for defining Behavior Trees. That can be achieved by extending the syntax of Lingua Franca and implementing a transformation that translates the behavior specified in the tree to the Lingua Franca semantics.

[1] https://www.lf-lang.org/

Contents

1 Introduction **1**
- 1.1 What are Behavior Trees? . . . 1
- 1.2 What is Lingua Franca? . . . 3
- 1.3 Problem Statement . . . 4
- 1.4 Outline . . . 4

2 Preliminaries **5**
- 2.1 Behavior Trees . . . 5
  - 2.1.1 Control Flow Nodes . . . 5
  - 2.1.2 Execution nodes . . . 7
  - 2.1.3 Memory nodes . . . 7
  - 2.1.4 Blackboards . . . 8
- 2.2 Lingua Franca . . . 8
  - 2.2.1 Communication between Reactors . . . 8
  - 2.2.2 Parallel Execution . . . 9
  - 2.2.3 Preamble . . . 9
  - 2.2.4 Modes . . . 10

3 Related Work **13**
- 3.1 BhTSL . . . 13
- 3.2 Behavior Designer . . . 16
- 3.3 Multi-agent coordination in Behavior Trees . . . 17

4 Concepts **21**
- 4.1 Foundations . . . 21
  - 4.1.1 Library for Behavior Trees . . . 21
  - 4.1.2 Select Mode Pattern . . . 23
  - 4.1.3 FSM Construction . . . 24
  - 4.1.4 Conclusion . . . 27
- 4.2 DSL and Transformation of Behavior Trees . . . 27
  - 4.2.1 Behavior Trees . . . 27
  - 4.2.2 Composite Nodes . . . 29
  - 4.2.3 Execution Nodes . . . 30
- 4.3 Inputs and Outputs . . . 31
  - 4.3.1 Motivation . . . 31
  - 4.3.2 Inputs . . . 32
  - 4.3.3 Outputs . . . 33
- 4.4 Local variables . . . 33
  - 4.4.1 Motivation . . . 33
  - 4.4.2 Syntax . . . 34
  - 4.4.3 Transformation . . . 35

5 Implementation **43**
5.1 Used Technologies . . . 43
5.1.1 Eclipse and Xtext . . . 43
5.1.2 Epoch . . . 43
5.2 A DSL for Lingua Franca . . . 43
5.3 Transformation to the Lingua Franca Semantics . . . 45
5.3.1 Behavior Trees . . . 45
5.3.2 Implementation of local communication . . . 46
5.4 Renaming ports . . . 46

6 Conclusion **49**
6.1 Summary . . . 49
6.2 Future Work . . . 49

Bibliography **51**

List of Figures

1.1 An Example for a Behavior Tree defining a Pacman agent [CÖ17] . . . 2
1.2 Example of a Lingua Franca program . . . 3

2.1 List of Behavior Tree node types [CÖ17] . . . 5
2.2 Behavior Tree with a Sequence memory node . . . 7
2.3 Two communicating reactors in Lingua Franca . . . 9
2.4 Lingua Franca program with shown reaction levels . . . 10
2.5 Example for a reactor with Modes. . . . 11

3.1 System architecture [OSM+20] . . . 13
3.2 BhTSL grammar [OSM+20] . . . 15
3.3 Behavior Tree view in Behavior Designer adapted from the tutorial . . . 17
3.4 Event-driven Behavior Tree Example [AGG20] . . . 18
3.5 Example of coordination for firefighter agents [AGG20] . . . 19

4.1 Pacman Behavior Tree divided in three sections [CÖ17] . . . 23
4.2 Application of Select Mode Pattern on Pacman example. . . . 24
4.3 Structure of a Behavior Tree root node in a Finite State Machine (FSM) [CÖ17] . . . 25
4.4 Behavior Tree example with a Fallback node as root [CÖ17] . . . 25
4.5 A FSM emulating the Fallback node [CÖ17] . . . 25
4.6 Transformation of a Behavior Tree with a Sequence node as root node into an FSM . . . 26
4.7 Example transformation for a Behavior Tree with a Task node. . . . 28
4.8 Example transformation for a Fallback node . . . 30
4.9 Example transformation for a Parallel node . . . 30
4.10 Example transformation for a Task node without additional sources and effects . . . 31
4.11 Transformation of a Sequence node with local communication support. . . . 36
4.12 Transformation of a Behavior Tree with local communcation between nodes with different parent nodes . . . 36
4.13 Transformation showing an edge case in the context of local communcation. . . . 37
4.14 FSM Construction for a complex Behavior Tree . . . 39
4.15 Transformation of a Behavior Tree with inputs . . . 40
4.16 Example transformation for a Behavior Tree with outputs being highlighted red . . . 41

List of Tables

List of Abbreviations

NPC	non-player character
FSM	Finite State Machine
HFSM	Hierarchical Finite State Machine
DSL	domain-specific language
BOD	blackboard observer decorator
IDE	integrated development environment
IO	Input/Output

Chapter 1

Introduction

The demand for intelligent agents in the industry has been increasing steadily for years. Whereas there were 254,000 annual installations of industrial robots in 2015, there were 517,000 in 2021, doubling the number within five years[1]. *Intelligent Agents*, autonomous computer systems performing tasks depending on the environment [WJ95], rely heavily on the definition of behavior. That raises the question of how behavior is specified. A relatively new model for these behavior declarations is becoming increasingly important and popular: The Behavior Trees. Behavior Trees have their origins in the video game industry for defining the behavior of non-player characters (NPCs). NPCs are entities within video games whose behavior is exclusively controlled by software [Ani20]. Behavior Trees are gaining popularity in the robotics industry as well [ISS+22]. In the past, FSMs were standardized because robots were used for executing less specialized tasks in a predictable environment. However, the next generation of industrial robots is trained to perform many different tasks in a less predictable environment [CÖ17]. For this purpose, Behavior Trees are particularly suitable, as they ensure reactivity and modularity.

1.1 What are Behavior Trees?

For better understanding, let us introduce a running example here, adapted from Colledanchise and Ogren [GBJ+20]. In Figure 1.1, a Behavior Tree is depicted, which could be used for modeling the behavior of a Pacman. In that Behavior Tree, four types of nodes are identifiable.

Each node of a Behavior Tree has three predefined return values: *success*, *failure*, and *running*. The *Fallback* node, visualized with a question mark, executes its children from left to right until one child returns success. Then it aborts the execution of its children and returns success. However, the Fallback node returns failure if all children return failure. The *Sequence* node, denoted with a right-pointing arrow, executes its children from left to right until one child fails. Then the Sequence node aborts the execution and returns failure. If all children return success, the Sequence node itself also returns success. These two node types belong to the control flow nodes, which direct the execution flow and indicate the tree's traversal according to their children's output. *Task* nodes, represented in a rectangle, and *Condition* nodes, visualized with an oval-shaped box, belong to the so-called execution nodes. Execution nodes allow the intelligent agent to perform actions and check conditions within the agent's scope.

When executing the Behavior Tree, the root node is arrived first. Since it is a Fallback node, the first child, the Sequence node, is executed. That node also starts with the execution of its first child, the Condition node labeled with `Ghost Close`. If this Condition node returns failure, the Sequence node returns failure without executing its next child. If this Condition node returns success, the Sequence node will execute the next child, the second Fallback node. If the condition node `Ghost Scared` returns success, the task node `Chase Ghost` is executed. If one of both returns failure, then the Task node `Avoid Ghost` is executed. Eventually, if the `Chase Ghost` or the `Avoid Ghost` node returns success, then the

[1] https://www.statista.com/statistics/264084/worldwide-sales-of-industrial-robots/

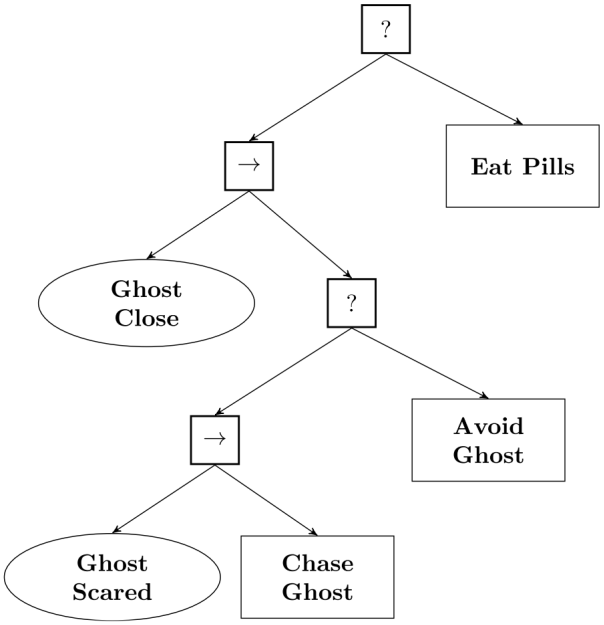


Figure 1.1. An Example for a Behavior Tree defining a Pacman agent [CÖ17]

execution loop for the tick is finished, and the Behavior Tree returns success. However, if one of the mentioned nodes returns failure, the first Fallback executes the Eat Pills node. The evaluation of the tree then solely depends on the return of the node labeled Eat Pills.

Behavior Trees are hierarchical models that describe decision-making. Each Behavior Tree has a root node, a node without a parent. Every other node has exactly one parent. Internal nodes are called control flow nodes, while leaf nodes are called execution nodes [CÖ17]. Behavior Trees are tick-based, which means that the tree is evaluated in discrete time steps, where each tick corresponds to one iteration of the tree evaluation loop. During each tick, the tree is traversed from the root node down to the leaf nodes, in which the desired execution takes place [GBJ+20].

One of their primary use cases is in defining the behavior of intelligent agents, such as robots and other embedded systems. FSMs were standardized for a long time for this purpose. However, Behavior Trees are being increasingly used [ISS+22] because they support higher modularity and code reusability than HFSMs and FSMs. That is because the nodes of a Behavior Tree can be implemented independently of the entire context of behavior. HFSMs and FSMs, transitions to other nodes are explicitly specified, requiring prior knowledge of the overall behavior. That is not the case with Behavior Trees, as the control flow nodes automatically give the execution flow. Therefore, Behavior Trees are more modular, allowing partial behavior to be implemented, tested, and deployed separately from the entire behavior. Code reusability is also more accessible, as partial behaviors are implemented independently of external nodes. Thus, behavior modeled once through a Behavior Tree can be reused by appending it to another, simplifying extending behavior through Behavior Trees. Unlike HFSMs, Behavior Trees have a native top-down view and a clearly defined top-to-bottom and left-to-right traversal. Thereby, Behavior Trees are easier to read and maintain than HFSMs [CÖ17]. Especially in complex systems, maintaining HFSMs can be difficult because the number of possible transitions increases exponentially with each added node.

```
target C

main reactor {
      reaction (startup) {=
                  printf("Hello World\n");
            =}
}
```

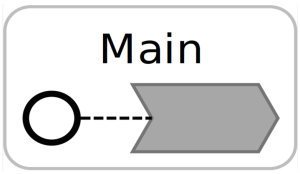


Figure 1.2. Example of a Lingua Franca program

1.2 What is Lingua Franca?

Figure 1.2 shows a minimal functional Lingua Franca program. Lingua Franca programs are created in text and can also be visualized graphically. The Figure 1.2 shows the code and the corresponding graphical representation. A Lingua Franca program requires the specification of a target language that defines the programming language used in the code for the reactions. In the given example, the defined target language is C. The Lingua Franca code generator will translate the Lingua Franca Program into a C program in the compilation process.

Lingua Franca programs consist of reactors, one of them being the main reactor. Our example program only defines the main reactor, which is comparable with the main method in traditional programming languages. The main reactor in our example only consists of a reaction triggered by the event `startup`. The execution of `printf("Hello World\n")` embodied by the reaction depends on whether the user-defined trigger event `startup` triggers the reaction. When the example Lingua Franca program is executed, the main reactor starts and the `startup` event triggers the reaction. By that, the main reactor prints `Hello World`. In Figure 1.2, the main reactor is visualized as a rectangular box, in which chevron shapes represent the corresponding reaction. The startup trigger is denoted as a circle attached to the reaction on the left.

Lingua Franca is a coordination language for concurrent and time-sensitive applications[2]. The language focuses on retaining determinacy and handling explicit time management. Lingua Franca is not a standalone programming language, as a Lingua Franca program is translated into a target language program. The target languages currently supported are C, C++, Python, TypeScript, and Rust. A Lingua Franca Program consists of reactors that can exchange data through messages. Reactors can contain an arbitrary number of reactions where code written in the target language by the user is executed. Reactors in Lingua Franca are defined, among other components, by *Reactions*. Other components will be covered in Chapter 2.

Lingua Franca is particularly well-suited for defining behavior. The language provides a high degree of modularity, as a Lingua Franca program is divided into individual reactors that can be implemented, tested, and deployed independently. Since reactors can be specified as extensions or subclasses of one another, code reusability is also integral to Lingua Franca. Additionally, Lingua Franca is a highly reactive language.

[2]https://www.lf-lang.org/docs/handbook/overview?target=c

1.3 Problem Statement

Implementing Behavior Trees in Lingua Franca could provide several benefits. Both Behavior Trees as a hierarchical model and the coordination language Lingua Franca possess a high degree of modularity. Additionally, Lingua Franca possesses an automatic synthesis of graphical views corresponding to the textual definition. The graphical view is fundamental as the visual component plays a crucial role in Behavior Trees. Moreover, introducing Behavior Trees natively in Lingua Franca would provide an understandable and readable interface for designing behavior. A further advantage would be the resulting high portability achieved by combining Behavior Trees and Lingua Franca. Behavior Trees can be implemented in various programming languages. Given that Lingua Franca is a platform-independent language, it would be possible to directly implement Behavior Trees in one of the different supported target languages. Furthermore, Behavior Trees also support the parallel execution of tasks. Lingua Franca is especially focused on concurrent applications, and therefore, combining the two concepts would lead to the efficient deterministic execution of agents with parallel tasks. Besides, Lingua Franca is a textual language, and practitioner experience has shown that working with systems textually is more efficient than working visually [HLF+22]. Therefore, the combination of textual editing of Behavior Trees with visual representation would be a highly efficient way of describing behavior. This thesis aims to extend the Lingua Franca language with an explicit textual DSL for Behavior Trees. This goal can be achieved by first designing a DSL. Subsequently, a transformation can be implemented that converts a Behavior Tree, which has been declared with the designed DSL, back into the semantics of the existing Lingua Franca language.

1.4 Outline

The next chapter will provide further details on Behavior Trees and Lingua Franca. In Chapter 3, related work addressing Behavior Trees, DSLs, and coordination between agents will be discussed. Chapter 4 focuses on designing a DSL and transformation patterns for the Behavior Trees in Lingua Franca. Afterward, Chapter 5 deals with the implementation of the DSL and the transformation. Finally, Chapter 6 summarizes the results of the thesis and provides an outlook on future work.

Chapter 2

Preliminaries

2.1 Behavior Trees

Behavior Trees are hierarchichal data structures that model decision-making. Each Behavior Tree has a root node, which is a node without a parent. Every other node has exactly one parent. Each node of a Behavior Tree has three predefined return values, which are success, failure and running. Internal nodes are called control flow nodes while leaf nodes are called execution nodes [CÖ17]. The Figure 2.1 outlines the standard node types of a Behavior Tree. As mentioned in Chapter 1, Behavior Trees are tick-based. That means the tree is traversed once and returns a result on each tick. Upon ticking a node, the corresponding actions are executed. During a tick, the tree's traversal is guided by the control flow nodes. The execution nodes are used for performing actions and checking conditions.

2.1.1 Control Flow Nodes

The node types in the first four lines of Figure 2.1 namely Sequence, Fallback, Parallel and Decorator are the so called control flow nodes which direct the flow of control and indicate the tree's traversal according to the output of their children. Each of these nodes have atleast one child node.

Fallback

As mentioned in Chapter 1, Fallback nodes execute their children chronologically until a child returns success. Once a child node returns success, the Fallback node returns success, and the execution of the Fallback ends. That means that children nodes that are to the right of the succeeded child are not executed. If any children returns failure or running the Fallback node also returns failure or running, respectively. The Listing 2.1 illustrates this algorithm with pseudocode.

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Decorator	◊	Custom	Custom	Custom
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never

Figure 2.1. List of Behavior Tree node types [CÖ17]

```
for i <- 1 to N :
    childstatus <- Tick(child(i))
    if childstatus = Running :
        return Running
    else if childstatus = Failure
        return Failure
return Success
```

Listing 2.1. Pseudocode of a Fallback node with N children [CÖ17]

```
for i <- 1 to N :
    childstatus <- Tick(child(i))
    if childstatus = Running :
        return Running
    else if childstatus = Success
        return Success
return Failure
```

Listing 2.2. Pseudocode of a Sequence node with N children [CÖ17]

Sequence

Sequence nodes execute their children sequentially until a child node returns failure, as stated in Chapter 1. When a child node returns failure, the Sequence node returns success, and the execution of the Sequence terminates. Consequently, any child node located to the right of the failed child node is not executed. If any child node returns failure or running, the Sequence node also returns failure or running, respectively. The algorithm shown in Listing 2.2 demonstrates the Sequence semantics.

Parallel

Parallel nodes do not execute their children sequentially but rather all at once. For this type of node, an integer M must be specified, which is used to determine the return value. If $\geqslant$ M children succeed, then the Parallel node returns success. If there are > N-M failed children, it returns failure, and otherwise, it returns running.

```
for 1 <- to N do parallel :
    childstatus(i) <- Tick(child(i))
successCounter <- numberOfSucceedChildren(childstatus)
if successCounter >= M :
    return Success
failureCounter <- numberOfFailedChildren(childstatus)
if failureCounter > N-M :
    return Failure
return Running
```

Listing 2.3. Pseudocode of a Parallel node with N children and success threshold M [CÖ17]

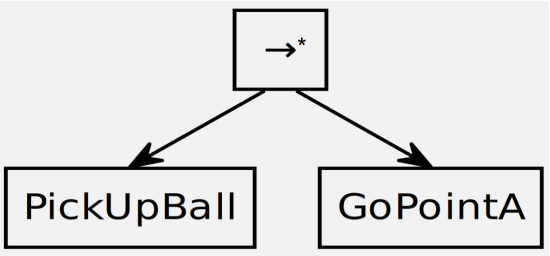


Figure 2.2. Behavior Tree with a Sequence memory node

Decorator

The Decorator node, a control flow node, alters the outcome of its single child node based on user-defined criteria [CÖ17]. Users can also specify rules for the composite node so it ticks the child node accordingly. An example of a decorator node would be a *MaxNTries* decorator, which ticks the child N times until it returns success. If the child node does not return success, the decorator can be defined to return failure. Several other types of decorators exist, such as an invert decorator that reverses the success or failure status of the child node.

2.1.2 Execution nodes

The next two node types are the execution nodes which allow the intelligent agent to perform actions and check conditions within the scope of the agent. Task nodes, also known as *Action* nodes, are used for executing commands given by the user. An example of a Task would be a command that instructs a robot to move from point A to point B. Action nodes return success upon completion, failure if the command could not be completed and running during completion.

Condition nodes are used to evaluate conditions within the agent's scope. An example could be a condition that checks whether the agent is in motion or a door is open. In contrast to other nodes, these nodes never return running; instead, they return success if the condition is true or failure if it is false.

2.1.3 Memory nodes

Memory nodes are composite node extensions used to prevent the re-execution of nodes. These nodes are marked with an asterisk [CÖ17]. Figure 2.2 shows an example of a Sequence memory node. The Behavior Tree defines the behavior of an agent tasked with picking up a ball and bringing it to point A. Once the node `PickUpBall` returns success, the Sequence node ticks the `GoPointA` node. The node `GoPointA` returns running if the agent is on his way to point A and thus the execution for the tick is done. In the next tick, normal Sequence nodes would attempt to execute the first child node `PickUpBall` again. Sequence nodes with memory prevent the execution of an already executed child node by using a state variable that remembers the last executed child node. In the example, the Sequence node would remember that `GoPointA` was executed last and will tick that node, skipping the `PickUpBall` node. Parallel and Decorator nodes dont have a version with an memory extension since both of them do not execute their children sequentially.

```
[main] reactor <class-name> {
    input <name>:<type>
    output <name>:<type>
    reaction(<triggers>) [<sources>] [-> <effects>] {= ... body ...=}
    <instance-name> = new <class-name>()
    <instance-name>.<port-name> -> <instance-name>.<port-name> % used for connections
}
```

Listing 2.4. General Form of a reactor

2.1.4 Blackboards

Blackboards are utilized for communication between Behavior Tree nodes. These blackboards are global key-value data structures that every node within the Behavior Tree and the environment can write and read to [GBJ+20]. For example, a `FindEnemy` Task node may store a value to the blackboard with the call `blackboard[enemyLocation] = (0, 0, 0)`, and an `AttackEnemy` Task node could retrieve that value with the call `blackboard[enemyLocation]`.

2.2 Lingua Franca

After providing basic information about Lingua Franca in Chapter 1, this section offers more details, gathered from the documentation[1] of the language.

2.2.1 Communication between Reactors

In Lingua Franca, reactors have a general structure as shown in Listing 2.4, which has been broken down into the important components for this thesis. Figure 2.3 shows an example of a Lingua Franca program in which two reactors communicate. The main reactor is the top-level reactor that is capable of initializing and connecting reactors. This reactor can be compared to the main method in conventional programming languages. In the example, the main reactor initializes two reactors, `Sender` and `DoublePrinter`. The Sender reactor has an output port `out` with the type integer and a reaction with the trigger `startup`. This trigger causes the reaction to execute the target language code within the `{= ... =}` body. The reaction of Sender is triggered by startup, meaning that the reaction is executed with the start of the program. After the `->` symbol, so-called effects can be declared. These are references to output ports on which the reaction wants to write on. In the example, the reaction wants to output `10` to the port out. Reactor DoublePrinter has an input `in` with the type integer and a reaction triggered by that input. This reaction prints the doubled value of the input integer to the console. The main reactor can connect the Sender's output with the DoublePrinter's input. When the program is executed, Sender sends the value 10 to the reactor DoublePrinter. The reaction with the trigger `in` of the DoublePrinter reactor is executed, and 20 is printed to the console.

Reactions can have not only effects and triggers but also sources. Sources are ports that can be read by the reaction but do not trigger the reaction. Suppose the reaction declaration of the DoublePrinter would be replaced with the reaction shown in Listing 2.5. Then the reaction would also be executed in the absence of input `in`. Since no value has been written to the input in that case, the default value for an integer, namely 0, would be used for the in input. Thus 0 would be print in the console.

[1] https://www.lf-lang.org/docs/handbook/overview?target=c

```
target C

main reactor {
   s = new Sender()
   dp = new DoublePrinter()

   s.out -> dp.in
}

reactor Sender {
   output out:int

   reaction (startup) -> out {=
      SET(out, 10);
   =}
}

reactor DoublePrinter {
   input in:int

   reaction (in) {=
      printf("%d", in->value * 2);
   =}
}
```

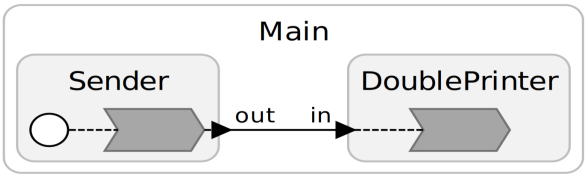


Figure 2.3. Two communicating reactors in Lingua Franca

```
reaction (startup) in {=
    printf("%d", in->value * 2)
=}
```

Listing 2.5. Reaction with in as a source

2.2.2 Parallel Execution

Lingua Franca is a dataflow language that guarantees deterministic concurrent execution. During program analysis, the Lingua Franca compiler analyses which reactions depend on which other reactions. Reactions that do not depend on any other reactions are referred to as *level-0* reactions. Reactions that process data from level-0 reactions are level-1 reactions. This process is carried on for each reaction until the Lingua Franca compiler has computed a level for each reaction. During program execution, reactions with the same level are executed concurrently. The Figure 2.4 shows an example program in which the reaction levels can be seen. The reaction of `Reactor0` and `Reactor1` are executed concurrently, while the reaction of `Reactor2` is executed afterward.

2.2.3 Preamble

The Lingua Franca program enables the execution of target language code in reactions. This feature allows for the invocation of libraries or the definition of custom methods within reactions. However,

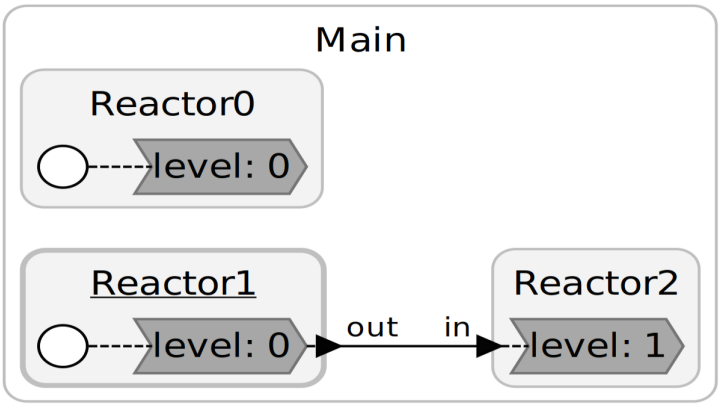


Figure 2.4. Lingua Franca program with shown reaction levels

```
target C
preamble {=
        #include <math.h> // included for sqrt() method
        int ownMethod() {
                return 42;
        }
=}
main reactor {
        reaction (startup) {=
                printf("%f ", sqrt(9));
                printf("%d \n", ownMethod());
        =}
}
```

Listing 2.6. Example for a Preamble definition in a Lingua Franca program

methods defined within a reaction are only accessible within the scope of the reactor, and another reactor cannot access these methods. To define global methods or include libraries accessible to all reactors in the program, the Preamble section can be used. Listing 2.6 shows an example of such a Preamble declaration. The Preamble was written outside of reactors, allowing all reactions of all reactors within the program to access the library and the defined method. The program in the Listing 2.6 will print `3.000000 42`.

2.2.4 Modes

Reactors in Lingua Franca can define Modes, which can be compared to states in FSMs. These Modes can have their own reactions, reactor instantiations, and connections. In each logical time frame, precisely one mode is active. That means that the execution of all other modes is paused. One mode is declared the *initial mode*, which resembles the start state. Mode switches can be explicitly performed by user code. Figure 2.5 shows a Lingua Franca program with the main reactor having two Modes. Upon execution of the program, the reaction of the initial mode is first executed, and `ONE!` is printed. Then, the code `SET_Mode(ModeTwo)` can be used to switch to the `ModeTwo` mode. The reaction in ModeTwo then

```
target C
main reactor {

   initial mode ModeOne {
      reaction (startup) -> ModeTwo {=
         printf("ONE! \n");
         SET_MODE(ModeTwo);
      =}
   }

   mode ModeTwo {
      reaction (startup) {=
         printf("TWO! \n");
      =}
   }
}
```

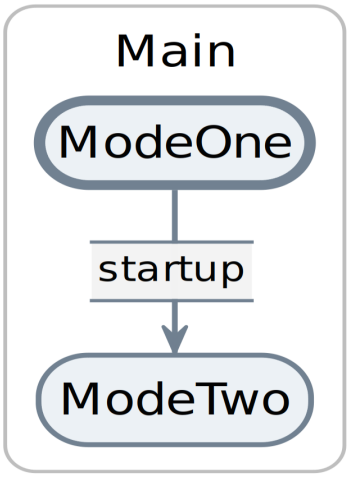


Figure 2.5. Example for a reactor with Modes.

prints `TWO!`. To perform a mode change, the name of the mode to which the switch should be made must be specified as an effect.

Chapter 3

Related Work

Behavior Trees represent an active research field, with numerous approaches being developed for defining them. This chapter will introduce two approaches showcasing different Behavior Tree design methods. Moreover, plenty of other areas of Behavior Trees are being researched. One is how coordination between agents, defined by Behavior Trees, can be implemented. To this end, a paper focusing on coordinating agents that utilize Behavior Trees is presented.

3.1 BhTSL

BhTSL is a DSL for defining Behavior Trees [OSM+20]. In addition to defining the DSL, an architecture was also implemented in Python, as depicted in Figure 3.1. The system architecture expects a text file defining the behavior based on the DSL specification. The compiler comprises four modules, LEXER, PARSER, PARSED TREE GENERATOR, as depicted in the green box in the Figure. The generator consists of two sub-generators, one of them being the PYTHON GENERATOR. This generator is responsible for producing a Python code fragment implementing the wanted behavior. The generated code fragment can be imported by any Python application that intends to use it. The LATEX GENERATOR, the second of the two sub-generators, produces latex code for drawing the defined Behavior tree. Each text file written in the BhTSL DSL language defines exactly one behavior. These text files contain three components: *Behavior*, *Definitions*, and *Code*. The Behavior component comprises the definition of the main Behavior Tree. The Definitions component is optional and can be empty. This component can contain node definitions which can be referenced in the main Behavior Tree or in other node definitions. The Code component contains Python code for each execution node. In Figure 3.1 an example for our pacman behavior definition in BhTSL is presented.

The BhTSL syntax is specified as a context free grammar which is shown in Figure 3.2. Using the root rule, it is possible to determine the exact structure of a text file in BhTSL. A text file contains exactly one main Behavior Tree, which is covered by the behavior rule, and a section for code. Additionally, it is possible to declare a node definition before the Behavior Tree definition or a set of node definitions,

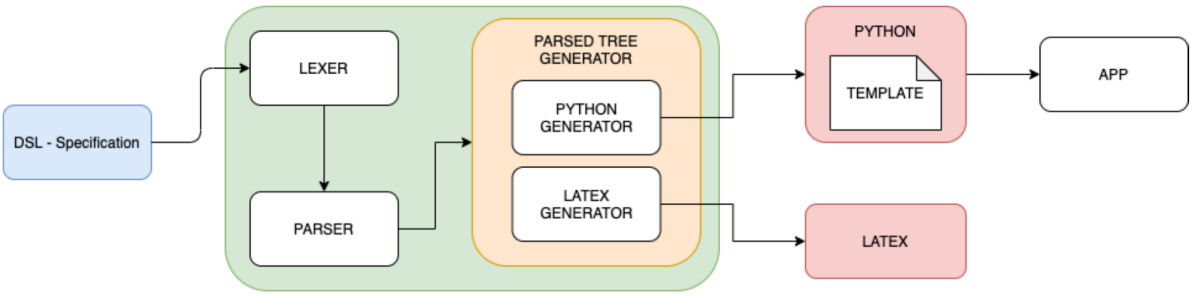


Figure 3.1. System architecture [OSM+20]

```
# main Behavior Tree
behavior : [
        selector : [
                sequence : $seq1,
                action : $eat_pills
        ]
]

# Definitions
sequence seq1 : [
        condition : $ghost_close,
        selector : [
                sequence : [
                        condition : $ghost_scared
                        action : $chase_ghost
                ],
                action : $avoid_ghost
        ]
]

# Code

def ghost_close(pacman):
        return pacman.ghost_close

def ghost_scared(pacman):
        return pacman.ghost_scared

def chase_ghost(pacman):
        pacman.chase_ghost()
        return SUCCESS

def avoid_ghost(pacman):
        pacman.avoid_ghost()
        return RUNNING

def eat_pills(pacman):
        pacman.eat_pills()
        return RUNNING
```

Listing 3.1. Pacman behavior definition code in BhTSL

```
root : behavior CODE
     | behavior definitions CODE
     | definition behavior CODE

behavior : BEHAVIOR ':' '[' node ']'

node : SEQUENCE ':' '[' nodes ']'
     | SEQUENCE ':' VAR
     | MEMORY SEQUENCE ':' '[' nodes ']'
     | MEMORY SEQUENCE ':' VAR
     | SELECTOR ':' '[' nodes ']'
     | SELECTOR ':' VAR
     | MEMORY SELECTOR ':' '[' nodes ']'
     | MEMORY SELECTOR ':' VAR
     | PROBSELECTOR ':' '[' prob_nodes ']'
     | PROBSELECTOR ':' VAR
     | MEMORY PROBSELECTOR ':' '[' prob_nodes ']'
     | MEMORY PROBSELECTOR ':' VAR
     | PARALLEL ':' INT '[' nodes ']'
     | PARALLEL ':' VAR
     | DECORATOR ':' INVERTER '[' node ']'
     | DECORATOR ':' VAR
     | CONDITION ':' VAR
     | ACTION ':' VAR

nodes : nodes ',' node
      | node

prob_nodes : prob_nodes ',' prob_node
           | prob_node

prob_node : VAR RIGHTARROW node

definitions : definitions definition
            | definition

definition : SEQUENCE NODENAME ':' '[' nodes ']'
    | SELECTOR NODENAME ':' '[' nodes ']'
    | PROBSELECTOR NODENAME ':' '[' prob_nodes ']'
    | PARALLEL NODENAME ':' INT '[' nodes ']'
    | DECORATOR NODENAME ':' INVERTER '[' node ']'
```

Figure 3.2. BhTSL grammar [OSM+20]

which must then be listed after the Behavior Tree definition. The Behavior Tree may only contain a single node, which then becomes the root node of the Tree. As the rule `node` shows, this can be one of the execution nodes, which reference a Python function in the code section. However, it can also be a control flow node. This node type can either be specified directly in the Behavior Tree or reference a node definition from the definitions section.

In conclusion, BhTSL is a DSL designed for implementing Behavior Trees, for which an architecture is provided that offers automatic Python code generation and Behavior Tree visualization. One advantage of the language and architecture is that they possess high modularity and code reusability through the ability to reference node definitions. Furthermore, the support for many different node types and the automatic generation of LaTeX code for visualizing the Behavior Tree is beneficial.

On the other hand, there are also drawbacks. For example, communication between nodes within the Behavior Tree only works via global variables of the python program, not allowing variable declarations with a scope restricted to specific nodes. Additionally, the static Behavior Tree visualization could lack essential features to help develop behavior. For example, in an interactive visualization, it could be possible to display and hide the code executed by an Execution node by clicking on it.

3.2 Behavior Designer

Behavior Designer is a tool developed by Opsive that allows designing of behavior for intelligent agent entities in the game engine *Unity* in a visual way[1]. The tool is directly integrated into the Unity system, meaning components such as NPC animations, game physics, and other related elements from the system and game environment can be embedded directly in the editor. Through its direct integration, the tool allows for deploying, testing, and modifying behaviors directly within the game environment. Other features include a visual debugger and local and global variables support.

Behavior Designer provides a graphical integrated development environment (IDE) that can be used to design Behavior Trees. This IDE includes several features that aim to simplify this process. Using this IDE, properties of the Behavior Trees can be modified, settings for nodes can be adjusted directly, and variables of the Trees can be managed. Furthermore, the graph view supports several intuitive operations as well. For example dragging, copy and pasting nodes or making connections by connecting nodes is feasible. This makes creating and adjusting the Behavior Tree fairly easy. Additionally, the IDE is suitable for debugging in the Unity environment through features such as start, stop, and pause. The distinctive feature of the tool is its support for visual creation and modification of behaviors without the need to write a single line of code. That is possible due to the tool's implementation of many predefined nodes for Unity, which allow for describing complex behaviors without requiring programming knowledge[2]. For instance, the tool offers 17 predefined tasks for determining movement and 13 for tactical decision-making. However, manual behavior coding is also supported by the integrated code editor.

Let us introduce an example here, adapted from a video published by the developers of the tool[3]. An intelligent agent is to be implemented to navigate to point A. If the agent encounters an object on its way to point A, it will follow the object until it reaches it. The user can add the required nodes for the Behavior Tree into the graph area. The Figure 3.3 shows the created Behavior Tree within the Behavior Designer IDE. Furthermore, the user can configure the nodes of the Tree. For that, there are many different configuration options available. For example, the user can specify a field of vision for the "Can See Object" node. For the first "Seek" node, the object that the agent should follow must be

[1] https://opsive.com/assets/behavior-designer/
[2] https://opsive.com/solutions/ai-solution/
[3] https://opsive.com/videos/behavior-designer-overview/

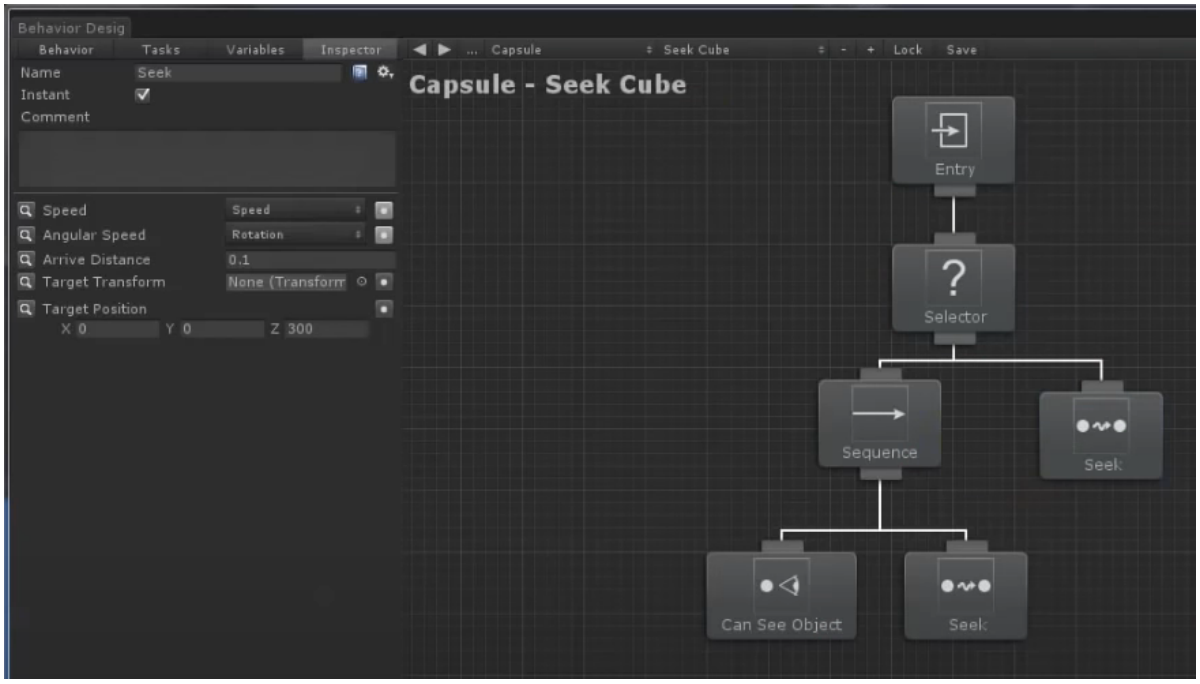


Figure 3.3. Behavior Tree view in Behavior Designer adapted from the tutorial

indicated. Finally, the coordinates of point A must be specified on the right-hand side for the "Seek" node. Other customizable settings for the seek node include movement and rotation speed. All of these customizations make the process of designing Behavior Trees very easy and intuitive. After implementating a Behavior Tree, it is possible to test it directly in Unity. Upon execution of the tree, the return status of the nodes is synchronously displayed in the Tree view. Running nodes are highlighted in green, while successful nodes are represented with a checkmark and failed nodes with a cross.

In Conclusion, Behavior Designer is a tool made for Unity which supports a visual way of implementing Behavior Trees. One advantage of Behavior Designer is its intuitive, user-friendly interface and the predefined nodes which makes creating and editing Behavior Trees easy. The visual debugging environment, coupled with additional debugging features, also makes working with Behavior Trees easier. Furthermore, the tool possess high modularity since other Behavior Trees can be referenced within another Behavior Tree. Additionally, Behavior Designer supports both local and global variables, making it easier to manage and reuse data across nodes and other Behavior Trees. The implementation of Behavior Trees in this thesis also supports a graphical view, simplifying the designing of behavior.

3.3 Multi-agent coordination in Behavior Trees

Behavior Trees are a powerful tool for designing the behavior of individual agents. However, it is often desired that agents defined by Behavior Trees coordinate to achieve common goals. Agis et al. present a concept for coordinating such agents [AGG20].

Video games are an application area where such coordination between intelligent entities is desired. Coordinated behavior of NPCs can make the gaming experience more immersive for the player.

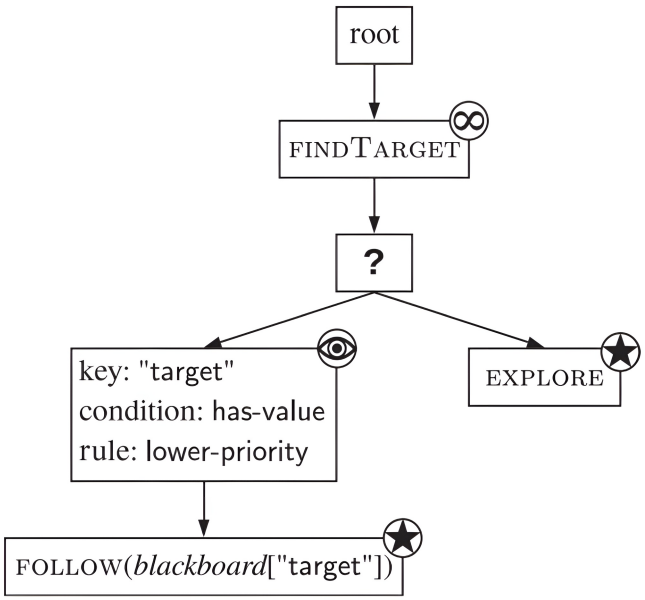


Figure 3.4. Event-driven Behavior Tree Example [AGG20]

Similarly, such coordination can be advantageous in robotics when multiple robots are required to perform a task. However, Behavior Trees typically focus on describing the individual behavior of agents, and coordination between NPCs is often achieved through hard-coding. That makes designing behavior less visually intuitive, scalable, and reusable. To address this issue, the proposed solution extends Behavior Trees, thus retaining the strength of Behavior Trees while defining coordination.

The proposed solution utilizes *event-driven Behavior Trees* with *blackboards*, *service*, and blackboard observer decorator (BOD) nodes for the extension. In Event-driven Behavior Trees, nodes can react to events and be aborted. Every event-driven Behavior Tree maintains a blackboard consisting of a set of key-value pairs that all nodes within the tree can access. Service nodes are nodes that have a method and a frequency and have a control flow node as a child. When a service node is executed, it ticks its child. As long as a successor node of the control flow node returns running, the service node executes the method with the specified frequency. If the child node returns success or failure to the service node, the service node aborts the execution of the method and returns the return value of the composite node.

An example would be a service node that frequently executes a `updateTargetPos()` method. This method could then update the value of `blackboard["targetPos"]`, allowing the service node's descendant nodes to access the target's current position and, for example, approach it. BOD nodes contain a blackboard key, a condition, and an abort rule. When a BOD node is ticked, it ticks its child if the condition is fulfilled; otherwise, it returns failure. An example of the application of the node is shown in Figure 3.4.

Service nodes are denoted by an infinity symbol, BOD nodes by an eye, and Task nodes by a star. When the tree is ticked, the FINDTARGET node is ticked first, which executes the corresponding method and ticks the Fallback node. The Fallback node then ticks the BOD node. We assume that `blackboard["target"]` does not yet have a value. The `has-value` condition is not fulfilled, and the BOD node returns failure. According to the `lower-priority` rule, all nodes descendants of the BOD's first composite ancestor to the BOD's right are labeled lower-priority nodes. In the example, only the

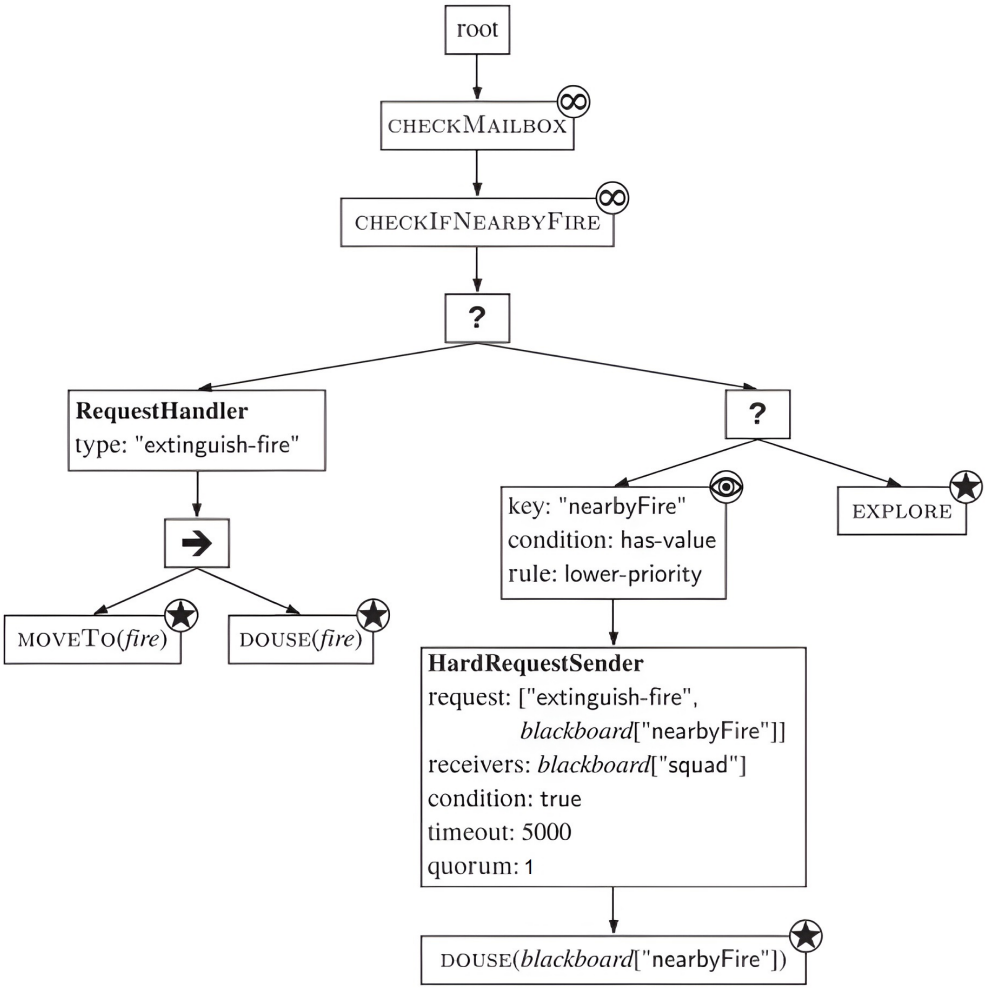


Figure 3.5. Example of coordination for firefighter agents [AGG20]

EXPLORE node, which is ticked next, is one. Since this node is a lower-priority node, the BOD node concurrently observes the blackboard entry with the key `"target"`. If the method corresponding to the FINDTARGET node now updates the value of `blackboard[target]`, all BODs observing this entry are notified. The running node EXPLORE is then aborted, and execution continues with the BOD node, which executes the FOLLOW(*blackboard*["target"]) node.

The Behavior Tree in Figure 3.5 presents a possible implementation for coordinating firefighter agents. To explain coordination, let us assume the tree defines the behavior of two firefighter agents, `f_1` and `f_2`. A firefighter agent should only extinguish a particular fire if he receives support from at least one other firefighter. When the tree of `f_1` is ticked, CHECKMAILBOX is executed first. The node repeatedly checks whether another agent sent a message. Also, the CHECKIFNEARBYFIRE node is ticked, which searches the agent's surroundings for a fire. Next, the Fallback node is ticked, which first ticks the RequestHandler node. *Request Handler* nodes are specifications of BOD nodes that additionally comprise a type and fields for parameters. These nodes process messages. As there is

no message for f_1 in the mailbox, the Fallback node is executed. Here, the BOD node returns failure, and EXPLORE is executed. If the method corresponding to the CHECKIFNEARBYFIRE node updates the value blackboard["nearbyFire"], EXPLORE is aborted, and the BOD is executed, which executes the HardRequestSender node.

Hard Request Sender nodes are responsible for sending messages. A message consists of a 4-tuple mes = (s, req, c, t), where s stands for the sender of the message, req for a request, c for a condition, and t for a timeout threshold. A request is a pair req = (type, parameters), where the type contains the node's name within the receiver's tree that s wants the receiver to execute. Parameters is a list of values for the node's fields, which can be empty. The agent sends a message with the values defined in the HardRequestSender node to all other firefighter agents included in the list blackboard["squad"]. Hard Request Sender stop their execution here and wait for quorum many confirmations. If the timeout passes, the node returns failure, and execution resumes. We assume that fire agent f_2 is currently running EXPLORE, and the message from f_1 arrives in the mailbox. The CHECKMAILBOX method now checks if the message's condition holds and if the timeout has passed. The condition true is fulfilled, and we assume that the timeout has not passed. Now, f_2 sends a confirmation and continues its execution normally. f_1 has now received enough confirmations and executes DOUSE(*blackboard*["nearbyFire"]). Furthermore, f_1 sends a reconfirmation to f_2, which aborts the currently running node and executes the child of the RequestHandler. The Sequence executes MOVETO(*fire*) and DOUSE(*fire*) with fire being the value f_1 wrote into the parameter. Both firefighters now attempt to extinguish the same fire. Additionally, there is a third coordination node called *Soft Request Sender*. That node behaves similarly to the Hard Request Sender node. The only difference is, that with the Soft Request Sender the execution does not pause and no confirmations are sent; instead, the execution continues normally.

There are numerous additional possibilities for controlling coordination. For instance, specific coordinations can be prioritized. That can be achieved by sorting the sub-trees from left to right according to their priority level, such that the highest-priority sub-trees are situated on the far left. Additionally, the authors propose a method for pausing and resuming the CHECKMAILBOX method, respectively. One possible use case would be when critical nodes are being executed that should not be aborted. In such a scenario, the CHECKMAILBOX method could be paused before the critical sub-tree is executed and then reactivated afterward. Expanding event-driven Behavior Trees with three new nodes, a priority queue called "mailbox," and a way for trees to send messages maintain the strengths of Behavior Trees when defining coordination.

This paper presents a complex way of handling communication between different agents. In Section 4.3, a much simpler and more intuitive way of communication is explained and implemented.

Chapter 4

Concepts

This thesis aims to implement a DSL for Behavior Trees in Lingua Franca while preserving the benefits of both Lingua Franca and Behavior Trees. A conceptual approach is essential to solve this problem. By dividing the problem into multiple sub-problems, it can be ensured that a complete and closed solution is implemented. In our case, the problem can be decomposed into two sub-problems, designing a DSL for Behavior Trees in Lingua Franca and implementing a transformation which translates the tree to Lingua Franca semantics. This chapter presents the concepts pursued to achieve this goal. The first section discusses the different transformation patterns that could be used. The second section presents the design of the DSL extension for the Lingua Franca syntax and the transformation patterns. The third section explains how the transformation is modified to extend the tree with Input/Output (IO), and the final section illustrates how local communication is implemented within the tree.

4.1 Foundations

This section presents solutions to the problem of transforming Behavior Trees into Lingua Franca semantics. The goal is to evaluate a solution which ensures that the implemented Behavior Trees have a high level of modularity, are supported by the implemented diagram view for Lingua Franca, and can utilize Lingua Franca's built-in concurrent execution feature. The following presents various methods of transforming behavior trees into Lingua Franca and discusses the applicability of these approaches. In the end, a transformation pattern will be chosen that will be used for the implementation.

4.1.1 Library for Behavior Trees

One methodology for achieving this would be to declare Behavior Trees using a library. With method calls to the library, it would be possible to define the entire Behavior Tree along with all of its nodes and the code for each of the execution nodes. The Listing 4.1 shows an example declaration of a Behavior Tree for the Pacman example with the *py_trees* libary. An example for defining a Behavior Tree with a library is introduced. To execute the code for the Behavior Tree, the user must first create a reactor `PacmanBehaviorTree` for the Behavior Tree. A *Preamble* is used to import the library. More details on Preambles are provided in Chapter 2. Within `PacmanBehaviorTree`, a reaction must be created that is fired upon the trigger `startup`. The Behavior Tree code can now be inserted into the code area `{= ... =}`.

One advantage of this approach is that it is possible to use different Behavior Tree libraries, which can be helpful when, for example, a library with more features is needed. Moreover, this approach does not require designing a DSL or modification of the Lingua Franca semantics, as Lingua Franca already supports the execution of target language code with the use of libraries.

However, a disadvantage of this approach is that it does not allow for modularity, as the library translates the Behavior Tree as a whole and does not convert the node modularly, which makes code reusability within other Behavior Tree reactors impossible. Moreover, the advantages offered

```
target Python
preamble {=
    import py_trees
=}
main reactor {
    bt = new PacmanBehaviorTree();
}
reactor PacmanBehaviorTree {
    reaction (startup) {=
        selector1 = py_trees.composites.Selector("Selector1")
        sequence1 = py_trees.composites.Sequence("Sequence1")
        selector2 = py_trees.composites.Selector("Selector2")
        sequence2 = py_trees.composites.Sequence("Sequence2")

        class EatPills(py_trees.behaviour.Behaviour):     #definition of behavior
        eat_pills = EatPills()
        class GhostClose(py_trees.behaviour.Behaviour):     #definition of behavior
        ghost_close = GhostClose()
        class GhostScared(py_trees.behaviour.Behaviour):     #definition of behavior
        ghost_scared = GhostScared()
        class ChaseGhost(py_trees.behaviour.Behaviour):     #definition of behavior
        chase_ghost = ChaseGhost()
        class AvoidGhost(py_trees.behaviour.Behaviour):     #definition of behavior
        avoid_ghost = AvoidGhost()

        sequence2.add_children([ghost_scared, chase_ghost])
        selector2.add_children([sequence2, avoid_ghost])
        sequence1.add_children([ghost_close, selector2])
        selector1.add_children([sequence1, eat_pills])

        behavior_tree = py_trees.trees.BehaviourTree(root=selector1)

        try:
            behaviour_tree.tick_tock(
                period_ms=500,
                num_iterations=5
                pre_tick_hanlder=None,
                post_tick_handler=None
            )
    =}
}
```

Listing 4.1. Declaration of a Behavior Tree with the libary `py_trees`

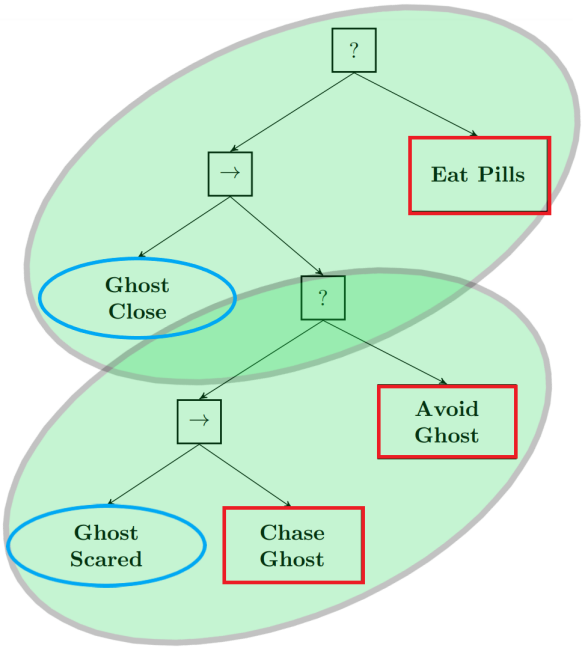


Figure 4.1. Pacman Behavior Tree divided in three sections [CÖ17]

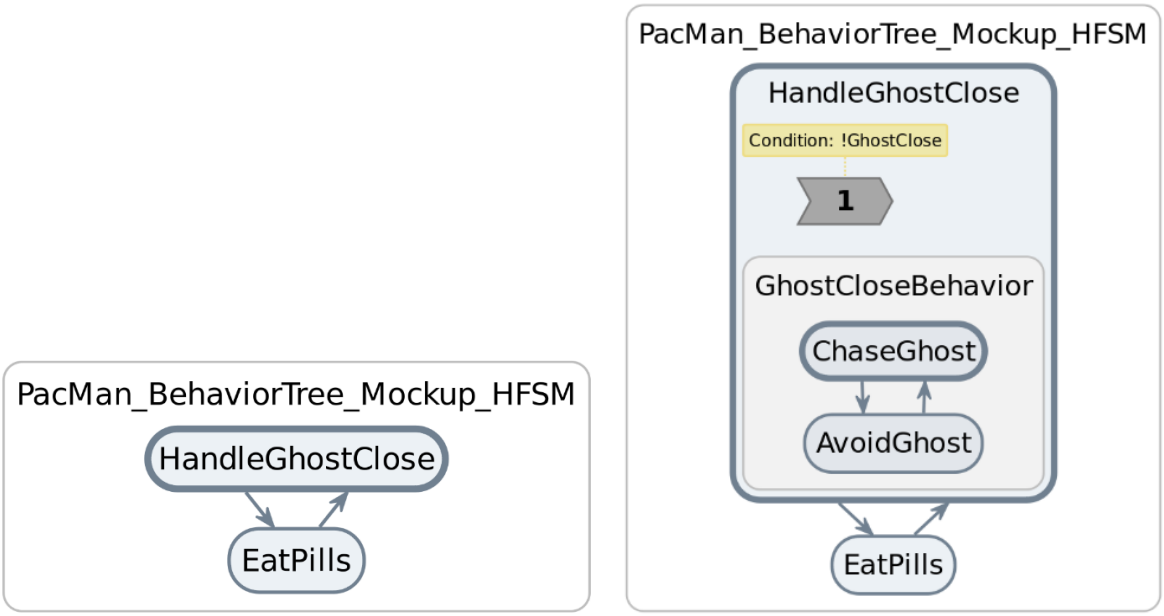
by Lingua Franca are not fully utilized. Lingua Franca is a coordination language that focuses on concurrent execution. As explained in Chapter 2, reactions are executed in parallel when the data flow dependency is at the same level. That is not the case here, as one reaction contains the Behavior Tree code. Another disadvantage is that that Behavior Trees defined with libraries cannot be visualized in the Lingua Franca diagram view. The Behavior Tree is defined only within the target language code, which then must be parsed to enable visualization. However, parsing user-input code must be avoided at all costs, as it could lead to unpredictable and error-prone results.

4.1.2 Select Mode Pattern

An alternative approach to introducing Behavior Trees in Lingua Franca would be a "Select Mode Pattern", which was a considered solution for the implementing Behavior Trees in Lingua Franca. This method makes use of Lingua Franca's built-in *Mode* concept, which was covered in Chapter 2. These modes best represent the state-based behavior for Behavior Trees in Lingua Franca. To explain the pattern, we again use the Pacman example from Chapter 1. The Behavior Tree can be divided into three sections, as Figure 4.1 illustrates.

The red-bordered nodes represent the three possible modes between which switching should occur. The node to be executed is decided by evaluating two condition nodes highlighted in blue. The green circles indicate the subtrees to which the Select Mode Pattern can be applied. When applying the Select Mode Pattern, a condition is evaluated to determine which of two nodes to select for execution. For example, the lower Fallback node is executed if the condition node `Ghost Close` returns success. However, if it returns failure, the action node `Eat Pills` is executed. Figure 4.2 shows the application of the Select Mode Pattern to the Pacman example.

One advantage of this approach would be that Behavior Trees could be displayed using the Lingua



(a) HandleGhostClose collapsed **(b)** HandleGhostClose expanded

Figure 4.2. Application of Select Mode Pattern on Pacman example.

Franca diagram view. Another benefit is that this implementation would resemble stateful execution, which is inherent to Behavior Trees. Memory nodes, explained in Chapter 2, would be natively implemented through the mode of reactors. Unfortunately, this approach is not suitable for general Behavior Trees. This is because the pattern only works for this type of tree, i.e., when the first child is a control flow node with a condition node as its first child. However, if the root node only had one child, then the Select Mode Pattern would not be applicable for this tree. A much more straightforward and more intuitive transformation approach is the following.

4.1.3 FSM Construction

The following approach is inspired by the FSM construction method by Colledanchise and Ogren [CÖ17]. In this method, the authors present how to design an FSM from a Behavior Tree that behaves like the Behavior Tree. In the FSM, the Behavior Tree root node has a structure seen in Figure 4.3.

As explained in Chapter 1, a Behavior Tree node is executed when it is ticked and returns either success, failure, or running. The node can be represented as a state in the FSM, and a transition for each return value represents the return values of the Behavior Tree node. The act of ticking nodes can be represented by a tick source, to which the return values are subsequently fed back. If the tick source receives a return value, it could lead the execution of the state back to the node. We obtain a FSM where a state corresponds to an execution or a composite node, three transitions which resemble the return values of a Behavior Tree node and the in-coming transition of the state the ticking of the Behavior Tree node. This basic structure can be applied to all Behavior Tree nodes, including both execution nodes and control flow nodes. If the root is a composite node, the pattern must also be applied to the nested nodes so that the FSM mimics the behavior of the Behavior Tree.

To illustrate this methodology, let us consider an example. Suppose we have the Behavior Tree

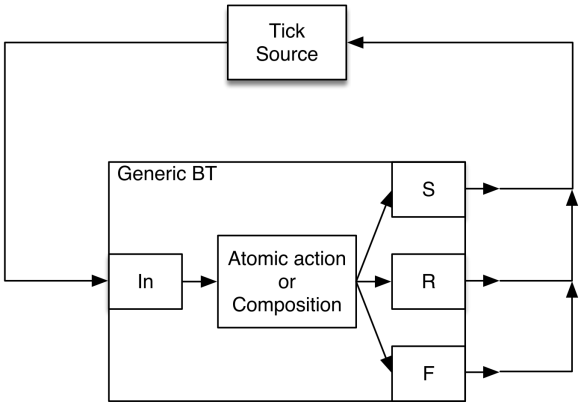


Figure 4.3. Structure of a Behavior Tree root node in a FSM [CÖ17]

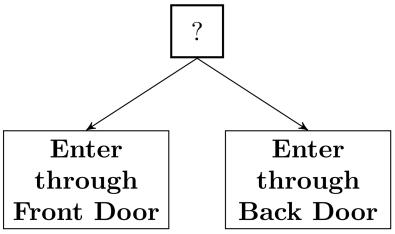


Figure 4.4. Behavior Tree example with a Fallback node as root [CÖ17]

shown in Figure 4.4.

This Behavior Tree has a Fallback as the root, which has two Task nodes as children. The FSM Construction shown in Figure 4.3 can be applied to the execution nodes. The Fallback node must manage the transitions as the parent node of the two task nodes. As explained in Chapter 1, a Fallback node executes its children from left to right until one of the children returns success. Such a transformation, as shown in Figure 4.5, would capture this functionality.

The Fallback node returns success or running if Use Front Door or Use Back Door returns success or running, respectively. Use Back Door is only executed if the failure transition of Use Front Door leads to this state. General rules for the FSM construction of Fallback nodes are now explained. The first child's state is entered when transitioning to the Fallback state through the in-transition. For this

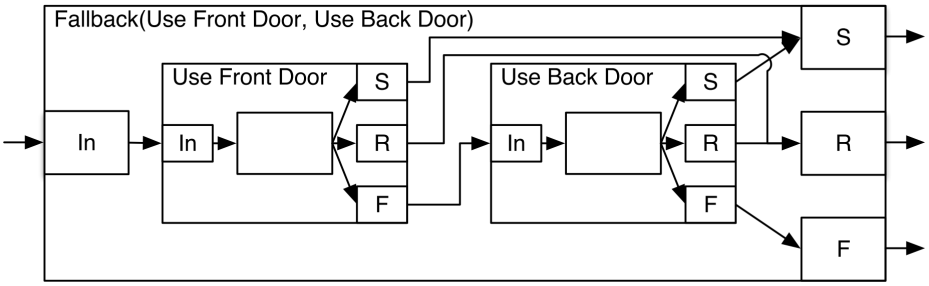
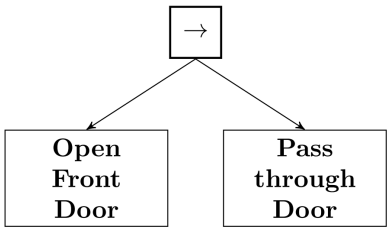
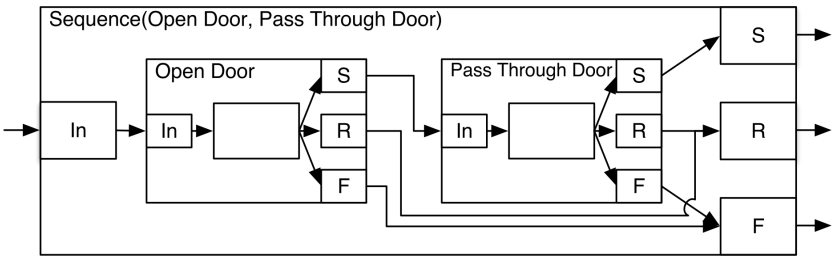


Figure 4.5. A FSM emulating the Fallback node [CÖ17]



(a) Behavior Tree example with a Sequence node as root [CÖ17]



(b) A FSM emulating the Sequence node. [CÖ17]

Figure 4.6. Transformation of a Behavior Tree with a Sequence node as root node into an FSM

state and all other states within the Fallback state, the successful transition of the parent Fallback node is taken upon a successful return value of the child state. However, a distinction must be made between the child nodes for failure return values. The rule here is that upon a child's failure, the state transitions to the next child's state, except for the first and last child states. The first state is entered upon an in-transition of the Fallback node. For the last child state, no further child state follows. Therefore, the failure transition of the Fallback node is taken upon a failure of the last child state. These rules implement a Fallback node into an FSM. For a Sequence, a similar pattern is showcased by Colledanchise and Ogren. Only the success and failure transitions of the child nodes need to be swapped. Figure 4.6 shows a transformation of a Sequence into an FSM. By combining the presented transformation patterns, more complex Behavior Trees can now be translated into an FSM. Figure 4.14 shows how a Behavior Tree consisting of Fallback and Sequence nodes can be converted to an FSM Upon closer examination of this approach, it can be noted that this transformation is not really a FSM construction since the nodes are not states but rather actors that fire when they receive input. The edges are not transitions but rather data token flows. That is highly suitable for Lingua Franca, which will be explained in more detail in the next section 4.1.4.

This approach has several advantages. Firstly, it is a modular solution since each Behavior Tree node is transformed into its own FSM. This ensures high code reusability, as each FSM can be used in other FSMs and recombined in other ways. Another advantage is that this type of transformation is supported by the Lingua Franca diagram view. Finally, this methodology offers an advantage by enabling concurrent execution which is natively supported in Lingua Franca. Further details are stated in the next section 4.1.4.

4.1.4 Conclusion

After evaluating the three presented approaches, the FSM Construction will be implemented as the solution. That is because it is the only solution that meets all three requirements adequately. The FSM Construction pattern offers high modularity, is supported by Lingua Franca's diagram view, and can utilize Lingua Franca's built-in concurrent execution. As Lingua Franca is an event-driven language based on data token flow, the pattern can be applied to reactors in Lingua Franca. Each node represents its reactor, which has an input for starting execution and an output for each return value. By transforming each of the nodes to own reactors high modulariy is achieved. The data token flow can be mapped using Lingua Franca connections, as explained in Chapter 2. Since one reactor can initialize another reactor in Lingua Franca, composite Behavior Tree nodes could also be represented by reactors. For example, a Sequence reactor could initialize instances of its child reactors and thus become the parent node. Another advantage of this approach is that, during the transformation, the Behavior Tree can read inputs and write outputs, as it involves reactors. The detailed specifics are explained in Section 4.3. This would also allow the implementation of local variables by restricting certain reactors to have input and output ports for specific variables only. Furthermore, Lingua Franca's built-in concurrent execution can be utilized by creating a separate reactor for each execution node, where the node-specific code is executed in a separate reaction. Further details regarding concurrent execution are provided in Chapter 2.

4.2 DSL and Transformation of Behavior Trees

This section presents the designed DSL and explains the corresponding transformation pattern. Capitalized words represent built-in Lingua Franca data components. An observation is that the running output is not needed in our case because the output of a node is running, if a node does not have a return value in a tick. Leaving out the wire for running saves a lot of computations and connections, and the graphical view would look less cluttered. Therefore the running return value is neglected in the transformation. For each component, a transformation pattern is presented, which demonstrates how that component can be translated into Lingua Franca semantics. The transformation involves utilizing the FSM Construction pattern in conjunction with the Lingua Franca components. Each constructed FSM represents a distinct reactor, with transitions depicted through ports and connections.

The transformation is performed in a depth-first search style, starting from the root node. The Listing 4.2 outlines the transformation traversal. If the root node is a composite node, the transformation of the root node is paused, and the transformation of the first child is started. After transforming an execution node, backtracking is performed. The objective pursued during the design of the rules was to make the definition of the components as intuitive as possible while achieving the highest level of expressiveness.

4.2.1 Behavior Trees

DSL

First, a rule for defining Behavior Trees is given, which can be seen in Listing 4.3. A Behavior Tree consists of a name, IO that can be declared in any order, and a `bt_node`, which serves as the root node of the Behavior Tree and can be either a composite or an execution node.

```
transformBT(tree):
    transformNode(tree.root)

transformNode(node):
    # for each implemented node type
    if node instanceOf Sequence : transformSequence(node)

transformSequence(sequence):
    # composite nodes first transform their children nodes
    for child in sequence.children:
        transformNode(child)
```

Listing 4.2. Pseudocode of the transformation process

```
behavior_tree : "behaviortree" NAME_STRING "{"
                    inputs_outputs
                    bt_node
                "}"

inputs_outputs : INPUT ("," inputs_outputs |  ϵ)
               | OUTPUT ("," inputs_outputs |  ϵ)

bt_node : Fallback | Sequence | Parallel | Task | Condition
```

Listing 4.3. Rules for a Behavior Tree

Transformation

In Figure 4.7 an example transformation for a Behavior Tree can be seen. The transformation of a `behavior_tree` component into a reactor can be achieved using the following pattern. First, a reactor is created with the name `NAME_STRING`, an input start, and an output for success and failure, respectively. Next, the IO from the `inputs_outputs` component are declared to the created reactor. Then, the corresponding ports of the Behavior Tree reactor are connected to the root node reactor. Inputs are directed from the Behavior Tree reactor to the root node reactor, and outputs are directed vice versa. How these IO are handled within the Behavior Trees will be explained in more detail in Section 4.3. The Behavior Tree reactor must then instantiate a reactor instance of the `bt_node`. This instance represents the root node of the Behavior Tree. How Behavior Tree nodes are declared and transformed will be explained in the following.

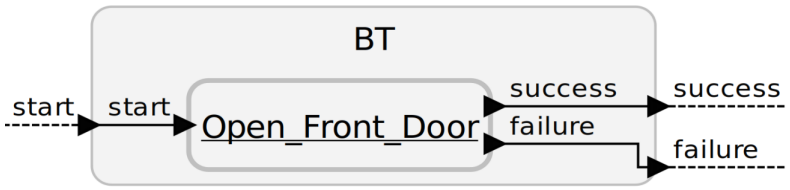


Figure 4.7. Example transformation for a Behavior Tree with a Task node.

```
sequence : "sequence {" bt_nodes "}"

fallback : "fallback {" bt_nodes "}"

parallel : "parallel" INT "{" bt_nodes "}"

bt_nodes: bt_node ("," bt_nodes |  ϵ)
```

Listing 4.4. Rules for composite nodes

4.2.2 Composite Nodes

In this section, the DSL and transformation pattern for composite nodes are presented. Composite nodes are the control flow nodes Fallback, Sequence, and Parallel.

DSL

The rules for these nodes are visible in Listing 4.4. Composite nodes contain a non-empty set of `bt_nodes` representing the child nodes. In addition, the Parallel component includes an integer value that represents the value M, as explained in 2.

Transformation

Like all other nodes, composite nodes are transformed into reactors. During the transformation, a reactor for the composite node is first created, which adds an input start and an output for success and failure, respectively. The composite node reactor instantiates the reactors of the child nodes. Next, the connections between the child nodes must be established. In Section 4.1, the rules for Sequence and Fallback for FSM Construction were presented. These constructions are applicable at reactor level, as Figure 4.8 shows. For Fallback nodes, a connection from the success port of each child reactor to the success port of the Fallback must be established. In Lingua Franca, however, outputs cannot have multiple sources, as this cannot guarantee determinacy, as explained in Chapter 2. In this specific case, however, this would be desirable to implement the Fallback semantics. It can be noted, that the Fallback rules make it impossible for multiple writers to write to the success output port of the Fallback node. That follows from the fact that as soon as one child node writes to the Fallback's success port, the execution of the Fallback node will be exited. Therefore, a reaction can be created to implement the desired behavior. There can always be precisely one reactor for a tick that triggers the reaction. Thereby, determinacy is assured. The concept of reactions is explained in Chapter 2. The introduced reaction, depicted by a chevron shape in Figure 4.8, receives all success ports of the child reactors as a trigger and writes to the Fallback's success output as soon as it is triggered. This constructions maps the Fallback rules.

To implement the desired behavior, connections must also be established between the child reactors such that the failure port of a child reactor is connected to the start port of the next child reactor. The first and last child reactors are exceptions to this rule. The first child has a connection from the start port of the Fallback to the start port of the child. For the last child, a connection from the failure port of the child reactor to the failure port of the Fallback reactor is established. The same transformation pattern can be used for Sequence nodes, but the failure and success ports must be exchanged.

The Parallel node is not covered in the paper by Colledanchise and Ogren on which the transformation idea is based [CÖ17]. The transformation of Parallel nodes and Fallback nodes differs only in

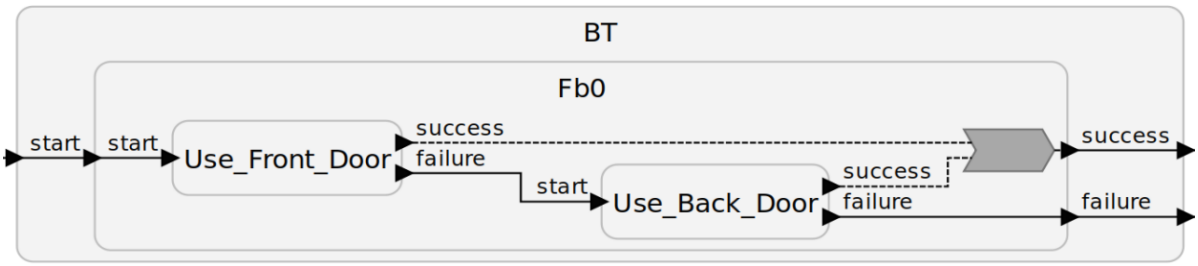


Figure 4.8. Example transformation for a Fallback node

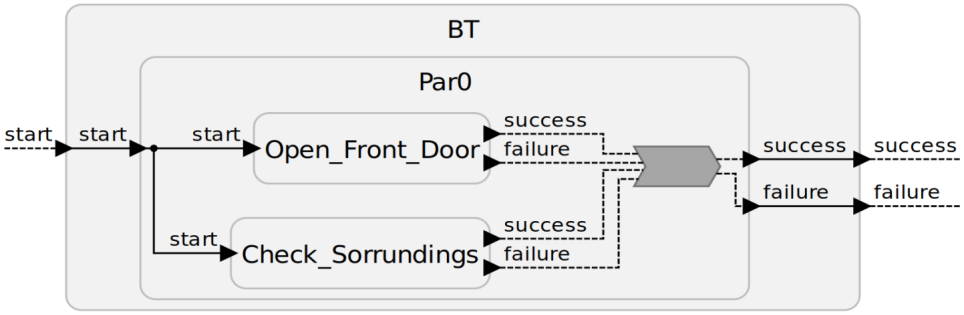


Figure 4.9. Example transformation for a Parallel node

the connections of their child reactors. While the Fallback node has a sequential execution, where the execution flow is passed from child to child, all child reactors are started simultaneously in the Parallel node. The Figure 4.9 shows an example for a transformation of a Parallel node. The transformation of the Parallel node is achieved by connecting the input start of the parallel reactor to all start ports of each child reactor. The Parallel node is defined to return success when $\geqslant$ M children return success and failure when > N-M children return failure. N here represents the number of children of the Parallel node. This behavior can be mapped by a Reaction that holds all children's success and failure outputs as triggers and success and failure ports of the Parallel node as effects. This Reaction must now check how many success or failure triggers are present and write to the corresponding output of the Parallel node according to the count. Next, the transformation pattern for the execution nodes will be presented.

4.2.3 Execution Nodes

Now, the execution nodes will be added, which are the leaf nodes.

DSL

A possible rule for the execution nodes, Task and Condition, is shown in Listing 4.5. Task and Condition nodes can have a name and declare sources and effects separated by an arrow ->. The target language code to be executed by the Task or Condition is written in the `{= ... =}` body.

Transformation

An example for a execution node transformation can be seen in Figure 4.10.

```
task : "task" (NAME_STRING | ϵ)
            (sources | ϵ) ("->" effects | ϵ)
            "{="   TARGET_LANG_CODE   "=}"

condition : "condition" (NAME_STRING | ϵ)
                (sources | ϵ) ("->" effects | ϵ)
                "{="   TARGET_LANG_CODE   "=}"

sources : SOURCE ("," sources | ϵ)

effects : EFFECT ("," effects | ϵ)
```

Listing 4.5. Rules for Task and Condition

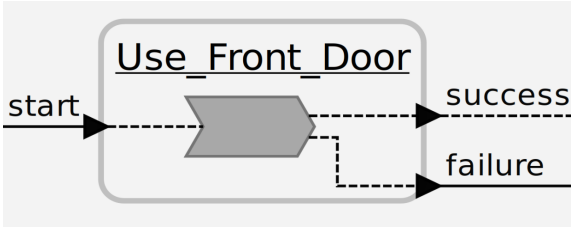


Figure 4.10. Example transformation for a Task node without additional sources and effects

Execution nodes can be transformed using the following pattern. A reactor is created with an start port and an input port for each declared source. This reactor receives an output port for success and failure, and for one for each declared effect. Within this reactor, a reaction is created, and the given target language code is copied into the code section of the reaction. This reaction is triggered by the start port of the Task reactor. The reaction receives the declared sources, and the two ports success and failure with the declared effects. How IO are handled in the Behavior Tree is explained in the following section 4.3.

4.3 Inputs and Outputs

Behavior Trees can declare IO as shown in Listing 4.3. The following section motivates why Behavior Trees should have IO. The subsequent sections explain how these are implemented and how they are used.

4.3.1 Motivation

Behavior Trees without IO are inflexible. They cannot be reused because all the data required to calculate the behavior of the Pacman is defined and computed within the tree. That limits the tree's flexibility, as it can only be used for a single agent. By adding IO to Behavior Trees, the trees become flexible and can be used by multiple agents. Agents could send their data and data from their environment to the Behavior Tree and receive an output based on that data. Defining Behavior Trees with IO also brings additional benefits. It is much easier to debug a Behavior Tree when the data

computation is done outside the tree since the focus can be on the IO of the Behavior Tree during testing.

To further motivate Behavior Trees with IO in Lingua Franca, a IO Behavior Tree interface is introduced for the Pacman example shown in Figure 1.1. This Behavior Tree could have inputs for the positions of the Pacman and ghosts, and a Boolean indicating whether the ghosts are scared. The output of this Behavior Tree could be the position of the Pacman for the next tick. IO interfaces like these are particularly beneficial in Lingua Franca because it offers many advantages over other IO Behavior Tree implementations in other languages. Clear Lingua Franca IO interfaces facilitate reuse, as a standardized interface is provided for every other component. Furthermore, dataflow implementations like those in Behavior Trees with IO often encounter determinacy problems in other languages. Defining Behavior Trees with IO in Lingua Franca would be highly beneficial, as Lingua Franca focuses on deterministic behavior even with dataflow. This determinacy leads to less error-prone behavior. For this purpose, execution nodes are able to read inputs from the Behavior Trees and write outputs. This section explains how the transformation for the Behavior Tree nodes must be adapted to enable this functionality.

4.3.2 Inputs

Syntax

When declaring a Behavior Tree with the DSL shown in Listing 4.3, it was possible to specify inputs, which were then connected with a connection from the Behavior Tree reactor to the root node reactor during transformation. Furthermore, execution nodes can specify sources, as shown in Listing 4.5, for the reaction that executes the target language code. That allows accessing the inputs of the Behavior Trees in the target language code.

Transformation

When declaring a Behavior Tree, it was possible to specify inputs that were connected to the root node reactor during the transformation. That means if the root node is an execution node, the necessary connections are established so the execution node can access the inputs fed into the Behavior Tree reactor. Therefore, the presented transformation only needs to be adapted for the composite nodes so that the inputs can reach the execution node reactors.

Figure 4.15 shows a visualization of the transformation of a Behavior Tree with inputs. The nodes `Attack` and `Hide` want to read the input `enemy_pos` and the `agent_pos` input of the Behavior Tree. The node `CallBackup` only wants to read the `agent_pos` input. The corresponding reactors should only create a port if needed. These port scopes must be considered during the transformation process. For transforming composite nodes into reactors, it must first be checked whether children exist that have an input for a Behavior Tree input. During the transformation of the composite node, for each input of the child that is not the start port, a corresponding input port with the same input name is created in the reactor. It should be noted that the inputs are grouped by name. If multiple children want to read the same input, only one input port with the same name is created for the parent composite node. That is not a problem because connecting one input port to multiple other input ports of children reactors is supported. Finally, the input of the composite node must be connected to each child node. This methodology ensures that each execution node can read the desired input from the Behavior Tree since the input is forwarded to the execution nodes by a connection starting from the Behavior Tree reactor, going through all composite node reactors, and ending at the execution node reactor, which is the leaf node. Furthermore, a reactor only creates an input port if needed.

4.3.3 Outputs

Syntax

Upon declaring a Behavior Tree with the DSL shown in Listing 4.3, it was possible to specify outputs. Each output of the root node was connected to the Behavior Tree reactor during transformation. Moreover, execution nodes can declare effects, as shown in Listing 4.5. These effects were added to the reaction holding the target language code for the execution node. That allows the code to write to outputs of the corresponding reactor.

Transformation

Just as with inputs, only the transformation of the composite nodes needs to be adjusted. Composite nodes must first create ports for each output that a child reactor wants to output to. As with inputs, the output ports are grouped by their name. However, it is impossible to connect all of the child's output ports with the same name to the corresponding output port of the composite node, as in Lingua Franca, an output node cannot have multiple sources. That is because determinacy cannot be guaranteed if, for example, multiple writers want to write a value to an output in the same tick. However, it could offer high functionality for our specific case if multiple execution nodes could write to the same output. The Behavior Tree shown in Figure 4.15 could be used to output the agent's position for the next tick. In this case, the Behavior Tree declares an output for the position. It would make sense if both Task nodes, Attack and Hide could output the agent's following position, as both Tasks are used to control the agent's position. This Behavior Tree only consists of Fallback and Sequence nodes as the composite nodes. Fallback and Sequence nodes execute their children sequentially. In these cases, it would make sense if multiple nodes were connected to the same output and could write to it, and the output value of the last executed node was output by the Behavior Tree. Implementing this approach is sensible and possible since it does not violate the Lingua Franca principle of determinacy.

The Figure 4.16 shows the transformation of the Behavior Tree with the mentioned outputs. To implement this behavior, as mentioned, only the transformation of the composite nodes needs to be adjusted. Here, as with inputs, it must be checked whether a child reactor wants to write to an output of the Behavior Trees. When transforming the composite node, for each output of a child that is not the success or failure port, an output port with the same output name is created for the corresponding reactor. These output ports are grouped by name, too. A reaction for the composite node must be created for each output port to achieve the desired behavior. This reaction receives all output ports of the children with the same name as its trigger and the corresponding output port of the composite node as its only effect. Now, for each of these reactions, the code must be set to check which triggers were present and to write the value of the last present trigger to the output of the composite node. In Parallel nodes, there is no sequential execution of children reactors. This prioritization can also be implemented for these nodes. It could prioritize the output value according to the order of declaration of the parallel child nodes.

4.4 Local variables

4.4.1 Motivation

Using the previously presented concepts, it is possible to receive messages from outside the Behavior Tree reactor and to write to the outputs of the Behavior Tree reactor. However, with the current transformation, Behavior Tree nodes cannot communicate with each other.

```
sequence : "sequence {"
                (locals | ϵ)   % new
                bt_nodes "}"

fallback : "fallback {"
                (locals | ϵ)   % new
                bt_nodes "}"

parallel : "parallel" INT "{"
                (locals | ϵ)   % new
                bt_nodes "}"

bt_nodes : bt_node ("," bt_nodes |  ϵ)

locals : local ("," local | ϵ)   % new

local : "local" LOC_NAME_STRING    % new
```

Listing 4.6. Adapted rules for composite nodes

This feature is, however, very advantageous, as can be seen in the following example. Suppose the agent defined by the Behavior Tree in Figure 4.15 would like to indicate the number of enemies it has seen while attacking or hiding when executing CallBackup. For this purpose, a local variable could be used within the scope of the Sequence. The number of sighted enemies is computed within the execution of the Attack or Hide node, and the CallBackup node could access this number. Blackboards, which were explained in more detail in Chapter 2, often provide the functionality of local communication in other Behavior Tree implementations. One issue with such blackboards is that they are tricky to manage and debug. It is challenging to determine which node wrote which entry for a local variable and who read it at what time. Another significant disadvantage of blackboard implementations is that they may be error-prone with inconsistent entries. For example, this could happen when two components simultaneously change the value of a local variable that another node wants to read. Lingua Franca could provide a solution for that problem since it is a dataflow language that ensures deterministic execution. The benefits of explicit data transmission and deterministic execution can be utilized for implementing local variables.

4.4.2 Syntax

Blackboard implementations are not chosen for enabling reactors of the Behavior Tree nodes to communicate with each other but a different approach. In our approach, local variables are managed through composite nodes.

To enable the declaration of local variables, the DSL of composite nodes is modified as shown in Listing 4.6. Composite nodes can define local variables. Local variables are in the scope of every Behavior Tree node reactor that is a descendant of the declaring composite node's reactor. If an execution node wants to process a local variable, it declares the variable in the source section if the node wants to read it. If the node wants to write to it, it declares the variable as an effect. The source and effect sections of execution nodes, which can be seen in the Listing 4.5, are used for IO, and local variables. The user do not have to make a differentiation which makes working with variables intuitive

and easy.

4.4.3 Transformation

Foundations

Local variables are treated like regular Behavior Tree IO, whose properties were explained in section 4.3, within their scope. That means, for example, that connections for inputs between a node and its child nodes with the same port name are established during the transformation, and composite nodes pass on the latest output of their child nodes of a local variable.

For Transformation, it is important to differentiate whether a node is a writer, a node that writes to the value of a local variable, or a reader, a node that reads the value of a local variable. The user makes the differentiation by declaring a local variable in the sources or effects section. If an execution node is a writer to local variable, a connection is established from this reactor to every afterward executed reactor reader to the variable. Once another execution node writes to the value of the local variable, all afterward executed nodes receive the value of the last executed node for the variable. Therefore, the update principle explained in the outputs also applies here.

Parallel nodes are special when it comes to local variables, as multiple execution nodes could write to the value of a local variable simultaneously. The Lingua Franca compiler rejects such programs since they are non-deterministic. Furthermore, if there were a writing and a reading child node to the same local variable, these two could not be executed in Parallel, as the reading child must wait for the value of the local variable. So the user must be aware of these cases when working with local variables in the scope of Parallel nodes.

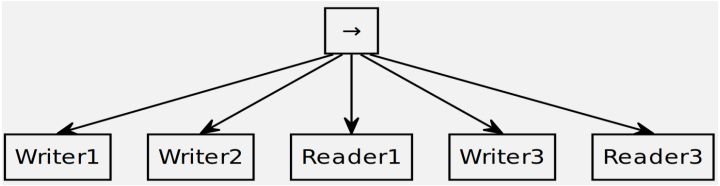
A case distinction can be made for transforming Behavior Trees with local variables.

Local communication between nodes with the same parent node.

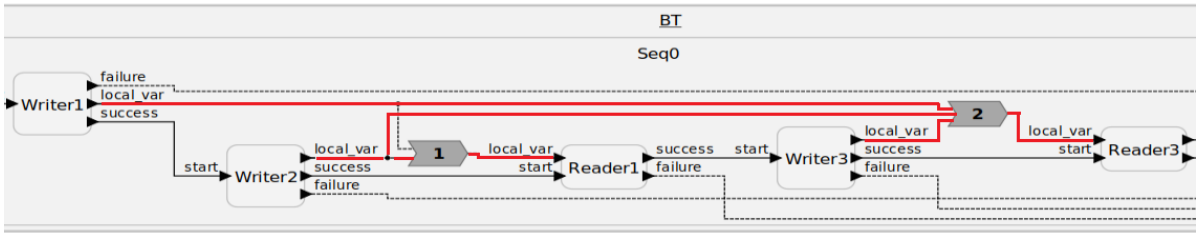
If nodes process the same local variable and have the same parent node, the transformation of the parent node must be carried out as follows. During the transformation of the parent node, this node identifies which children nodes are writers and which are readers to which local variables. Figure 4.11 shows an example Behavior Tree with local communication. The following transformation pattern is applied for each local variable in the scope. A new reaction is created for the reactor for each chronological switch between a writer and a reader. The first of those reactions receives the local variable outputs of each writer execution node executed before the first reader as its trigger. As its effects, this reaction receives all reader nodes executed before the second writer. The reaction code should implement passing the value of the last present trigger to the reactors of the effects. The second reaction receives the outputs of all writers, which are executed before the next reader as its triggers in addition to the triggers of the previous reaction. As its effects, this reaction receives all nodes' inputs executed before the next writer. The reaction code must also be defined so that the last trigger value is given to the effect's associated reactors. This process must be carried out for each of these reactions. This structure ensures that every reader receives the most up-to-date value for the local variable written by a previously executed node. It is not necessary to create a port for the parent node as long as the communication takes place only between descendant nodes of the parent node.

Local communication between nodes with the different parent nodes.

Execution nodes that do not have the same parent node must be treated differently, as in this case, at least one composite node must open a port so that the local variable's value can be passed through.

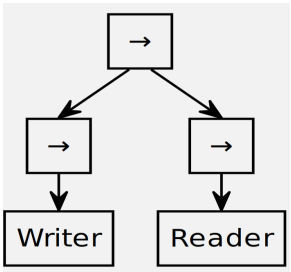


(a) Behavior Tree with local communication.

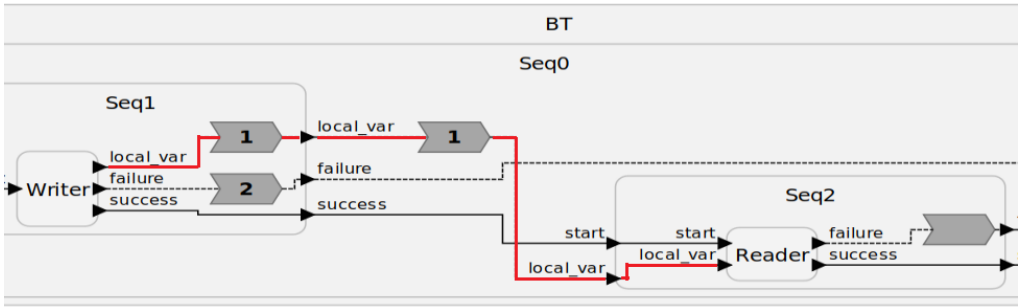


(b) Transformation of the Behavior Tree.

Figure 4.11. Transformation of a Sequence node with local communication support.



(a) Behavior Tree with local communication.



(b) Transformation of the Behavior Tree.

Figure 4.12. Transformation of a Behavior Tree with local communcation between nodes with different parent nodes

So far, it was not necessary to analyze the tree's overall structure. That means each node is visited exactly once during its transformation, as shown in Listing 4.2. However, the tree must be evaluated once beforehand for local communication between nodes with different parent nodes. The necessity of this structural analysis can be explained in Figure 4.12. When transforming `Seq1`, it is impossible to know whether another node outside that Sequence is a reader of that local variable. Thus it is

```
For localVar in BehaviorTree
    Determine last-to-execute reader node "lastReader" of localVar
    for node in localVar.localOuts
        while node.parent != lastReader.parent
            add output port for local var to parent
            node = node.parent

    Determine first-to-execute writer node of local variable "firstWriter"
    for node in localVar.localIns
        while node.parent != firstWriter.parent
            add input port for local var
            node = node.parent
```

Listing 4.7. Algorithm for adding the necessary ports to each node.

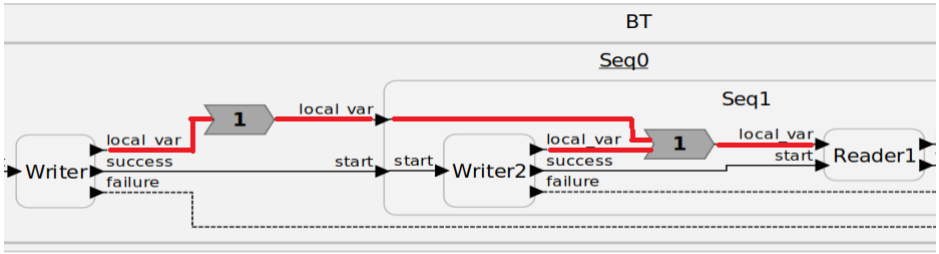


Figure 4.13. Transformation showing an edge case in the context of local communcation.

unknown whether the Sequence has to create a port for passing the value of the local variable. That means that before the beginning of the depth-first transformation, it must be checked which nodes have to create a port for which local variable. The following can be done to evaluate to structure of the local variables: First, two lists are created for each local variable in the tree. The first list, `localOuts`, contains all execution nodes that are writers of that local output. The second list, `localIns`, contains all reader execution nodes of that local variable. Now, the algorithm shown in Listing 4.7, ensures that every writer node can send a value for the local variable to every afterward executed reader node. The transformation shown in Figure 4.12 can be used to explain this algorithm. In this example, a writer and a reader for the local variable `local_var` are shown. Lines 1-6 ensure that `Seq1` creates the output for the local variable. Lines 7-13 ensure that `Seq2` creates an input for this variable. As `Seq1` has an output for the local variable and `Seq2` has an input, the rules for local communication with the same parent are applied during the transformation of `Seq0`, allowing data transfer between these two nodes. Combining these two presented transformation rules enables the implementation of local variables.

Edge case

For this transformation to work, an edge case must be considered the transformation does not cover. This edge case occurs, for example, in the tree shown in Figure 4.13. If a composite node has an input for a local variable and a child node of the composite node also wants to write to this variable, then a additional reaction has to be created. For this reaction, the input of the composite node must be set as a trigger in addition to all writers before the first reader. The input trigger must be defined as a trigger for all upcoming created reactions which handle the local communication. That enforces the rule that

the last output value of the local variable is read. This rule is essential, for example, when a Sequence has a writer and a reader as child nodes in that order. Suppose the writer has an output port for a local variable that the reader wants to read but does not write to that output. In that case, the reader will receive the last written value of the local variable provided by the Sequences' input. IF

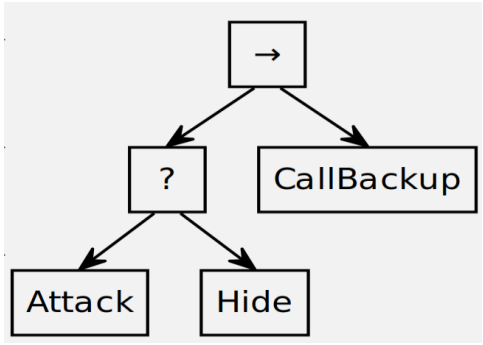
Conclusion

These transformation changes establish the functionality of local communication, which offers many advantages. Firstly, it is visually more understandable, as the origin of a value can be better traced in the diagram through explicit connections. Furthermore, this type of implementation also makes it possible to define scopes for variables intuitively.

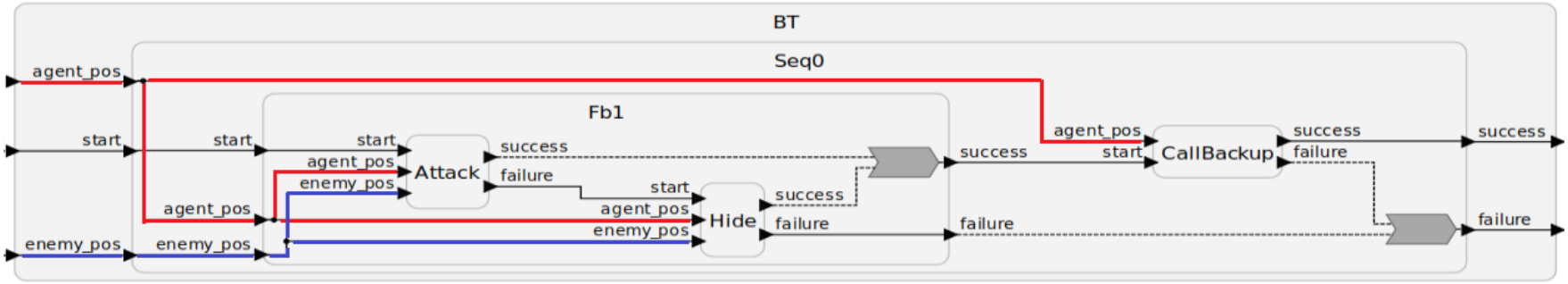
(a) Behavior Tree with a Fallback and two Sequence nodes [CÖ17]

(b) An FSM emulating the Behavior Tree depicted in 4.14a [CÖ17]

Figure 4.14. FSM Construction for a complex Behavior Tree



(a) Behavior Tree, wherein nodes want to read inputs



(b) Transformation of the Behavior Tree depicted in Figure 4.15a. Red lines denote dataflow of input agent_pos. Blue lines denote dataflow of input enemy_pos.

Figure 4.15. Transformation of a Behavior Tree with inputs

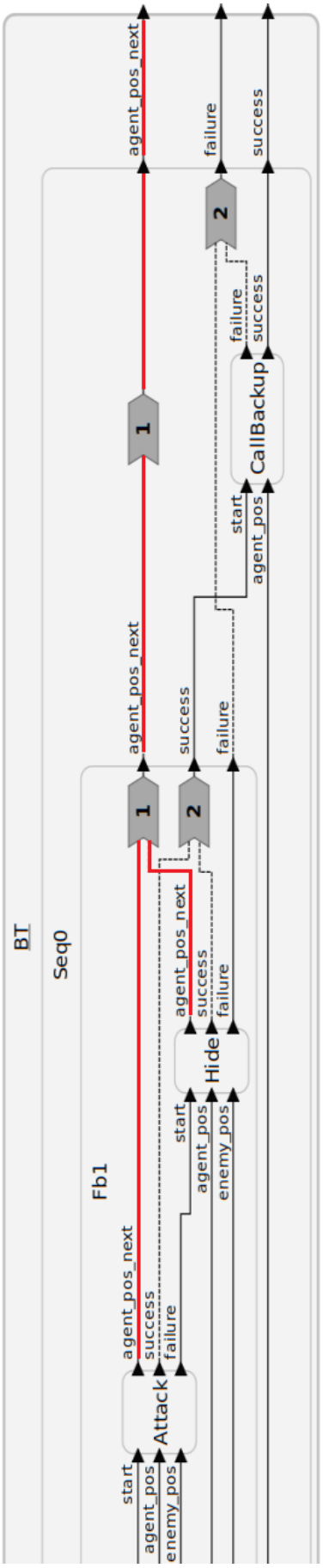


Figure 4.16. Example transformation for a Behavior Tree with outputs being highlighted red

Chapter 5

Implementation

This chapter presents the implementation of the DSL and the transformation for Behavior Trees in Lingua Franca explained in Chapter 4.

5.1 Used Technologies

5.1.1 Eclipse and Xtext

Eclipse[1] is a free and open-source IDE origally created for Java development. Today, Eclipse supports numerous programming languages and is highly extensible through many plugins. Eclipse, in conjunction with the Xtext[2] framework, was used to develop, design, and deploy the Lingua Franca language. Xtext is an Eclipse framework for developing DSLs, which builds the language infrastructure based on the defined grammar.

5.1.2 Epoch

Epoch[3] IDE is a software program based on Eclipse used for developing Lingua Franca programs. Epoch IDE supports an editor with highlighting for Lingua Franca programs, an compiler and a diagram synthesis. Epoch has been used for testing and debugging the presented DSL and transformation.

5.2 A DSL for Lingua Franca

The DSL for Behavior Trees in Lingua Franca has been implemented with the Xtext framework. Xtext syntax is comparable with Extended Backus-Naur Form[4]. The Extended Backus-Naur Form is not more expressive than the conventional Backus-Naur Form; instead, it simplifies the process of defining the grammar of a language through the use of syntactic sugar[5]. We will start by defining the rule for Behavior Trees. The Listing 5.1 shows the presented rules in Section 4.2 in the Lingua Franca grammar file. One peculiarity of this implementation is that it includes using so-called `VarRefs` for sources and effects. VarRefs are data types that reference variables within the allowed scope. Another notable property is that the Condition node does not have to be defined separately. The Condition and Tasks behave the same way since the output running is missing, as previously mentioned.

[1] https://eclipseide.org/
[2] https://www.eclipse.org/Xtext/
[3] https://www.lf-lang.org/docs/handbook/epoch-ide
[4] https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html
[5] https://www.informatik.uni-kiel.de/ mh/lehre/pps20/vorlesung/kap21.pdf

```
// Behavior Tree
BehaviorTree:
    {BehaviorTree}
    'behaviortree' (name=ID) '{'
        ((inputs+=Input) | (outputs+=Output))*
        rootNode=BehaviorTreeNode
    '}'

BehaviorTreeNode:
    Sequence | Fallback | Parallel | Task

// Composite nodes
Sequence:
    {Sequence}
    'sequence' '{'
        locals+=Local*
        nodes+=BehaviorTreeNode+
    '}'

Fallback:
    {Fallback}
    'fallback' '{'
       locals+=Local*
       nodes+=BehaviorTreeNode+
    '}'

Parallel:
    {Parallel}
    'parallel' (M=INT) '{'
        locals+=Local
        nodes+=BehaviorTreeNode+
    '}'

// Execution Nodes
Task:
    {Task}
    ('task' | condition?='condition')
    (taskName=STRING)?
    (taskSources+=VarRef (',' taskSources+=VarRef)*)?
    ('->' taskEffects+=VarRef (',' taskEffects+=VarRef)*)?

// Local communication
Local:
    {Local}
    'local' name=ID (':' type=Type)? ';'?
```

Listing 5.1. Implemented rules for defining Behavior Trees in Lingua Franca

```
private Reactor transformBTree(BehaviorTree bt, List<Reactor> newReactors) {
    var reactor = LFF.createReactor();
    newReactors.add(reactor);
    reactor.setName(bt.getName());
    setBTInterface(reactor);
    copyInOutputs(reactor, bt.getOutputs(), bt.getInputs());
    var localOuts = new HashMap<Local, List<String>>();
    var localIns = new HashMap<Local, List<String>>();
    computeLocalPaths(bt.getRootNode(), "0", localOuts, localIns);
    var nodesLocals = computeNodesLocals(bt.getRootNode, localOuts,
            localIns, new HashMap<BehaviorTreeNode, NodesLocalOutInputs);
    var nodeReactor = transformNode(bt.getRootNode(), newReactors, nodesLocals);
    // ...
}
```

Listing 5.2. Method transformBTree() for transforming a Behavior Tree `bt`

5.3 Transformation to the Lingua Franca Semantics

In the following, the implementation of the transformation is presented. This will particularly focus on the implementation of local communication.

5.3.1 Behavior Trees

When compiling a Lingua Franca program, a *model* is created that holds all the reactors of the program. The idea is to transform the Behavior Trees into reactors and pass these newly created reactors to the model. After this step, the Behavior Trees are irrelevant to the model. To achieve this, for each defined Behavior Tree `bt` of the Lingua Franca program, the method `Reactor transformBTree(BehaviorTree bt, List<Reactor> newReactors)` shown in Listing 5.2, is called. Each Behavior Tree is defined by the Behavior Tree rule depicted in Listing 5.1. That means that we have access to the user-defined components during the transformation. All Behavior Tree nodes transformed into reactors are added to the `newReactors` list so that they can also be added to the model.

A reactor is first created and given a name for each Behavior Tree node. Each Behavior Tree reactor is also given an input start and outputs success and failure. Next, the defined IO from the `bt` component are copied to that reactor. As described in Section 4.4, the structure of communication of the local variables is then analyzed in the transformation. The mentioned `localOuts` and `localIns` lists are `HashMaps` which map each local variable to a string list. The `computeLocalsPaths()` method fills the two HashMaps, with each local variable having a list of execution nodes that want to read or write to it. Each string in the list represents a path to an execution node. Paths are created by numbering the children from 0 that are visited during the traversal of the tree until an execution node is reached. The traversal of the tree is possible because composite nodes have a list in which their children are saved chronologically. A "-" symbol separates the numbering of the children. For example, the node `GhostScared` in Figure 1.1 has the path `0-0-1-0-0` and `EatPills` has the path `0-1`. The computeLocalsPaths() method fills these lists with values by adding the path as a value to the local variable key, if the Task reads or writes it. Then, the computeNodesLocals() method is called, which follows the algorithm shown in Listing 4.7. For the `localVar` list in the algorithm, a `HashSet` can be declared, which creates a set of the key lists. The nodes `lastReader` and `firstWriter` can be easily read from

```
var localCompleted = new ArrayList<TriggerAndEffectRefs>();
var localCommunication = new TreeMap<String, TriggerAndEffectRefs>();
for (var node : seq.getNodes()) {
        // ...
}
```

Listing 5.3. A

the localIns and localOuts lists, respectively, by using `firstWriter = localOuts.get(0)` and `lastReader = localIns.get(localIns.size() - 1)`. However, since IO ports cannot yet be added because the associated reactors have not yet been created, the `HashMap<BehaviorTreeNode, NodesLocalOutInputs> nodesLocals` is created, which saves for each Behavior Tree node the local variable IO it wants to use. At last, the root node of the Behavior Tree is transformed.

5.3.2 Implementation of local communication

To explain further details of the local communication, an example Sequence transformation is introduced. First, a reactor is created and the Behavior Tree ports, namely start, success and failure, are created. Then, the ports for the local variables are created, which are obtained from the `nodesLocals` list. Next, we come to the main for loop, which transforms the children.

In the following, I will explain the general structure of how the principle of local communcation within the same parent node presented in Section 4.2.2 can be implemented. In Listing 5.3, the fields necessary for the computation of local communication between child nodes with the same parent node are depicted, along the for-loop, which transforms the children sequentially. The field `localCompleted` holds a list of triggers and effects. The field `localCommunication` is used for processing local communications between child nodes. Using these fields, the transformation described in Section 4.2.2 can be applied. An example Sequence node is introduced for presenting the concept, which can be used for the implementation of local communication. Whenever a writer child node `w` of the Sequence writes to the output of a local variable `localV`, an entry `(localV, (trigger=w, effect=))` is created in the field localCommunication. Upcoming writer nodes will be added to the trigger list of the second component. When the Sequence node processes a reader child node `r`, the entry in localCommunication is changed to `(localV, (trigger=w, effect=r))`. The trigger list of the second component is now locked, which means that only readers, which are the effects, can be added. When the Sequence node transforms another writer child `w2` for the same local variable, the entry `(lV, (trigger=w, effect=r))` is added to the list in localCompleted, and a new entry `(localV, (trigger=w2, effect=))` is replacing the old entry in localCommunication. When all children are executed, all reactions in localCommunication are added to localCompleted. After that, a reaction with the specified triggers and effects is created for each entry in localCompleted. Subsequently, a method automatically writes the code for each of these reactions so that the latest value of the triggers is forwarded to all effects. This enforces the presented pattern of the chronological switch between writer and reader.

5.4 Renaming ports

All concepts presented in Chapter 4 can be successfully ported to Lingua Franca, with one exception, as shown in Listing 5.4. The concept for local communication involved a composite node creating an input port when a child node wants to read the value of a local variable and creating an output port

```
behaviortree BT {
    sequence {
        local comp:int
        task "Writer1" -> comp {==}

        sequence {
            task "Reader1" comp {==}
            task "Writer2" -> comp {==}
       }

        task "Reader2" comp {==}
    }
}
```

Listing 5.4. Port problem in Lingua Franca

when a child node wants to write to the output of the local variable. However, the given program is rejected by the Lingua Franca compiler because port names cannot be the same, even if it would conceptually work if one port is an input port and the other is an output port. This problem can be quickly resolved by adapting the transformation of composite nodes such that the names of local variables are not copied but rather modified so that the shown program can be compiled. The input port for the local variable could be renamed to comp_in and the output port to comp_out. The Lingua Franca compiler would accept that program and the presented concept could be implemented.

Chapter 6

Conclusion

This chapter summarizes the thesis and indicates future work.

6.1 Summary

Within this thesis, Behavior Trees were implemented in Lingua Franca. The transformation is based on a paper by Colledanchise and Ogren, which explains the concept of an FSM construction. That involves converting Behavior Trees into FSMs, which simulate the behavior of Behavior Trees. This concept was directly applicable to reactors in Lingua Franca and has been used for the transformation.

Behavior Trees are implemented in Lingua Franca by designing a DSL for Behavior Trees, which was integrated into the grammar of Lingua Franca, and implementing a transformation for each Behavior Tree component. The implementation supports Behavior Trees with Fallback, Sequence, Task, Parallel, and Condition nodes. Furthermore, it also supports IO for Behavior Trees and local communication between Behavior Tree nodes. The implemented Behavior Trees in Lingua Franca combine the strength of both and make up a beneficial way for designing agents' behavior. In conclusion, the advantages are high modularity, deterministic behavior of agents, support for concurrent execution, high portability and graphical views.

6.2 Future Work

The implementation of Behavior Trees could be expanded with additional functionalities. For example, it could be possible to implement Memory and Decorator nodes, which were explained in Chapter 2. Another feature that could be implemented is support for intuitive operations for reusability and extension of Behavior Trees. The Listing 6.1 shows an example of the feature. In Behavior Tree A, another Behavior Tree named B has been inserted. For that to be possible, the DSL must be adapted to recognize sub-trees as Behavior Tree nodes. Additionally, the transformation must be adjusted to establish connections as expected. A challenge here would be correctly connecting the IO so that the attached Behavior Tree can also read from and write to the output of the higher-hierarchy Behavior Tree. Another challenge would be to support local communication in such Behavior Trees.

```
behaviortree A {
    sequence {
        task "Output1" {=
            printf("1\n");
        =}
        subtree B
    }
}
behaviortree B {
    task "Output2" {=
        printf("2\n");
    =}
}
```

Listing 6.1. Behavior Tree invoking another Behavior Tree

Bibliography

[AGG20] Ramiro A. Agis, Sebastian Gottifredi, and Alejandro J. García. "An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games". In: *Expert Systems with Applications* 155 (2020), p. 113457. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.113457. URL: https://www.sciencedirect.com/science/article/pii/S0957417420302815.

[Ani20] Alex Anikina. "What moves non-player characters: network aesthetics in the gamespace". In: *Parallax* 26.1 (Jan. 2020), pp. 89–102. DOI: 10.1080/13534645.2019.1685785. URL: https://doi.org/10.1080/13534645.2019.1685785.

[CÖ17] Michele Colledanchise and Petter Ögren. "Behavior trees in robotics and AI: an introduction". In: *CoRR* abs/1709.00084 (2017). arXiv: 1709.00084. URL: http://arxiv.org/abs/1709.00084.

[GBJ+20] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wąsowski. "Behavior trees in action: a study of robotics applications". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 196–209. ISBN: 9781450381765. DOI: 10.1145/3426425.3426942. URL: https://doi.org/10.1145/3426425.3426942.

[HLF+22] Reinhard von Hanxleden, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard. "Pragmatics twelve years later: a report on lingua franca". In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 60–89. ISBN: 978-3-031-19756-7.

[ISS+22] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. "A survey of behavior trees in robotics and ai". In: *Robotics and Autonomous Systems* 154 (2022), p. 104096. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2022.104096. URL: https://www.sciencedirect.com/science/article/pii/S0921889022000513.

[OSM+20] Miguel Oliveira, Pedro Mimoso Silva, Pedro Moura, José João Almeida, and Pedro Rangel Henriques. "BhTSL, Behavior Trees Specification and Processing". In: *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*. Ed. by Alberto Simões, Pedro Rangel Henriques, and Ricardo Queirós. Vol. 83. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 4:1–4:13. ISBN: 978-3-95977-165-8. DOI: 10.4230/OASIcs.SLATE.2020.4. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13017.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. "Intelligent agents: theory and practice". In: *The Knowledge Engineering Review* 10.2 (June 1995), pp. 115–152. DOI: 10.1017/s0269888900008122. URL: https://doi.org/10.1017/s0269888900008122.