School of Mechanical and Manufacturing Engineering

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Teleo-Reactive Language for Reactive Agent Behaviour

by

Adam Joffe

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Mechatronic Engineering

Submitted: May 2018 Student ID: z3459791

Supervisor: A/Prof. Jay Katupitiya

Supervisor: Prof. Claude Sammut

ORIGINALITY STATEMENT

Signed [illegible] ……………………………………………….............

Date 25/5/2018 ……………………………………………….............

Abstract

The purpose of this work is aimed at improving the development of agent based reactive behaviour systems. To achieve this, a domain specific language based on the Teleo-Reactive (TR) construct was developed, TR-Lang. It was implemented with an interpreter allowing rapid development of behaviours without the need to compile code. The interpreter, built with C++, is portable and can easily be embedded in an already existing system with the provided API. Crucially, the language provides a shared thread safe data structure accessible through both the language itself and C++ API allowing storage of persistent data and allow optimizing of TR hierarchy evaluation. It was also demonstrated that the language could be implemented in multiple different systems including as a stand-alone language, embedded in a C++ pipeline, or as an individual process within a multi-process system, such as ROS.

Acknowledgements

I would like to thank Claude Sammut and Jay Katupitiya for allowing me to combine my two undergraduate degrees into one thesis, and guiding me through my work. Without their help and understanding I would not have been able to complete such a unique project.

I would like to note the help of Ben Constable, whose expertise with Flex/Bison greatly aided with a crucial aspect of my project. Also, I'd like to note Jingling Xue, whose compiler course was an amazing basis for me to work from for this thesis.

I owe a great gratitude to Heidi Saban, my friends and family for their support and understanding with my involvement in my thesis and work on student projects.

Finally, I'd like to thank the University of New South Wales, particularly the Faculty of Engineering, for their support of my endeavors for my thesis, academic education and involvement in rUNSWift and Sunswift as student projects.

Abbreviations

SPL	Standard Platform League
TR	Teleo-Reactive
TR-Lang	Teleo-Reactive Language
ROS	Robot Operating System
SWIG	Simplified Wrapper and Interface Generator
DKF	Distributed Kalman Filtering
TRAIL	Teleo-Reactive Agent with Inductive Learning
XABSL	Extensible Agent Behaviour Specification Language
XML	Extensible Markup Language
SVG	Scalable Vector Graphics
BP	Behavioural Programming
IR	Intermediate Representation
BNF/EBNF	Backus-Naur Form / Extended Backus-Naur Form
LR	Left to Right, Rightmost derivation parser
LALR	Look-Ahead LR parser
AST	Abstract Syntax Tree
ROS	Robot Operating System
GUI	Graphical User Interface
API	Application Programming Interface

Contents

List of Figures

Chapter 1

Introduction

The problem of controlling the behaviour of robots in a multi-agent environment has been integral to the advancements of robotics. In particular, the ability of these agents to react quickly and appropriately to a changing environment is at the intersection of Artificial Intelligence and Control Systems fields. Various approaches have been researched for this multi-faceted problem. This includes high level communications to reach a consensus for a distributed system, how the structure of the program effects the behaviours and various language-based approaches for defining a behavioural system. Both the Aldebaran Nao [2] and the Lego Mindstorms EV3 [3] platforms provide perfect environments for implementation and testing of the TR-Lang.

The Aldebaran Nao is used by the UNSW Team rUNSWift as the platform for the RoboCup Standard Platform League (SPL) competition [2]. The current implementation for behaviours is a hierarchy of decision trees and state machines [1], implemented in Python to interface with C++ via Simplified Wrapper and Interface Generator (SWIG). There are two major interfaces provided between C++ and Python (Figure 1). On one side the central data on the C++ side, called "Blackboard", is transferred to Python. On the other side a "Request" object that gets passed from Python to C++ using a similar wrapper [1]. One issue with this implementation is the delay in the conversion between Python and C++ is relatively unknown. As a result, sometimes a behavioural decisions may be made with outdated information, and an action performed based on an older decision. Another problem is the current framework and implementation requires between 5-20ms to execute. This takes up a large portion of the runtime budget for the perception thread, which also must perform vision and localization calculations whilst ideally running at 30Hz. Furthermore, over time, the structure set out by the original implementation has been muddied resulting in code that is harder to maintain and debug (Figure 1).

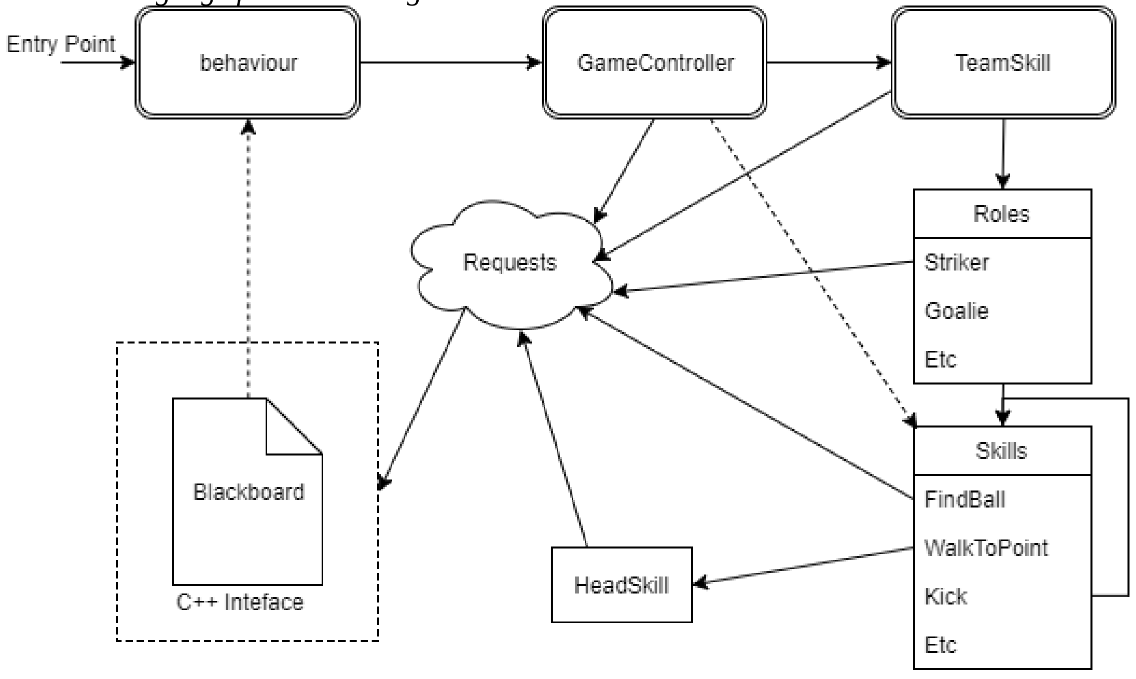


Figure 1. rUNSWift Behaviour Module Infrastructure

The EV3 with ev3dev operating system is used to teach first year computer science and engineering students in the UNSW ENGG1000 course. The students currently complete two projects using this platform, a robot sumo competition and a maze rescue. To accomplish this, the ev3dev platform provides a number of open source APIs for various languages to interface with the hardware, such as C++ [6]. The issue with this current implementation is that it is used by first year students with limited knowledge of agent behaviour, multi-threaded systems or hardware. Often, this results in long development times and poorly structured behaviours that struggle to complete the objectives. Furthermore, this provides a poor basis for understanding of agent behaviour systems, but does provide a reasonable learning system of realistic, not perfect robotics.

The aim of this project is to fix the issues identified with these implementations and others by creating a TR structure based language that cements the logic flow in the semantics. This means that over time, the TR hierarchy and structure will be more constantly maintained. Furthermore, the language is intended to be implemented with the use of a native C++ interpreter. The point of this is to improve the speed of both data flow and execution time; keeping it as an interpreter removes the requirement of compiling which helps reduce development time [1]. By speeding up the execution of the behavioural module, the continuous time assumption for TR is close to being fulfilled. Finally, debugging, profiling and visualization tools have been implemented to aid development of meaningful behaviours.

Similar approaches have been explored with domain specific languages for agent behaviour, and TR programs. They have been implemented on various real robotic systems including RoboCup leagues such as Four-Legged, Standard Platform and Middle-Size leagues. The GermanTeam developed Extensible Agent Behaviour Specification Language (XABSL) [4] has been used in both the Four-legged and Standard Platform league by various teams. TR has also had a basic implementation with a domain specific language compiled into C++ and used on the Middle-Size RoboCup league [5].

Chapter 2

Background

There are many approaches to achieve an objective with a multi-agent system where the environment is changing. As a result of these changes, traditional planning approaches are often not possible due to heavy computational costs each time the environment changes. Because of this, behavioral control of individual agents in the system, or the system as a whole, at a multi-agent level has become a larger area of topic of research. This research covers a wide range of approaches that look at the distributed nature of the multi-agent system, structures of controlling agents given a changing environment and low-level ways of implementing these systems using domain specific languages.

2.1 Distributed Consensus

One approach of achieving a goal as a multi-agent system is distributed consensus. The intention of a distributed system, in this case a physically distributed multi-agent system, is to share information about the external environment to reach a consensus about what actions are required in order to achieve the goal. In particular, the study of self-organizing networked system (swarms) [7] is of particular interest as they investigate complex actions such as flocking [8], and information fusion algorithms [9]. Flocking investigates how multi-agent systems can perform coordinated actions without the need of a leader in the flock. This creates a system without a single point of failure in the case of failures such as connection or byzantine failures [10]. The information fusion algorithms also allow sensor networks to be robust in the event of node and link failures occurring within the multi-agent system [9].The best instances of information fusion can be seen in Distributed Kalman Filtering (DKF). It does this by using consensus filters to calculate an average-consensus of time-varying signals across a distributed system. The behavior of the consensus filter is to asymptotically reach the agreement amongst the agents [7]. Typically, this is done synchronously as a serial algorithm, however, some asynchronous algorithms can be used [12]. This allows the calculations to continue without having to wait for the communication between the nodes. As a result, a consensus through the filter can be reached concurrently. A consensus filter like this can be implemented to reach an understanding of the state of the environment around an agent. As such, this can be used to generate flocking behavior to achieve a single objective without the need of a 'leader' in a fault tolerant system. However, these systems will often use hundreds of nodes with thousands of links [9]. This is consequent to the requirement to detect Byzantine faults as needing 3f + 1 replicas to tolerate f faults [11]. Even though using a trusted subsystem can reduce this to requiring 2f + 1 replicas [11], for rUNSWift, there aren't enough agents to use these sorts of techniques. With the current number of players, five per team, only 1 byzantine fault can be tolerated. Once, robots become mis-localized, penalized or the software crashes, they are no longer able to tolerate any fault and the distributed consensus fails. As such, a different method of achieving an objective needs to be found for an extremely small multi-agent system.

2.2 Agent Behaviour

When the distributed system is smaller and more prone to failure, a different solution is required; as such, a popular choice is to look at individual agent behavior, with some shared information. Hierarchical state machines have been used extensively to implement complex agent behavior [13]. Traditionally, these state machines have worked well, but can be slow to react to their environment. Teleo-Reactive systems implore the idea of a hierarchy structured decision tree using durative actions with a continuous time assumption [14]. The structure forms circuitry during runtime which allows the system to quickly adapt whilst the hierarchy allows complex behaviors to be formed (Figure 3). The core to this structure is a series of conditions (K_i) that invoke a durative action (a_i) on a true evaluation (Figure 2):

$$\begin{array}{ccc} K_1 & \rightarrow & a_1 \\ K_2 & \rightarrow & a_2 \\ & \cdots & \\ K_i & \rightarrow & a_i \\ & \cdots & \\ K_m & \rightarrow & a_m \end{array}$$

Figure 2. TR Definition [14]

However, there are two important properties that the TR should obey to guarantee its robustness; these are *regression* and *complete* property [14]. *Regression* is achieved if for K_m conditions, each condition K_i ($m \geq i > 1$) achieves the condition of a higher condition K_j ($j < i$). This means that each condition will achieve the objective of the ones above it. Thus, if first condition of the root of the hierarchy achieves the overall objective, then the system is guaranteed to achieve it as well. This is assuming that the TR is *complete*, which requires that all conditions form a tautology. TR sequence is thus *universal* if it satisfies these two properties [14]. This results in TR programs being compact due to the circuit based structure as the complexity in the behaviors are realized during runtime. The benefit of structuring behaviors in this manner allows for it to be intuitive and easy to write for the programmer. TR also allows for recursion since a condition can call the set of conditions of which it is a member. This works well for actions such as moving to a point, as sometimes it will require walking to intermediate points [14]. The ability to combine this with the hierarchy structure makes TR extremely powerful for generating complex behaviors that can adapt to any environment to achieve an objective.

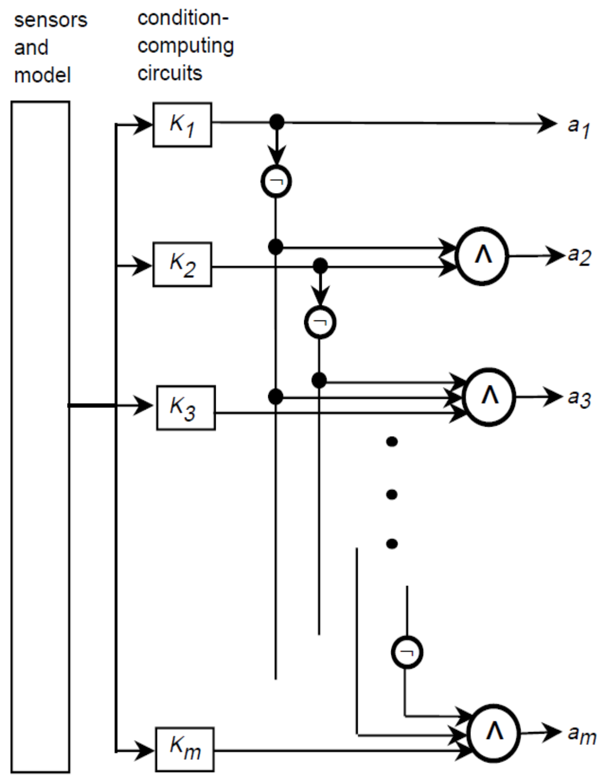


Figure 3. TR Sequence Circuitry

TR structures are very robust, however, they require a few assumptions and are hard to validate. The initial assumption of the continuous time sensing has to be justified first before the behavior can be considered valid [14]. If the continuous time assumption is not met, the system will have a latency before it can adjust its behavior for the environmental changes. One approach to providing validation of a final program is to use a *rely/guarantee* premise that provides reasoning about program and environment in a compositional manner [15]. The *rely* condition describes properties of the environment and the *guarantee* condition describes how the program behaves under the assumption of the *rely* condition [15]. As such, the program doesn't have to ensure the guarantee outside of the *rely* condition. By ensuring this, the program can be validated for any *rely* condition for which it must account.

There has also been implementation of the TR that gives validity to it through the results, such as the implementation on a RoboCup middle-size robot [16]. Similarly, due to the robust design of the TR programs, debugging can be difficult due to accomplish. Often the program will still reach the goal, due to the regressive nature of the conditions, but it may be inefficient. This can also be caused by conditions which never evaluating as *true*, resulting in unused and redundant code that should be removed. A lot of work is still to be done to formalize some of these aspects of TR, but initially the results shown positive benefits over this structure that allows quick adaption to environmental changes over traditional state machine behaviors.

There have also been many extensions to the original TR proposed by Nilson. These include allowing parallel actions to occur, *fuzzy* (*Zadelan*) *logic* instead of Boolean for evaluating conditions and learning algorithms to generate TR programs [17]. In Nilson's original definition of TR programs [14], there was only one action per condition indicated, however, for many robotic systems, it would be beneficial to invoke multiple durative actions at once (Figure 4). As such, the TR programs can be

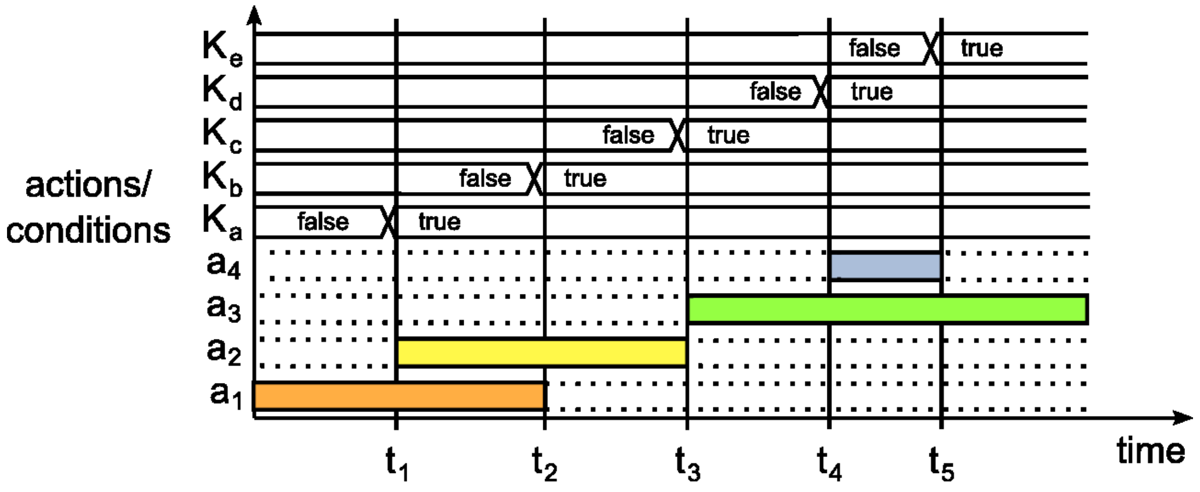


Figure 4. Condition Evaluation with Parallel Durative Actions [16]

extended to provide concurrent running of durative actions for any single condition [16]. This can be even further extended to allow more than one sub-tree of conditions to be evaluated as a result of a flow down in the hierarchy. Another area of improvement is removing the binary Boolean condition resulting in a move more towards *fuzzy logic*. Boolean logic can cause behavior reminiscent of state transition models due to its discrete behavior. However, TR being based in continuous time evaluation conflicts with this concept, as such a continuous-valued logic such as *fuzzy logic* would be an ideal match [18]. This reflects the fact that the sensor inputs aren't as crisp as binary Boolean logic. Often for traditional TR conditions an arbitrary split in the range of sensor inputs is made to choose what action is taken as a result. This becomes extremely useful when a smooth transition between previous and current actions would be preferable [18]. It does this by changing the first true condition paradigm implied by the original definition. In this case, a condition may become "partially" energized with subsequent other conditions also being partially energized. As such, there is a case that all conditions can become in some way energized, often at varying levels. However, this approach does require an amount of care, since the membership function shape significantly affect the performance creating an undesirable effect.

While the initial implementations in TR use manually written conditions and actions, the structure of it lends itself to being good candidate for machine learning approaches [14]. One of these involves the use of genetic programming to provide suitable, efficient behaviors for agents [19]. By creating a metric for evaluating the fitness of TR programs in completing a task, a random population of TR programs can be bred together and selected to produce an optimal solution. Kochenderfer [19] showed that for a simple block stacking task, 300 generations produced several TR programs that could solve 99% of the 11,500 generated problems. However, the final TR programs had a large amount of redundant code generated from the cross-breading. But, once this excess code was stripped, the TR program was simpler requiring fewer rules and sub-trees than most human produced programs [19]. The Teleo-Reactive Agent with Inductive Learning (TRAIL) system also applies machine learning techniques to the TR program structure [20]. This is an action-model learning system with a backward-chaining planner to construct plans by building models to describe the effects of actions in an environment. This system, in particular, is able to handle continuous state variables, non-atomic actions, unreliable sensors, and unpredictable events in line with the TR tree control structure. An architecture such as this employs user defined goals given to a planner to create TR trees that are then executed. After this, the execution is recorded and fed into the learner, using an inductive logic programming algorithm, to adjust the planner to perform better [20]. There is a wide range of machine learning techniques that can be used to build and optimize TR programs, but a lot of work is required on complex physical systems with multiple agents, such as RoboCup SPL, to learn the required behaviors in a reasonable time. Despite these limitations, the TR programming paradigm is still extremely useful for compact, intuitive agent behavior specification.

2.3 Domain Specific Programming Languages

Finally, there are various domain specific programming languages for controlling robot behavior. Many robotic behaviors have been programmed using an efficient low level language such as C++. However, these languages are often not well suited for this high level application. They require a large amount of developer time to extend and maintain complex behavior control systems [23]. As such, there has been a move to look for a high level behavior specification language that is quick and intuitive to develop with whilst running efficiently. One solution is the Extensible Agent Behavior

Specification Language (XABSL) [23] used by 2016 and 2017 RoboCup SPL champions, B-Human [21, 22]. This is a domain specific language uses a hierarchical state machine structure as the basis for its implementation [23]. These state machines are called “options” and are self-contained state machines that can then flow down control to lower level state machines (Figure 5). The lowest level of behaviors, *basic behaviors*, are still written in C++, these lie at the terminal ends of the state machine hierarchy. XABSL is complete with Boolean logic, arithmetic operators and the ability to insert custom arithmetic functions.

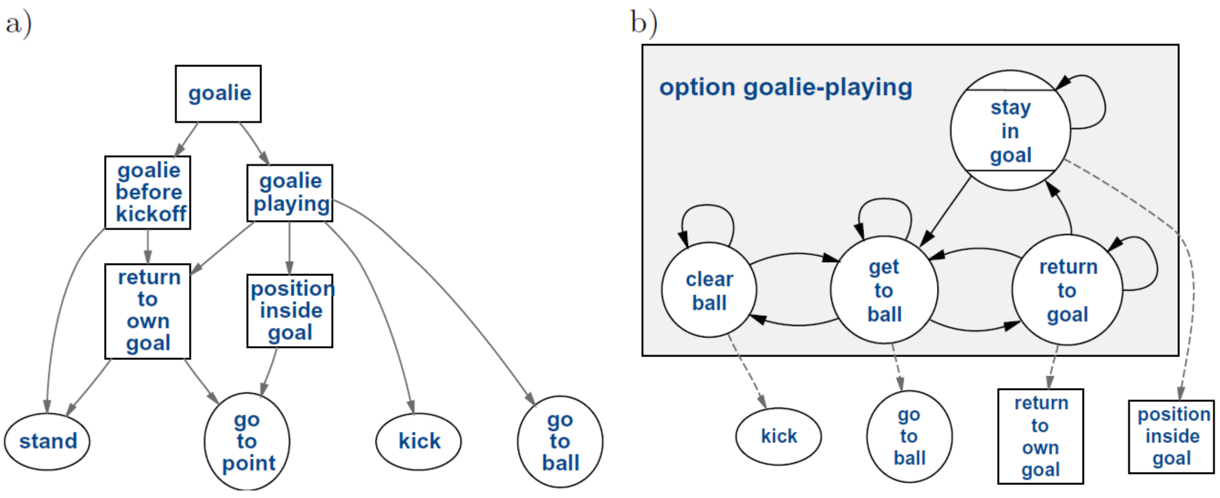


Figure 5. a) Option Graph of Goalie Behaviour b) Internal State Machine of Goalie-Playing Option [23]

XABSL has a XABSL-compiler, written in Ruby, which generated four documents [24]:

1. Intermediate code for runtime system
2. Debug symbols for debugging
3. Symbol files for code completion and syntax highlighting in text editors
4. XML representation XABSL specification

This provides XABSL with a wide arrange of debugging and visualization tools, such as the conversion of the XML into Scalable Vector Graphics (SVG) to allow the user to understand the structure they are creating. There is also a *Xabls Profiler* tool which can analyze the behavior over time, allowing the user to detect state oscillations and unsure states [24].

The approach for XABSL was to create a new language required for these behaviors. However, other approaches have been to use a suitable language that already exists, and create infrastructure for it to allow these sort of intuitive agent behavior implementation. One of these is Behavioral Programing (BP) for Erlang. Erlang has desirable characteristics making it suitable for distributed, concurrent, fault-tolerant real-time systems [25]; this makes it ideal for use on multi-agent systems. As Erlang has emphasis on threads, BP implements an architecture involving behavior threads (b-threads), which each aim to accomplish their own objective [26]. Each b-thread performs its required computations to decide on an action, then a synchronous auction is held between the active b-threads. Each b-thread bids with three parts [26]:

1. Request: the actions this b-thread proposes *to do*
2. Waited-for: the actions this b-thread wants to be *notified of*
3. Blocked: the actions this b-thread *forbids*

The arbitrator then decides what actions to take as a result, where the rules for resolution are either based on priority, by default, or decided by the user. This structure allows for concurrent distributed processing to reach a unified decision on the behavior to perform. The main issue with this approach is the required constant syncing of the b-threads and the potential for one b-thread to hold up the auction by not syncing. One solution around this is to allow auctions to proceed with all b-threads currently waiting for syncing and ignoring those still computing [26]. While this approach seems logical, the application to RoboCup may not be ideal due to how rapidly changing the environment can be; as such each b-thread would likely have to be complex in their own right, or have a complicated arbitrator for deciding what actions should be priorities in which instances.

Chapter 3

Language Definition and Implementation

To build an interpreter, five distinct components are required (Figure 6), this can be split into two stages, the front end and the back end. The front end converts and validates the code file into an Intermediate Representation (IR), and consists of the scanner, which tokenizes the code, the parser, which checks the syntax of the code, and the semantic analyzer, which checks the semantics of the

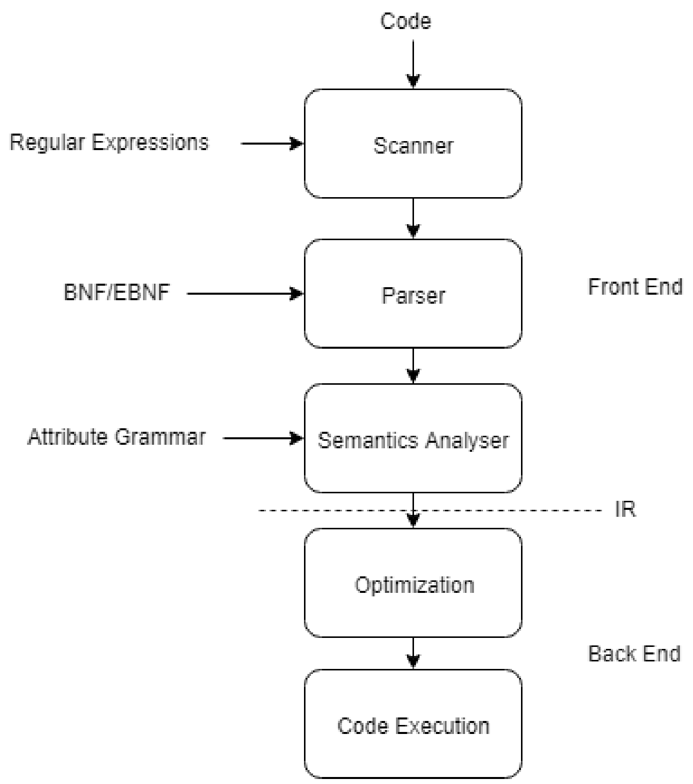


Figure 6. Interpreter Pipeline

code. The back end takes the IR and optimizes it, then executes it. TR-Lang is intended to be executed multiple times, so all stages up to execution are completed before the first execution to improve efficiency. The interpreter is built to include multiple configurable levels of output information (error, warning, info and debug) which provides useful information about the execution of the interpreter and the TR source code.

3.1 Syntax and Semantics

The tokens that make up the syntax for the language consist of six groups of tokens: *literals*, *identifiers*, *keywords*, *operators*, *separators* and *blackboard-symbols* (Appendix A1.1). Six types of *literals* (Integer literals, Hex literals, Boolean literals, Double literals, String literals, Vector literals) tokens describe five underlying TR-Lang types (Integers, Booleans, Doubles, Strings, Vectors). *Identifiers*, in TR-Lang, require the first character to be alphabetic, then any other alphabetic, numeric or underscore character can be used; this is consistent with most common programming languages. The *keywords* consist of Python-like if/elif/else control flow statements, imports and Boolean statements, as well as some other tokens. The syntax for TR-Lang is described using the standard Extended Backus-Naur Form (EBNF) for the relationships between language elements (Appendix A1.2). A TR-Lang program consists of any number of five different declarations: *blackboard*, *variable*, *function*, *TR-node*, *once* declarations and a series of shared components.

The blackboard is the persistent, shared, thread-safe data structure that sits at the core of the language. It consists of multiple modules, indexed by a name, with the *trmodule* always being

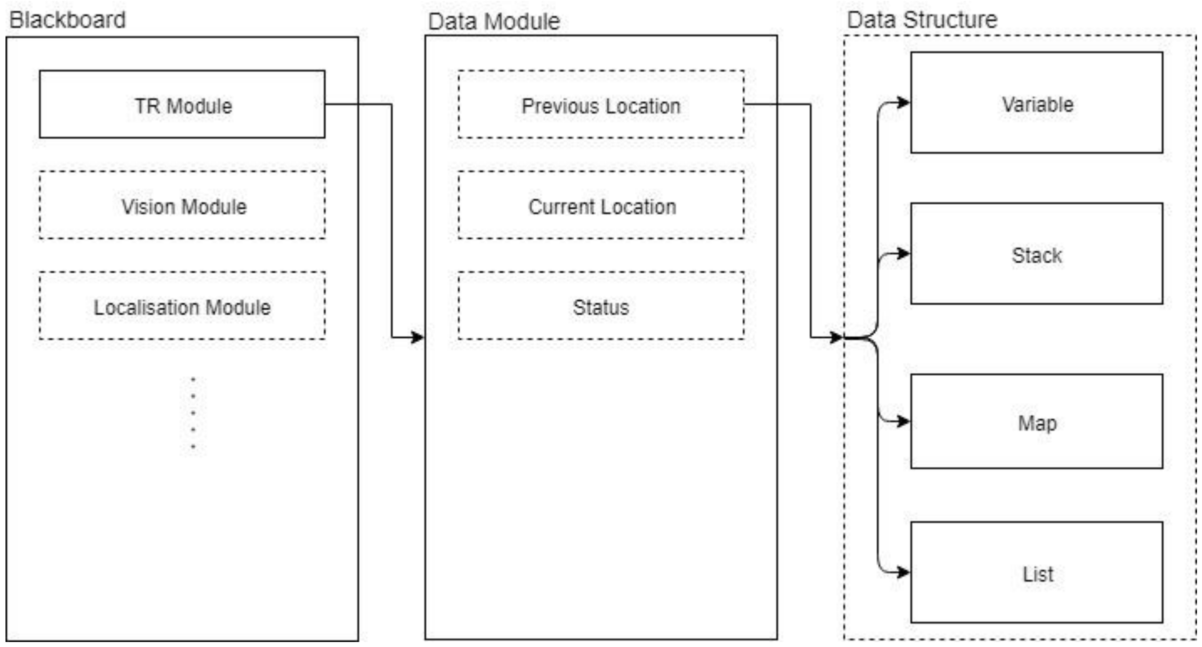


Figure 7. Blackboard Structure

included by default as the languages module (Figure 7). These modules serve a similar purpose to C++ namespaces by allowing data structures to share names, as a consequence of being in different modules. Within each module there can be any number of independent data structures, these can be any of four common data structures: Variables, Stacks, Maps/Dictionaries and Lists (Figure 7). All data structures in the blackboard are accessible from both within the language itself and through the C++ interpreter API. Within TR-Lang blackboard data structures can be created via a *blackboard declaration*, with a given structure and value type. These are prefixed using the *init* keyword and identified with a user chosen identifier. Structures declared from within TR-Lang are always inserted into the *trmodule* module.

TR-Lang interpreter is single pass, so all variables, functions and TR-nodes are required to be declared before use since C-style forward declarations are not provided. Variables can only be declared in TR-Lang in a global scope and are required to be declared with an initializer *value expression*. On each execution, the value of these variables is reset back to the initialized values, so they cannot be used for persistent state. The blackboard serves to contain persistent data for evaluation. Functions, like most languages, provide a method to modularize and reuse code; it provides if/elif/else control flow statements and returns the result of any *value expression* allowing recursion to also be possible. TR-nodes are the core component of the language. Similar to functions, they can take in any number of user defined arguments except for the entry point node, which takes no arguments. The entry point node by default is *main*, however, it can be renamed to whatever the user desires via the correct program options. These nodes are the key component of the TR hierarchy and contains the decision circuitry, in the form of condition statements, crucial to creating the agents reactive behaviour. The nodes have two syntax components that enforce regression and tautology of rules that are fundamental to the TR decision-action paradigm described by Nilson [14]. The regression is checked by an *achieves* statement (Figure 8) at the end of each condition which checks that each achieves the goal of the named condition above it, or the node of which it belongs. To ensure a tautology of conditions, each node is required to have a default catch-all condition (*true*) which guarantees at least one condition is chosen in each evaluated node (Figure 8). When a condition evaluates as true, the list of actions it contains will be executed in order, then it will return out from the node.

```
// TR Node
find_ball() :-
ball_seen : ball_frames() > 10 -> eyeLED(RED), approachBall(v) achieves
find_ball;
search_known : last_seen_frame() < 5 -> headspin(LEFT) achieves ball_seen;
true -> spin_search()achieves search_known;
```

Figure 8. Simple TR Node Example

Finally, there is a special declaration that can be made that forces the interpreter to only evaluate the statements inside on the first execution, the *once* statement. This is useful when initializing the persistent data of the blackboard, or initializing external components, such as hardware. Function calls are done with C-like syntax, referencing the function by identifier, passing any number of arguments between parentheses, and a single return value that can be void. The same call syntax works for calling TR-nodes, but they can only have a void return value. They can also only be called from within another TR-node, this is how the core hierarchy structure is formed. The TR-Lang C++ API also provides a method for users to define their own built-in variables and functions that are then usable from within the language itself. Functions can be registered as a built-in with either a basic function pointer or via binding it to a callable object. The ability to bind allows functions with non-valid TR-Lang-C++ type parameters to still be used by binding an argument to it. Similarly, member functions of objects can be bound by passing the object to the call. There are five valid primitive C++ types usable for TR-Lang include, int, float, double, std::string and bool, as well as the Vector type which has a default implementation, but users can modify it to use their own Vector class if desired. However, the language itself is dynamically typed so type declarations are not required for functions and variables, and type coercions are done when required. The only part required to have explicit type declaration is the blackboard, as it essentially exists within C++ for ease of interfacing and efficiency of use. This is necessary, as in most systems, large amounts of load/store operations are in practice being performed on the data structure.

3.2 Interpreter Front End

Two of the three front end components, the scanner and the parser, can be created using generators. For this, a popular C++ based scanner/parser pair, Flex and Bison, was used. Flex takes in token descriptions in the form of a regular expression and outputs a compilable C++ source file that handles file streams, tokenizes input and provides them for Bison. The scanner is also responsible for handling *import* directives by switching input file streams as directed by the import. Additionally, it provides protection against importing the same file multiple times with an import guard mechanism. Furthermore, there are two inbuilt TR-Lang libraries (math.lib and utility.lib) provided based on functionality from C libraries available via the import directive. Bison takes in a complete set of grammar rules in BNF or EBNF notation, and generates a LALR(1) parser table [25] in a compilable C++ source file. This reads in tokens provided by the Scanner and attempts to construct an Abstract Syntax Tree (AST) according to the table rules. Once the AST is constructed, the Semantic Analyzer can be run. Since there are no widely accepted generators for the semantic analyzer, it was built by hand and does static analysis on the code to ensure it is correct. For the TR-Lang C++ API, the Scanner and Parser build the AST during construction of the interpreter object. Once this is done, users can define built-ins and blackboards modules and structures as they desire before running the pre-process function that triggers the semantic analyzer to run. The semantic analyzer must be run after the built-ins and blackboard changes such that the analyzer can use them during static analysis. The semantic analyzer also maintains two tables. One is the symbol table that tracks function, TR-node and variable declarations to ensure these are declared before they are used. This table is later carried over to the executor to index when variables are used and set and function/TR-nodes are called. The second table that is maintained is the regression table, which is only used by the semantic analyzer when validating the *achieves* statements to ensure that they point to a previous condition.

3.3 Intermediate Representation

The IR for TR-Lang uses an AST as the data structure for encoding the information from the source files for execution. The AST consists of ~45 different concrete nodes, each of which falls into one of seven types of abstract node (Figure 9). The semantic analyzer, executor and all the tools are able to use this IR to achieve their objective by leveraging the Visitor design pattern [27]. This pattern is used as an alternative to having each node explicitly calling their children with virtual calls in their

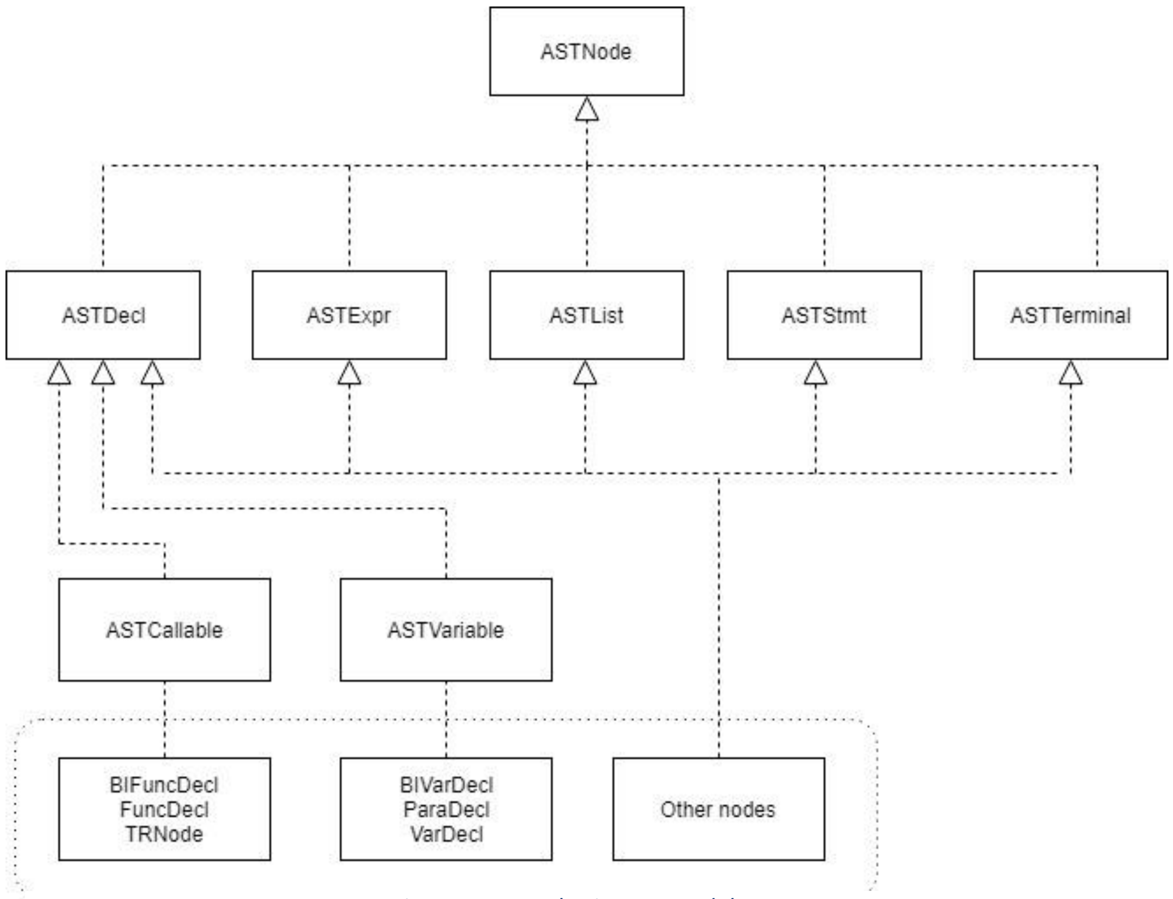


Figure 9. AST Inheritance Model

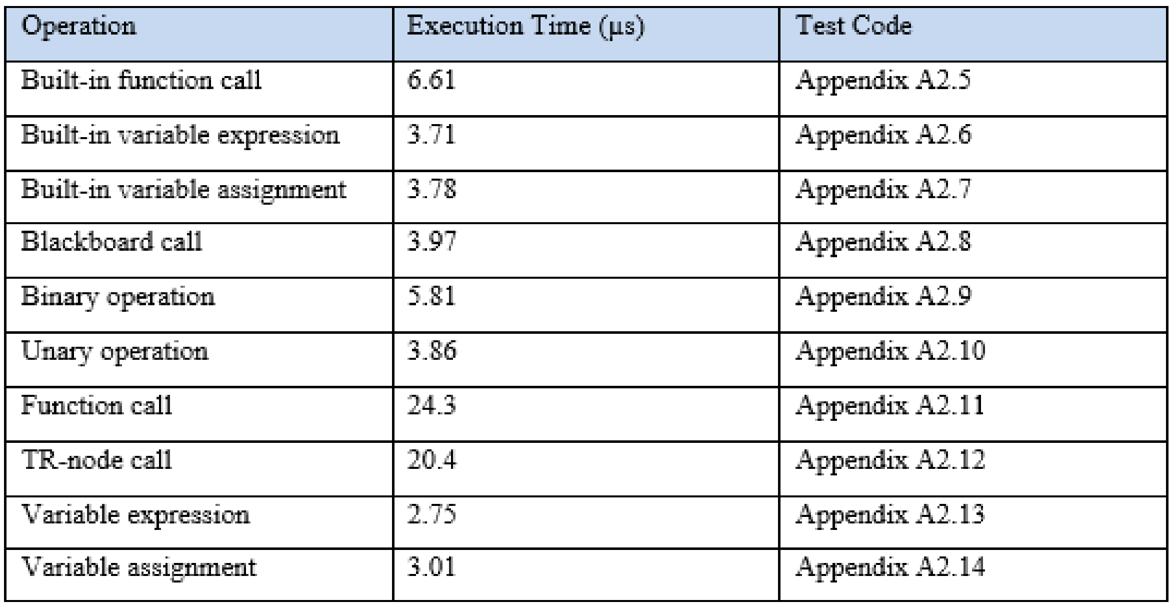
individual implementations. This was chosen to avoid having all the implementation for the semantic analyzer, executor and tools within the AST files. Instead, each component can be implemented in their own file by inheriting the visitor interface and implementing its abstract functions. This provides more modular code that is easier to maintain and read since all the relevant parts are within a single file.

3.4 Interpreter Back End

The interpreter backend consists of the executor which walks through the AST tree via the Visitor interface. As it walks the tree, any variable assignments and expressions are indexed from the symbol table are values used/updated directly to the AST node reference stored in the table. Similarly, for function and TR-node calls, the declarations are stored in the table, then an abstract call method is used to achieve dynamic dispatch of function calls. This allows all calls to be abstracted from the objects they are calling on, providing a generic interface with which to implement TR-Lang and built-in call objects. TR-Lang is dynamically typed, thus type coercion is done implicitly when operating on values. To achieve this behaviour, an abstract generic underlying type is created to which all concrete types are mapped. Each of these can be constructed, given any other type, provided coercion is valid between these types. For example, a TR-Lang double type can be constructed from a TR-Lang integer type. Since TR-Lang is intended to do repeated loop-like or interrupt-like execution, when the executor is run, it first re-initializes all global variables before starting at the designated TR-node entry point. During execution, if a runtime error occurs, a *TR_Runtime_Exception* is thrown exiting out of the program, or the user can catch is via the C++ TR-Lang API. Similarly, if an *exit* statement is reached, an *ExitTR* object is thrown containing a Boolean exit status and string message. This can then be handled the same as a runtime error.

3.5 Performance

There are a number of approaches taken in the TR-Lang implementation to try and improve the performance of the language overall. The simplest approach that is incorporated is the use of lazy evaluations and returns. For this, evaluations of expressions, in particular Boolean expressions such as '*and*' and '*or*', will attempt to evaluate as few operands as possible to achieve the result. With *and* expressions, this means that if the first operand is deduced as false, then the entire expression will be evaluated as false without having to further evaluate the second operand. Similarly, for *or* operation, if the first expression is deduced as false, the entire expression will be evaluated as true. For functions, this lazy approach is done with return values. As such, as soon as a return statement is reached, the function immediately returns. This is done by throwing the value object the function returns, causing the call stack to unravel. The value it then caught by the callee with a *try-catch* block. The same approach to only evaluate the necessary parts can also be applied to condition evaluations that rely on data from the blackboard, however this has yet to be implemented. This is done by each data structure within the blackboard having a *changed* flag that is set when the value or state is updated from its current, and unset on read operations from within TR-Lang. A dependency tree can then be constructed from what information each condition requires for its evaluation, and a map of what data determines the circuitry at each level of the TR hierarchy. Then, when a value in the blackboard is changed, only the required nodes can be reevaluated, rather than the entire tree structure itself. Finally, during semantic analysis, as much static evaluation is done on the code to pre-evaluate expressions if it can be done without runtime information. For example "*(1 + 2) * 4 > 10*" can always be deduced as false prior to execution. For normal interpreters, this wouldn't provide additional performance, as the analysis is done in a single pass with the execution. However, since TR-Lang's execution is intended to be run multiple times, this shifts computational effort away from execution time and into static analysis. TR-Lang performance metrics were measured over one million executions and measured using the inbuilt profiler (Figure 10):



Operation	Execution Time (µs)	Test Code
Built-in function call	6.61	Appendix A2.5
Built-in variable expression	3.71	Appendix A2.6
Built-in variable assignment	3.78	Appendix A2.7
Blackboard call	3.97	Appendix A2.8
Binary operation	5.81	Appendix A2.9
Unary operation	3.86	Appendix A2.10
Function call	24.3	Appendix A2.11
TR-node call	20.4	Appendix A2.12
Variable expression	2.75	Appendix A2.13
Variable assignment	3.01	Appendix A2.14

Figure 10. TR-Lang Operation Performance Metrics

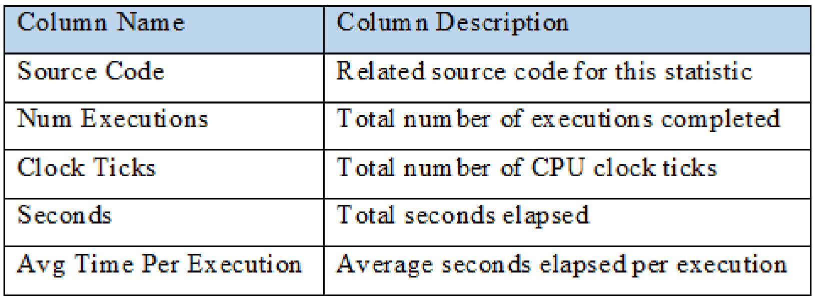
Chapter 4

Language Tools

TR-Lang provides three inbuilt tools for analyzing source code and runtime execution to allow for higher quality code and behaviour. These tools are easily accessible using the interpreter program options and have configurable output files via the configuration file.

4.1 Profiler

The profiler runs during execution and logs performance statistics about the program during runtime. This is extremely useful for users when optimizing the runtime of the code for investigating which parts of the code are causing poor performance. Since TR-Lang is designed to execute multiple times, the profiler allows the yield rate of the output to be decided via the program options. By default, the yield is set at 0, which results in the output only occurring after the program exits gracefully, when set to a non zero value *N*, it will out every *N* runs. The profiler captures the source code, and evaluates each critical segment for the following statistics (Figure 11, Appendix A2.1):



Column Name	Column Description
Source Code	Related source code for this statistic
Num Executions	Total number of executions completed
Clock Ticks	Total number of CPU clock ticks
Seconds	Total seconds elapsed
Avg Time Per Execution	Average seconds elapsed per execution

Figure 11. Profiler Output Columns

4.2 Drawer

One of the crucial aspects of the TR structure is the ability to create a hierarchy of action-decision nodes. To allow users to better understand how the TR-Nodes form this hierarchy the drawer constructs a tree diagram of the TR structure implemented by the source code. Each node of the tree represents one TR-node, with the appropriate directional links between them, labelled with the name of the condition to which the link belongs. Function, blackboard and TR-node calls are also displayed with the same links, but with different shaped and coloured nodes to represent each (Figure 12). The drawer runs after the semantic analyzer during the pre-process stage and generates a Graphviz [28] *dot* file as well as a *png* file.

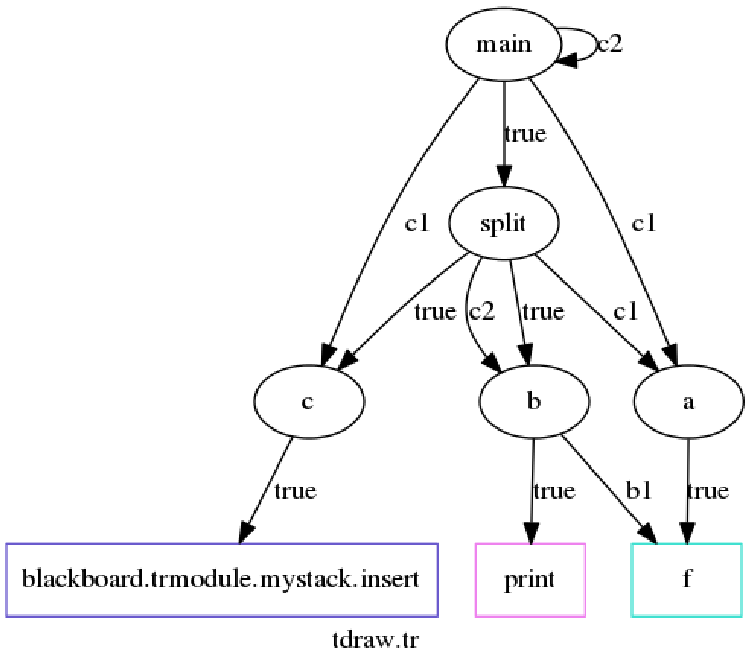


Figure 12. Sample Drawer Output (generated from code in Appendix A2.3)

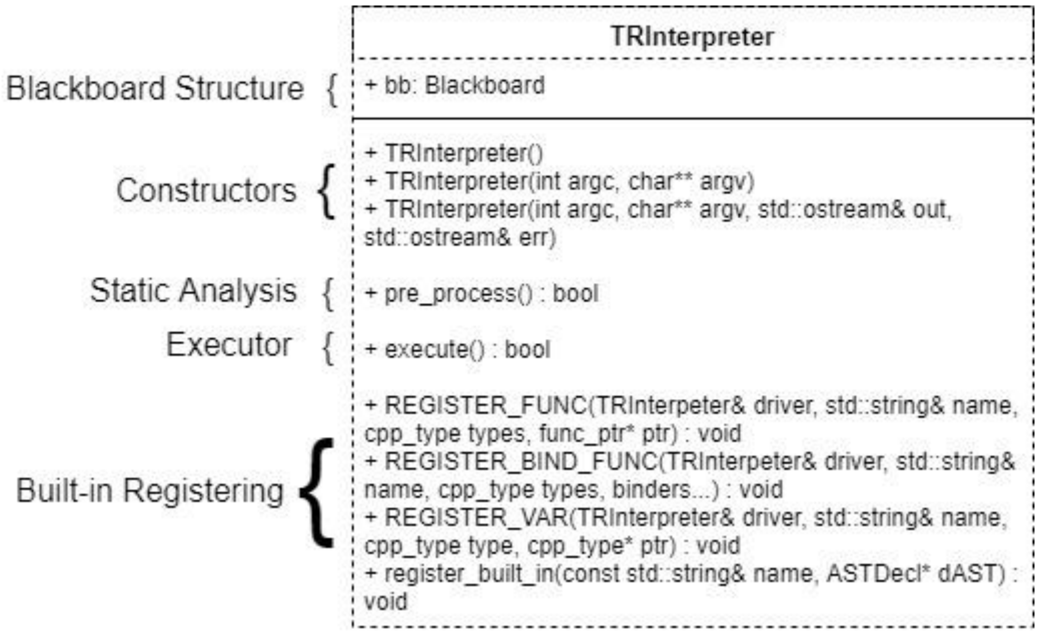
4.1 Debugger

The debugger, similar to the profiler, runs during execution time to log runtime information about the user program. It outputs information about what source code is being executed and the value of expressions being evaluated (Appendix A2.4). Like most debugger tools, a complete stack trace and depth information is also maintained in the output. The output itself is purposely hard to read to attempt to reduce file sizes, whilst still being human readable. This was chosen, since most agent systems run on an independent platform where additional network traffic would cause undesirable performance losses as well as potentially interfering with the operation of the agent system itself. As such, the debugger merely outputs this log file which can later be human read or parsed into a separate application to replay the execution of the program as if it was running in real time. As with most debuggers, using it slows the execution of the program and is not intended to be run on a final system, but instead as a tool for understanding how the code is executing. This is particularly useful for TRs decision-action hierarchy, as the user can see which circuit paths are being executed as a reaction to which sensor evaluations are made. This provides a method of improving and tuning the agent behaviour to achieve the designed goals of the system.

Chapter 5

Application and Usability

TR-Lang's C++ API (Figure 13) provides multiple ways of applying the TR structured behaviour language to an agent system. The generic nature of the interpreter and its components allows it to fit the role of most agent systems. The API provides three choices of constructors for the *TRInterpreter*



```
TRInterpreter
+ bb: Blackboard
+ TRInterpreter()
+ TRInterpreter(int argc, char** argv)
+ TRInterpreter(int argc, char** argv, std::ostream& out,
std::ostream& err)
+ pre_process() : bool
+ execute() : bool
+ REGISTER_FUNC(TRInterpeter& driver, std::string& name,
cpp_type types, func_ptr* ptr) : void
+ REGISTER_BIND_FUNC(TRInterpeter& driver, std::string&
name, cpp_type types, binders...) : void
+ REGISTER_VAR(TRInterpreter& driver, std::string& name,
cpp_type type, cpp_type* ptr) : void
+ register_built_in(const std::string& name, ASTDecl* dAST) :
void
```

Figure 13. TR-Lang C++ API

object, a basic default constructor, a *program-option* only constructor, and a *program-option* and stream constructor. The program options take input as a series of Unix-style switches (Appendix A1.3), and the stream constructor remap the standard output and error streams the interpreter uses. During construction, the scanner and parser will generate the IR data structure that is given by the provided source code specified via the appropriate command. The static analysis is then run via the *pre_process* function, this is required to be run before execution, and also all built-ins or additional blackboard module needs to be declared prior to running *pre_process*. For the blackboard, the registering can be done using the publically exposed blackboard stored within the *TRInterpreter* object using the provided add/remove functions. Similarly, for built-in variables the registering macro takes any C++ variable and its type, that's allowed by TR-Lang, and maps it to the given name. For functions, two macros are provided, one provides a simple way of registering free functions and a more complex one that allows for variadic parameters to bind to the function. This binding macro allows object member functions, lambda functions and functions with fixed parameters passed to it. Finally, the *execute* function provides the core running of the program and can be called repeatedly in a loop returning a Boolean success status. All functions for the API, except for those that operate on blackboard data structures, are not thread safe, and the user should use locks or similar concurrency methods if a race condition may occur.

5.1 Stand-Alone Language

The simplest way to use TR-Lang is as its own stand-alone language; this sets all system control flow to be done by the user TR-Lang source code. Often this will require hardware interfaces to be registers into the interpreter API, such that lower level systems are able to be controlled from the higher-level language. In this application, the behaviour is constantly being reevaluated allowing the system to approach a continuous system as required by the TR structure.

A demonstration system was built for the EV3 platform used by first year software engineers. The platform runs with an ARM processor, as such TR-Lang sources and libraries were cross-compiled using the GNU ARM compiler, *arm-linux-gnueabi* [29]. Then the C++ ev3dev hardware API [6] was registered into the interpreter object so all low level functionalities would be available from the high level TR structure. The platform was tested for basic functionality to ensure that the hardware was correctly usable with the following hardware (Figure 14):

- Large motor
- Medium motor
- Touch sensor
- Ultrasonic sensor
- Gyro sensor

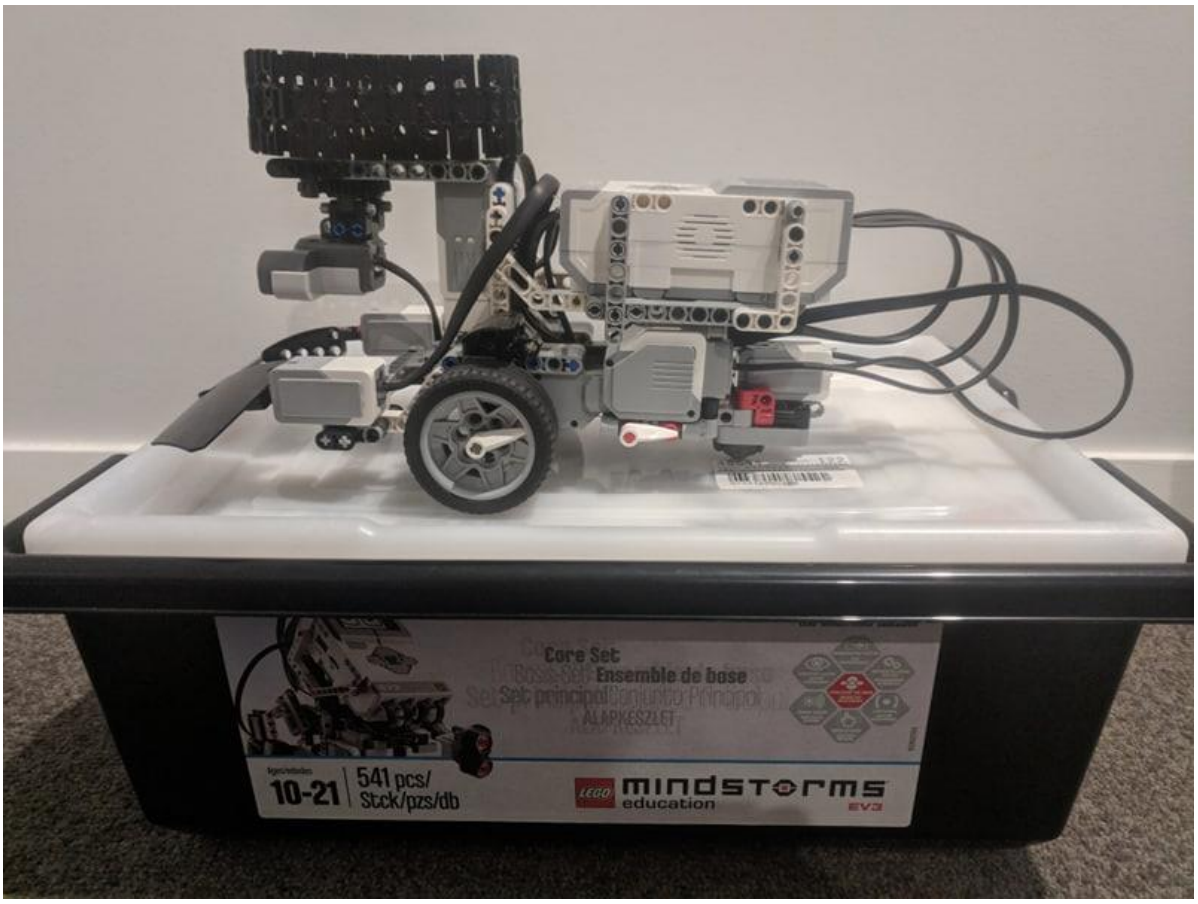


Figure 14. EV3 Demonstration Platform

5.2 Embedded Interpreter

The interpreter is also able to be used as an embedded component in an entire pipeline for a system. This is particularly useful when the system has to do significant processing of sensory data before a behavioural decision can be evaluated. In this case, the program control flow is done by the core system in C++, with data storage and evaluation done through the TR-Lang C++ API. For example, in the rUNSWift perception thread, vision information is first analyzed for field features, ball locations and robot detection. After that, this information is then used to update the localization equations for the robot and in both steps, updates the globally accessible data structure with relevant information. Then finally, the behaviour module is run, using the information in the global data structure to make decisions about what actions should be taken. Currently, the module uses Python to implement its solution, but TR-Lang can be used to replace and improve upon the current implementation in this pipeline. By replacing the current global data structure with TR-Lang's blackboard, it make use of the evaluation optimization of the TR tree to only reevaluate portions of tree where sensory inputs have changed. Allowing TR-Lang to have this control of the central shared data structure, allows any system where the interpreter isn't the core flow controller to have improved efficiency where reactive behaviours are required.

5.3 Multi-Process System

One of the ways robotic systems can be implemented is with the use of multiple processes, each doing a particular role to allow the system to function as a whole. A popular infrastructure that achieves this is Robot Operating System (ROS) [30], which interlinks processes with a subscriber-publisher system for messaging between processes. TR-Lang can easily be implemented into this kind of system since interrupt-like behaviour can be attained for the evaluation of the TR circuitry. To do this the blackboard is required to store the relevant sensory information from other processes, such that each time an update is required, there would be a message given to the TR-Lang process. This, by its nature, notifies when the TR structure may need reevaluating, hence interrupt-like behaviour. Also, since the blackboard data structure is thread safe, updates to it can be done by multiple threads simultaneously without causing the evaluation of the TR-Lang source code to fail. As such, multiple processes can safely transfer data to and from TR-Lang through multiple threads within the TR-Lang process.

Chapter 6

Conclusion

The central features of TR-Lang have been designed and implemented allowing for an efficient domain specific language for reactive agent behaviour. TR-Lang interpreter and dynamic typing allows for quick development cycles allowing users to build and test behaviours in an agile manner. The language and its core data structure, the blackboard, provide an excellent key component to any agent system. Additionally, the tools provided natively for the language allows programmers to gain a deeper insight into their program. The profiler and debugger ensure the source code is executed efficiently and quickly, while the drawer provides a graphical representation of the TR hierarchy which allows for greater understanding of the behaviour encoded in the program. As shown, TR-Lang can be implemented into many systems, including already established robotic systems such as rUNSWift and ROS. Overall, the language provides a more intuitive way of generating agent behaviour than current general purpose languages such as C++ or Python, whilst using the underlying TR structure to be robust a system as possible.

6.1 Future Work

The core language design and base implementation has been completed as part of this thesis, but there are many improvements and extensions that can be incorporated into TR-Lang. These range from small functional or grammar modifications, such as adding *modulo* operators, to larger changes that improve performance or allow the user more control over multiple aspects of the system. One such grammar improvement is to allow Vectors to be any dimension. This allows users to provide their own Vector implementation that is more or less dimensions than the current implementation allows. Another minor, but useful, improvement that can be implemented is to allow *exit* statement to accept expressions values for the status and message arguments. Currently, only Boolean and string literals are valid arguments, but allowing for expressions provides additional flexibility for the user. However, this does reduce type safety, since not all values can be coerced to Booleans for the status argument. A critical optimization component that still has yet to be implemented is circuitry reevaluation dependency tree, but all the components are in place for it to be implemented. This would link sensory data placed in the blackboard data structure with conditions within TR-nodes such that the highest node in the tree with a change, and below, is reevaluated. This means that all nodes above the highest reevaluation node, aren't reevaluated since their decisions are still valid. For small TR-Lang programs with shallow hierarchy this won't provide much efficiency change and might slightly reduce performance. However, for large complex systems, with a very deep hierarchy, this provides a large performance improvement. Furthermore, the current implementation doesn't account for a system with large amounts of noise or errors in sensory inputs, such as the Nao platform the rUNSWift. To fix this, an input and output hysteresis should be introduced into the language which is controllable through the language itself. This would act as a filter by buffering the data going into the blackboard, and actions executed within TR-node conditions. Without this, the system may rapidly switch between circuitries, causing inefficiencies in the system and potentially crippling its ability to achieve goals. Finally, the other extension that can be implemented is the debugger GUI application. This takes the log file generated by the TR-Lang debugger, parses it and allows a visual walkthrough of the code, as if it was running step by step. It could also have additional information panels displaying current state of blackboard, current path through the TR hierarchy tree, and call stack. All these improvements aim to either enhance usability and performance of the TR-Lang, allowing it to become a more powerful behavioural language than it is at this point.

Bibliography

[1] Claridge, David. *Generation of Python Interfaces for RoboCup SPL Robots*. Taste of research report, The University of New South Wales, 2011.

[2] Harris, Sean, et al. "Robocup standard platform league-runswift 2012 innovations." *Australasian Conference on Robotics and Automation*. 2012.

[3] ev3dev.org. (2018). *Ev3dev Home.* [online] http://www.ev3dev.org/ [Accessed 12 May 2018].

[4] Lötzsch, Martin, et al. "Designing agent behavior with the extensible agent behavior specification language XABSL." *Robot Soccer World Cup*. Springer, Berlin, Heidelberg, 2003.

[5] Gubisch, Gerhard, et al. "A teleo-reactive architecture for fast, reactive and robust control of mobile robots." *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, Berlin, Heidelberg, 2008.

[6] ev3dev.org/docs/programming-languages. (2018). Ev3dev Programming Languages. [online] http://www.ev3dev.org/docs/programming-languages/ [Accessed 13 May 2018].

[7] Olfati-Saber, Reza, J. Alex Fax, and Richard M. Murray. "Consensus and cooperation in networked multi-agent systems." *Proceedings of the IEEE* 95.1 (2007): 215-233.

[8] Olfati-Saber, Reza. "Flocking for multi-agent dynamic systems: Algorithms and theory." *IEEE Transactions on automatic control* 51.3 (2006): 401-420.

[9] Olfati-Saber, Reza. "Distributed Kalman filter with embedded consensus filters." *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*. IEEE, 2005.

[10] Driscoll, Kevin, et al. "Byzantine fault tolerance, from theory to reality." *SafeComp*. Vol. 3. 2003.

[11] Distler, Tobias, Christian Cachin, and Rüdiger Kapitza. "Resource-efficient Byzantine fault tolerance." *IEEE Transactions on Computers* 65.9 (2016): 2807-2819.

[12] Tsitsiklis, John, Dimitri Bertsekas, and Michael Athans. "Distributed asynchronous deterministic and stochastic gradient optimization algorithms." *IEEE transactions on automatic control* 31.9 (1986): 803-812.

[13] Bordeleau, Francis, J-P. Corriveau, and Bran Selic. "A scenario-based approach to

hierarchical state machine design." *Object-Oriented Real-Time Distributed Computing, 2000.(ISORC 2000) Proceedings. Third IEEE International Symposium on*. IEEE, 2000.

[14] Nilsson, Nils. "Teleo-reactive programs for agent control." *Journal of artificial intelligence research* (1994).

[15] Dongol, Brijesh, Ian J. Hayes, and Peter J. Robinson. "Reasoning about goal-directed real-time teleo-reactive programs." *Formal Aspects of Computing* 26.3 (2014): 563.

[16] Gubisch, Gerhard, et al. "A teleo-reactive architecture for fast, reactive and robust control of mobile robots." *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, Berlin, Heidelberg, 2008

[17] Morales, Jose Luis, Pedro Sánchez, and Diego Alonso. "A systematic literature review of the teleo-reactive paradigm." *Artificial Intelligence Review* 42.4 (2014): 945-964.

[18] Katz, Edward P. "Extending the Teleo-Reactive paradigm for robotic agent task control using Zadehan (Fuzzy) logic." *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*. IEEE, 1997.

[19] Kochenderfer, Mykel J. "Evolving hierarchical and recursive teleo-reactive programs through genetic programming." *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 2003.

[20] Benson, Scott Sherwood. *Learning action models for reactive autonomous agents*. Diss. stanford university, 1996

[21] Spl.robocup.org. (2017). *Standard Platform League Results 2016 – RoboCup Standard Platform League.* [online] Available at: http://spl.robocup.org/history/results-2016/ [Accessed 14 Nov. 2017].

[22] Spl.robocup.org. (2017). *Standard Platform League Results 2017 – RoboCup Standard Platform League.* [online] Available at: http://spl.robocup.org/results-2017/ [Accessed 14 Nov. 2017].

[23] Lötzsch, Martin, et al. "Designing agent behavior with the extensible agent behavior specification language XABSL." *Robot Soccer World Cup*. Springer, Berlin, Heidelberg, 2003.

[24] Loetzsch, Martin, Max Risler, and Matthias Jungel. "XABSL-a pragmatic approach to behavior engineering." *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE, 2006.

[25] Erlang.org. (2017). *Erlang Programming Language*. [online] Available at: http://www.erlang.org/ [Accessed 17 Nov. 2017].

[26] Harel, David, et al. "Towards behavioral programming in distributed architectures." *Science of Computer Programming*98 (2015): 233-267.

[27] Vlissides, John, et al. "Design patterns: Elements of reusable object-oriented software." *Reading: Addison-Wesley* 49.120 (1995): 11.

[28] Ellson, John, et al. "Graphviz—open source graph drawing tools." *International Symposium*

on Graph Drawing. Springer, Berlin, Heidelberg, 2001.

[29] “GCC, The GNU Compiler Collection – GNU Project – Free Software Foundation (FSF)”. *Gcc.Gnu.Org*, 2018 [online], https://gcc.gnu.org/. Accessed 20 May 2018.

[30] Garber, Lee. "Robot OS: A new day for robot design." *Computer* 46.12 (2013): 16-20.

Appendix 1

A1.1 Tokens RegEx

```
INTLITERAL        -> digit (digit)*
BOOLEANLITERAL    -> true | false
DOUBLELITERAL     -> digit* fraction exponent? | digit+ . | digit+ .? exponent
HEXLITERAL        -> 0x(digit | [a-f])+
IMPORTLITERAL     -> ID\.tr
STRINGLITERAL     -> \'.*\' | \".*\"
ID                -> letter (letter | digit)*
letter            -> [A-Za-z_]
digit             -> [0-9]
fraction          -> . digit+
exponent          -> (E | e) (+|-)? digit+
keywords          -> or | and | not | if | elif | else | achieves | import |
exit | blackboard | init | once
frame-types       -> VAR | STACK | MAP | LIST
tr-types          -> INT | DOUBLE | BOOL | STRING | VECTOR
add               -> +
subtract          -> -
multiply          -> *
divide            -> /
less-than         -> <
less-than-eq      -> <=
greater-than      -> >
greater-than-eq   -> >=
EQEQ              -> ==
NOTEQ             -> !=
EQ                -> =
seperators        -> ; | , | arrow | : | block | { | } | ( | ) | .
arrow             -> '->'
block             -> ':-'
```

A1.2 Grammar EBNF

```
// .tr EBNF
tr-program       -> tr-prog-list
tr-prog-list     -> END | blackboard-decl tr-prog-list | var-decl tr-prog-list
| func-decl tr-prog-list | tr-node tr-prog-list | once-decl tr-prog-list

// imports - handled by the scanner, not in literal parser
imports          -> 'import' IMPORTLITERAL

// blackboard definition
blackboard-decl -> 'init' identifier frame-type tr-type
struct-type      -> 'VAR' | 'STACK' | 'MAP' | 'LIST'
tr-type          -> 'INT' | 'DOUBLE' | 'BOOL' | 'STRING' | 'VECTOR'

// variable definition
var-decl         -> identifier '=' initialiser
initialiser      -> expr

// function definition
func-decl        -> identifier '(' para-list-opt ')' '=' stmt
stmt             -> logic-stmt | return-stmt
logic-stmt       -> if-stmt
if-stmt          -> 'if' '(' expr ')' '->' stmt otherwise-stmt
otherwise-stmt   -> else-stmt | elif-stmt otherwise-stmt
elif-stmt        -> 'elif' '(' expr ')' '->' stmt
else-stmt        -> 'else' '->' stmt
return-stmt      -> expr ';'

// TR node
tr-node          -> identifier '(' para-list-opt ')' ':-' cond-stmt
cond-stmt        -> 'true' action-stmt | cond cond-stmt
cond             -> identifier ':' expr action-stmt
action-stmt      -> '->' action-list regr-stmt ';'
regr-stmt        -> 'achieves' identifier
action-list      -> action | action ',' action-list
action           -> expr-stmt | assign-stmt | exit-stmt

// once definition
once-decl        -> "once" "{" action-list "}"

// general components
identifier       -> 'ID'

para-list-opt    -> para-list | e
para-list        -> para-decl | para-decl ',' para-list
para-decl        -> identifier
arg-list-opt     -> arg-list | e
arg-list         -> arg | arg ',' arg-list
arg              -> expr

expr-stmt        -> expr
assign-stmt      -> identifier "=" expr
exit-stmt        -> 'exit' '(' BOOLEANLITERAL ')' | 'exit' '(' BOOLEANLITERAL
',' STRINGLITERAL ')'
exit-stmt        -> 'exit' '(' expr ')' | 'exit' '(' expr ',' expr ')'

expr             -> or-expr
or-expr          -> and-expr | or-expr 'or' and-expr
```

A1.3 TR-Lang C++ API Constructor Switches

```
Options:

Generic Options:
  -h [ --help ]                      Display help message
  -e [ --starting-node ] arg (=main) TR-Node name used as entrance to 
program.
                                     Default is 'main'
  -n [ --non-default-print ]         Turns off default printing function for 
TR
                                     Lang.
                                     User required to supply own printing
                                     function.
  -T [ --intern-throw ]              Internalize error and exit throws to the
                                     driver

Tools Options:
  -t [ --draw-tree ]                    Draws the structure of the TR tree
                                        implied by TR code
  -l [ --execution-profiler ] [=arg(=0)]
                                        Profiles execution time and number of
                                        executions for TR code.
                                        Optionally, a number for the yield 
rate
                                        of the profiler can be given
                                        Default is '0', meaning only yields
                                        after final execution
  -X [ --tr-debugger ]                  Activates TR-Debugger output which
                                        provides information for debugging

Debug Options:
  -S [ --scanner-trace ]     Allows debugging trace for scanner
  -P [ --parser-trace ]      Allows debugging trace for parser
  -D [ --display-ast-tree ]  Prints the AST tree after parsing.
                             Warning: for large files output will be verbose
  -W [ --warnings ]          Turns on additional warnings
  -s [ --silent ]            Allows silencing of interpreter output
  -v [ --verbose ]           Allows verbose output of interpreter
  -x [ --debug ]             Allows debug output of interpreter
```

Appendix 2

A2.1 Example Profiler Output

```
Source code | Num Executions | Clock Ticks | Seconds | Avg Time Per Execution
-------------Profiler Out 1-------------
main() :- | 100 | 11403 | 0.011403 | 0.00011403
check_count : myCount > 2500 -> exit(1) achieves main; | 0 | 0 | 0 | -nan
exit(1) | 0 | 0 | 0 | -nan
true -> print('hello world, i'm ' + myCount), myCount = myCount + 1 achieves
main; | 100 | 7967 | 0.007967 | 7.967e-05
print('hello world, i'm ' + myCount) | 100 | 5057 | 0.005057 | 5.057e-05
print('hello world, i'm ' + myCount) | 100 | 4961 | 0.004961 | 4.961e-05
myCount = myCount + 1 | 100 | 1753 | 0.001753 | 1.753e-05
-------------Profiler Out 2-------------
main() :- | 200 | 23033 | 0.023033 | 0.000115165
check_count : myCount > 2500 -> exit(1) achieves main; | 0 | 0 | 0 | -nan
exit(1) | 0 | 0 | 0 | -nan
true -> print('hello world, i'm ' + myCount), myCount = myCount + 1 achieves
main; | 200 | 16024 | 0.016024 | 8.012e-05
print('hello world, i'm ' + myCount) | 200 | 10114 | 0.010114 | 5.057e-05
print('hello world, i'm ' + myCount) | 200 | 9908 | 0.009908 | 4.954e-05
myCount = myCount + 1 | 200 | 3593 | 0.003593 | 1.7965e-05
-------------Profiler Out 3-------------
main() :- | 300 | 35357 | 0.035357 | 0.000117857
check_count : myCount > 2500 -> exit(1) achieves main; | 0 | 0 | 0 | -nan
exit(1) | 0 | 0 | 0 | -nan
true -> print('hello world, i'm ' + myCount), myCount = myCount + 1 achieves
main; | 300 | 25352 | 0.025352 | 8.45067e-05
print('hello world, i'm ' + myCount) | 300 | 15098 | 0.015098 | 5.03267e-05
print('hello world, i'm ' + myCount) | 300 | 14809 | 0.014809 | 4.93633e-05
myCount = myCount + 1 | 300 | 7004 | 0.007004 | 2.33467e-05
-------------Profiler Out 4-------------
main() :- | 400 | 50347 | 0.050347 | 0.000125868
check_count : myCount > 2500 -> exit(1) achieves main; | 0 | 0 | 0 | -nan
exit(1) | 0 | 0 | 0 | -nan
true -> print('hello world, i'm ' + myCount), myCount = myCount + 1 achieves
main; | 400 | 35670 | 0.03567 | 8.9175e-05
print('hello world, i'm ' + myCount) | 400 | 21174 | 0.021174 | 5.2935e-05
print('hello world, i'm ' + myCount) | 400 | 20799 | 0.020799 | 5.19975e-05
myCount = myCount + 1 | 400 | 9609 | 0.009609 | 2.40225e-05
-------------Profiler Out 5-------------
main() :- | 500 | 63234 | 0.063234 | 0.000126468
check_count : myCount > 2500 -> exit(1) achieves main; | 0 | 0 | 0 | -nan
exit(1) | 0 | 0 | 0 | -nan
true -> print('hello world, i'm ' + myCount), myCount = myCount + 1 achieves
main; | 500 | 43455 | 0.043455 | 8.691e-05
print('hello world, i'm ' + myCount) | 500 | 25668 | 0.025668 | 5.1336e-05
print('hello world, i'm ' + myCount) | 500 | 25208 | 0.025208 | 5.0416e-05
myCount = myCount + 1 | 500 | 11979 | 0.011979 | 2.3958e-05
-------------Profiler Out 6-------------
main() :- | 600 | 77515 | 0.077515 | 0.000129192
check_count : myCount > 2500 -> exit(1) achieves main; | 0 | 0 | 0 | -nan
exit(1) | 0 | 0 | 0 | -nan
true -> print('hello world, i'm ' + myCount), myCount = myCount + 1 achieves
main; | 600 | 54471 | 0.054471 | 9.0785e-05
print('hello world, i'm ' + myCount) | 600 | 33708 | 0.033708 | 5.618e-05
print('hello world, i'm ' + myCount) | 600 | 33143 | 0.033143 | 5.52383e-05
myCount = myCount + 1 | 600 | 13903 | 0.013903 | 2.31717e-05
-------------Profiler Out 7-------------
main() :- | 700 | 91175 | 0.091175 | 0.00013025
```

A2.2 Example Profiler Output TR-Lang Source Code

```
main() :-
check_count : myCount > 2500 -> exit(true) achieves main;
true -> print("hello world, i'm " + myCount), myCount = myCount + 1 achieves
main;
```

A2.3 Example Drawer TR-Lang Source Code

```
init mystack VAR INT

f() = print("");

a() :- true -> f(), exit(false) achieves a;

b() :-
    b1 : true -> f() achieves b;
    true -> print("") achieves b;

c() :-
    true -> blackboard.trmodule.mystack.insert(1) achieves c;

split() :-
    c1 : true -> a() achieves split;
    c2 : false -> b() achieves split;
    true -> b(), c() achieves split;

main() :-
    c1 : true -> a(), c() achieves main;
    c2 : false -> main() achieves main;
    true -> split() achieves main;
```

A2.4 Example Debugger Output

```
ncycle=0
vstate=in%@alpha%@double%@0.000000%@TR%@0%@
vstate=in%@myCount%@int%@0%@TR%@0%@
vstate=pre%@a%@null%@noval%@TR%@0%@a = 1
vstate=post%@a%@int%@1%@TR%@0%@a = 1
nscope=pre%@main%@TRD%@1%@main() :-
paramv=pre%@main%@main() :-
vstate=pre%@myCount%@int%@0%@TR%@1%@myCount = 1
vstate=post%@myCount%@int%@1%@TR%@1%@myCount = 1
fret=pre%@print%@void%@noval%@BI%@print('this rarely happens')
fret=post%@print%@void%@noval%@BI%@print('this rarely happens')
cond=post%@falsecond%@main%@falsecond : myCount == 0 -> myCount = 1,
print('this rarely happens') achieves main;
nscope=post%@main%@TRD%@0%@main() :-
ncycle=1
vstate=in%@alpha%@double%@0.000000%@TR%@0%@
vstate=in%@myCount%@int%@1%@TR%@0%@
vstate=pre%@a%@int%@1%@TR%@0%@a = 1
vstate=post%@a%@int%@1%@TR%@0%@a = 1
nscope=pre%@main%@TRD%@1%@main() :-
paramv=pre%@main%@main() :-
vstate=pre%@a%@int%@1%@TR%@1%@a = 2
vstate=post%@a%@int%@2%@TR%@1%@a = 2
fret=pre%@f%@null%@noval%@TR%@f('hello')
nscope=pre%@f%@TRD%@2%@f(word) :-
paramv=pre%@f%@word%@string%@'hello'%@f(word) :-
fret=pre%@print%@void%@noval%@BI%@print(word)
fret=post%@print%@void%@noval%@BI%@print(word)
cond=post%@true%@f%@true -> print(word) achieves f;
nscope=post%@f%@TRD%@1%@f(word) :-
fret=post%@f%@void%@noval%@TR%@f('hello')
fret=pre%@z%@null%@noval%@TR%@z(1, 2)
nscope=pre%@z%@FD%@2%@z(a, b) =
paramv=pre%@z%@a%@int%@1%@b%@int%@2%@z(a, b) =
fret=pre%@print%@void%@noval%@BI%@print('none')
fret=post%@print%@void%@noval%@BI%@print('none')
nscope=post%@z%@FD%@1%@z(a, b) =
fret=post%@z%@void%@noval%@TR%@z(1, 2)
fret=pre%@print%@void%@noval%@BI%@print(m())
fret=pre%@m%@null%@noval%@TR%@m()
nscope=pre%@m%@FD%@2%@m() =
paramv=pre%@m%@m() =
nscope=post%@m%@FD%@1%@m() =
fret=post%@m%@string%@'word'%@TR%@m()
fret=post%@print%@void%@noval%@BI%@print(m())
exitc=post%@1%@test success%@exit(1, 'test success')
```

A2.5 Built-in Function Call Test Code

```
main() :- true -> getVal() achieves main;
```

A2.6 Built-in Variable Expression Test Code

```
main() :- true -> alpha achieves main;
```

A2.7 Built-in Variable Assignment Test Code

```
main() :- true -> alpha = 2.5 achieves main;
```

A2.8 Blackboard Call Test Code

```
init myvar VAR INT

main() :- true -> blackboard.trmodule.myvar.insert(1) achieves main;
```

A2.9 Binary Operation Test Code

```
var = 1

main() :- true -> 1 + var achieves main;
```

A2.10 Unary Operation Test Code

```
var = 1

main() :- true -> -var achieves main;
```

A2.11 Function Call Test Code

```
myfunc(a) = a;

main() :- true -> myfunc(12) achieves main;
```

A2.12 TR-node Call Test Code

```
mynode() :- true -> 1 achieves mynode;

main() :- true -> mynode() achieves main;
```

A2.13 Variable Expression Test Code

```
var = 1

main() :- true -> var achieves main;
```

A2.14 Variable Assignment Test Code

```
var = 1

main() :- true -> var = 2 achieves main;
```