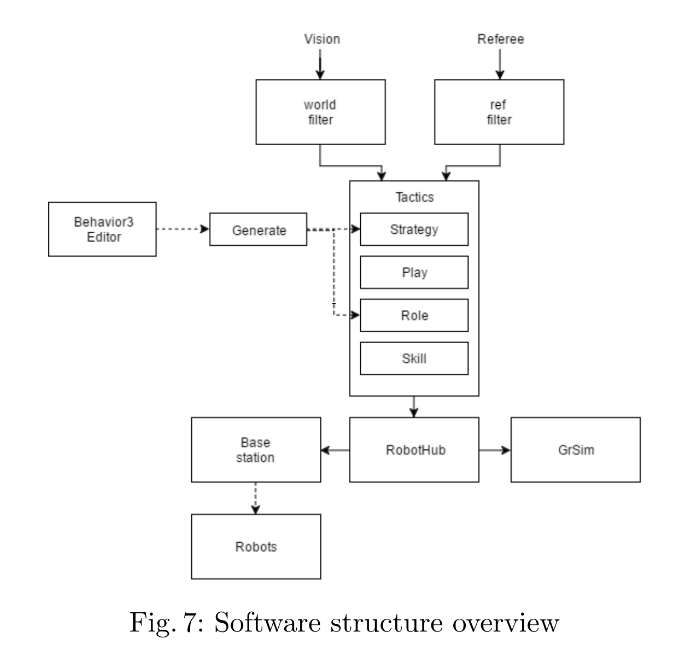
*Most of the text in this document (up to the Installation Guide) is copied from the Team Description Paper for RoboCup 2017. Additions are written in italics.*

**General Overview**

The software structure consists of several blocks, each representing a node which runs in a separate process. These nodes are based on and interconnected by the Robotic Operating System (ROS), which will be covered later in this section. All the solid arrows represent ROS communication lines over which the different nodes can post and receive data. At the top of the diagram, the incoming vision and referee data are filtered and sent to the tactics block, where this data is kept in memory until the next vision or referee update. The tactics block itself is separated into four layers. Once the tactics module has decided what every robot needs to do, it sends these commands to the RobotHub. This component is the interface between the tactics PC and the individual robots, managing the communication between them and performing all low-level transformations of data. Additionally, the RobotHub can be configured to send data to the GrSim simulation program instead, which will then feed data to the vision system. To the tactics module, these two configurations look exactly the same, giving the ability to test the software in the simulation the same way it would run in the full system.

****

*The software is written primarily in C++, with some Python for the debugging tools. It is designed to run on Ubuntu 16.04, and may not work anywhere else. An installation guide is available at the end of this document.*

**ROS**

As mentioned before, the software makes use of ROS, a flexible framework for writing robot software. ([*http://www.ros.org*](http://www.ros.org)) It is a collection of tools, libraries and conventions that are meant to simplify the task of creating robust software for different robotic applications. ROS is primarily used because of its built-in communication infrastructure. Within ROS, messages can be sent between distributed nodes via an asynchronous and anonymous publish-subscribe mechanism. Messages are sent over 'topics' with a fixed message type, thereby enforcing clear interfaces between nodes. The main advantage of using this system is that different software modules are detached from one another, and can easily be adapted or replaced independently. ROS is housed in a ROS Master process, which acts as a central communication broker. The Master keeps track of active nodes and subscriptions and provides subscribers with their respective publishers' contact information, so that these two nodes can then communicate independently. The Master also runs a parameter server, which stores and provides access to shared variables. The actual messages are sent in a binary format (including a checksum) using the XMLRPC protocol. The transport layer used is determined at runtime, but usually it is TCPROS, a variant of TCP.

Communication via topics is very effective when the message type, as well as the sender and receivers, are fixed. However, because skills are fixed functions but their use within roles can vary, this is not an appropriate infrastructure for communication between skills (e.g. for synchronization). For this type of communication, parameters in the ROS parameter server are used. Global parameters can be set by any node and read at any time by any other node. So when one robot reaches a certain state, this can be communicated to other robots that depend on the first robot by changing a ROS parameter. The parameter server does not offer any atomicity guarantees, but at this time we do not use it in a way which could cause a race condition. Should that change, we will have to implement a synchronization mechanism to act as a safeguard.

A key advantage of enforcing clear interfaces between nodes (as ROS does) is that nodes become interchangeable; as long as two nodes expose the same interface (by publishing and subscribing to the same set of topics and advertising the same services) they can occupy the same 'slot' in our infrastructure. *Descriptions of the interfaces for every node are provided in the nodes' individual documentation.* An example of this is the way in which robot commands are generated: Normally, our AI node would perform this task, but for testing we can choose to run a keyboard or gamepad controller instead. These nodes all publish on the '/robotcommands' topic, and to any subscribers to that topic, the messages look exactly the same no matter where they originate from.

Another useful function of ROS is that all messages between nodes can be easily recorded and played back by using the 'rosbag' tool, which is distributed as a part of the ROS suite. Rosbag keeps a record of all messages sent, and allows the user to play them back in real time at a later date. This means that the separate software modules can also be tested separately. One software module can be run and tested on its own, even if it depends on messages from other modules, by playing back recorded communication from an earlier session. This feature gives the additional advantage that the pre-recorded messages are well-known and unchanging over multiple tests, simplifying the debugging process.

**Behaviour Trees**

The use of behaviour trees enables the team to speed up the process of creating and debugging strategies. Because behaviour trees are easily constructed, one can quickly put together a new strategy to see how it performs. And because of the visual nature of behaviour trees, it is expected that bugs are found more easily than in regular code.

This simplicity of behaviour trees also gives the ability to involve people outside of the team in discussions about strategy. Even people who are inexperienced in computer programming, but well-versed in football strategy, can understand and contribute to behaviour trees. At the moment the team is working on collaboration with human soccer teams. It is envisioned that if the process of creating and testing behaviour trees is made simple anIn several layers within the strategic architecture, behaviour trees are used in order to determine how the robots should behave. Before explaining the strategy and tactics some information about these trees is given. Simply put, a behaviour tree is a hierarchically structured set of actions, combined with conditions that specify which branch of the tree should be executed.

d fun, many people will enjoy collaborating with the team to contribute to the soccer strategies.

The formal version of behaviour trees that we use is largely based on the work done by Marzinotto *et al,* May 2014, "Towards a unified behaviour trees framework for robot control.", in Robotics and Automation, (ICRA), 2014 IEEE International Conference on Robotics and Automation (pp. 5420-5427).

The described formal and general definition of behaviour trees provides us with a robust platform for structuring our AI. Furthermore, a customized internal fork of the behaviour3 Editor is used for editing the behaviour trees. Combined with custom tooling and interfaces in ROS and the rest of our tactics module we have already used behaviour trees for various tasks, including debugging our robots and designing our qualification performance.

*The building blocks of behaviour trees are as follows:*

* *The root node is the starting point of execution. It does nothing, it's just an anchor point.*
* *All nodes return a result, which is one of these: (bt::Node::Status)*
  + *Success: The operation/check succeeded.*
  + *Failure: The operation/check failed.*
  + *Running: The operation/check is in progress and will continue on the next iteration.*
  + *Invalid: An error occurred and the node cannot produce a result.*
* *Composites and Decorators make up the internal nodes of the tree. Composites can have more than one child, decorators must have exactly one.*
  + *Composites:*
    - *Selector (?): Executes its children in order until one of them succeeds. It starts at the first child every iteration. It succeeds when a child succeeds, and fails when all children fail. Equivalent to a lazy logical OR operation.*
    - *MemSelector (?\*): Like Selector, but resumes where it left off last iteration. So if a node returns running, that node will be updated first in the next iteration.*
    - *Sequence (->): Executes its children in order until one of them fails. Like Selector, it starts at the first child every iteration. Equivalent to a lazy logical AND operation.*
    - *MemSequence (->\*): As MemSelector is to Selector.*
    - *ParallelSequence: Executes all of its children, then evaluates their results. Depending on arguments, it can fail or succeed when a certain number (or all) of its children do so.*
  + *Decorators:*
    - *Failer: Executes the child node, then returns failure regardless of what the child returned.*
    - *Succeeder: Take a good guess.*
    - *Inverter: Executes the child node. If the child returns success, the inverter returns failure and vice versa. If the child returns running or invalid, the inverter does the same.*
    - *Repeater: Executes the child a number of times. A limit may be given as an argument. If no limit is provided, then the child will be executed forever (well, 2^32 times).*
    - *UntilFail / UntilSuccess: Executes the child until it fails/succeeds. A limit on the number of iterations may be given as an argument. If the child returns invalid, then this decorator returns failure.*
* *Conditions and Skills are at the leafs of the tree.*
  + *Conditions perform some check of the world state, and should adhere to these rules:*
    - *They should never return running.*
    - *They should never affect the world state; the checks must be passive.*
    - *Their return values should indicate the result of the check, not whether the check itself succeeded or not.*
  + *Skills send commands to the robot executing the tree.*
    - *Skills must never call ros::spin or ros::spinOnce. If a skill requires feedback from another node, it must check for the presence of this feedback in a following iteration.*
    - *Skills should always return in a timely manner. If they need to do expensive operations, these should be spread over multiple iterations.*
    - *Skills can call other skills internally, but you should try to handle this mostly in the tree structure.*

**Strategic Architecture**

The key challenge of the RoboCup Small Size League is intelligent coordination of a multi-robot system while dealing with rapidly changing circumstances and an unpredictable opponent. In order to ensure that all the robots always act according to the collective goal of scoring more goals than the opponent, a multi-level strategic planner has been developed. Just like many other SSL-teams an approach based on different levels of abstraction is used as described in Browning *et al*.

On the lowest level a number of skills are defined. Skills are functions that transform specific, short-term goals for a single robot to low-level robot instructions. One level higher roles are used. A role is assigned to a single robot and determines which skill to execute with what parameters. Roles are generated from behaviour trees. The next level is the plays layer. A play directs an arbitrary number of roles. When initializing the roles, it can decide on what parameters the role should use. And while the roles are running, the play keeps the overview and decides when to terminate the roles. On the highest level is the strategy. Only one strategy is running during any game. The strategy is generated from a behaviour tree, just like a role, and determines which plays to execute based on the state of the game. The strategy also takes into account referee commands, and chooses the appropriate plays when new commands are issued.

The design of the software structure is based on requirements considered important, like flexibility and simplicity. The system developed is especially flexible, because a play can control any number of robots, rather than a fixed number. On top of that, behaviour trees increase flexibility as discussed before.

**Skills**

The most low-level nodes in the behaviour trees are skills and conditions. Skills are functions that transform specific, short-term goals for a single robot, such as 'get the ball' or 'kick the ball', to velocity commands. This abstraction level allows for very concise and clear behaviour trees in the higher levels of the strategic architecture, because most of the logic and computation is done within the skill.

Skills are functions that take certain input arguments, based on the task they are designed to perform, and output a velocity command for a single robot. Besides this simple functionality, however, skills also have access to the complete current state of the game, hence their output can be (partly) based on events currently happening in the field. Because skills are updated as soon as new information about the world state becomes available, it allows for quick responses to changing and unpredictable circumstances.

An example of one of the skills is the *GoToPos* skill, an advanced position controller, which takes as input argument a target position and orientation, and computes a velocity that moves the robot towards its goal. In addition, the *GoToPos* skills can take into account the position of all other robots in the field, and adapt the computed velocity such that it does not collide with them.

Another example of a skill is the *ReceiveBall* skill, which takes as input argument a position where a robot should wait to receive the ball. Then it computes the exact position where the robot can receive the ball, based on the current position and velocity of the ball, but also on whether another robot in the game is currently exerting influence on the ball and is expected to change its velocity.

Skills can internally call other skills and conditions, which allows for more complex behaviour. Although this opens countless possibilities for skills, it is important to note that skills remain focused actions executed to achieve a specific, short-term goal. A situation in which skills can be combined is in the *GetBall* skill, which internally calls *GoToPos* in order to steer a robot towards the ball.

**Conditions**

Conditions can be used in behaviour trees to determine which play, skill, or set of skills should be executed based on the state of the game. Conditions can be used in the strategy behaviour as well as in the role behaviour trees. A condition is a function that returns either true or false, based on which a certain branch of a tree can be chosen. Examples of conditions that are used are *IHaveBall*, *BallOnOurSide* and *IsRefereeState*.

**Roles**

A role is a behaviour tree that is built up of skills and conditions and directs one robot. Unlike skills, the functionality of a role is quite simple. A role is basically a sequence of skills, combined with instructions on when to execute them. Combined with the behaviour tree format this makes it very easy to construct and adapt roles. This means that once there is a sufficiently large base of skills it is very easy to quickly develop and test completely different strategies. It is also useful when working together with people who are very experienced in football strategy but not so much in computer programming.

**Plays**

*Plays used to be called Tactics, and are often still referred to as such in code.*

A play directs an arbitrary number of roles that work together to reach a collective goal. When a play is initialized, it starts the relevant roles and chooses parameters for these roles based on the current state of the game. Every time the play is subsequently called, it checks whether the conditions for executing this play are still present, and whether the parameters given to the roles are still optimal. If they are not, it either quits executing or adapts the role parameters.

**Strategies**

The strategy is the highest level of abstraction within the strategic architecture. Only one strategy can be active at a time. A strategy chooses which plays should be executed. It does this partly based on the state of the game. The strategy also keeps track of referee commands, and only chooses plays that comply with the rules applicable to the current state of the game. A functionality that will be looked into in the future is a strategy which keeps track of all executed plays in a game and takes this information into account when choosing new plays in the rest of game.

***StrategyComposer***

*To avoid having to deal with a single, huge behaviour tree and to gain flexibility, we use StrategyComposer. This class contains a list of partial strategies, one for every state the referee may indicate, and one more for general play. The composer creates an artificial node (a* RefStateSwitch*) which looks at the current referee state, and picks the corresponding strategy branch to execute.*

***Control Flow***

*A single StrategyNode (an executable in the roboteam\_tactics repo) executes the strategy tree.*

*As described above, a RefStateSwitch is at the root of the strategy tree. The strategy will pick one or more (in the case of a ParallelSequence) plays to execute. These plays create a number of roboteam\_msgs::RoleDirective messages, which contain a robot ID, the name of the role the robot is to execute, and all parameters it needs. For every robot in the field, a RoleNode will run, which receives such a RoleDirective. These RoleNodes are all separate processes, and so the roles execute in parallel. Communication between nodes is provided through the ParamSet skill and the ParamCheck condition. These use the ROS parameter server to store and retrieve values. It is up to the programmer to decide when roles and tactics should end. Keep in mind that the RefStateSwitch may start executing a different branch of the strategy tree if needed, so you cannot assume that your play/role will always get another iteration.*

*All nodes have three main member functions: Initialize, Update and Terminate.*

*Initialize should perform all initialization the node needs before executing. Don't do this in the constructor, as the node may be reused in a later iteration. For plays, the Initialize function should send the RoleDirectives.*

*Update does the actual processing and returns the result.*

*Terminate should clean up any resources the node holds and whatever else the node needs to exit cleanly. It is not necessary to reset variables to initial values, that will be done in Initialize if the node is reused.*

**Installation Guide**

First of all, make sure you are running Ubuntu. We use version 16.04. Other versions or distros may work, but we offer no guarantees.

Follow the official ROS instructions for installing ROS and its dependencies: <http://wiki.ros.org/kinetic/Installation/Ubuntu>

Now create a folder which will serve as your workspace. Go to that folder and do this:

<http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment>

You should now have a valid but empty catkin workspace. Go to the src folder you created, and execute this command:

$ git clone <https://github.com/RoboteamTwente/roboteam_sync.git>

You will also need some dependencies:

$ sudo apt install python3 python3-pip

$ sudo pip3 install gitpython pygithub

$ cp roboteam\_sync/rts.py ..

$ cd ..

$ ./rts.py get

Note : after "./rts.py get", run:

"rm -rf src/roboteam\_steering"

"rm -rf src/roboteam\_tactics"

These repos are old and will give errors on "catkin\_make"

Enter usernames and passwords when requested. This tool will get all GitHub repos in the RoboTeam Twente organization, as well as the private roboteam\_tactics repository from BitBucket. You should read the documentation of rts.py, it can do many more useful things.

At this point you should have all of the software downloaded in your src directory. Before building, you first need to install a couple more libraries:

$ sudo apt install libsdl2-2.0-0 libsdl2-dev libqt4-dev qt5-default libboost-all-dev ros-kinetic-uuid-msgs ros-kinetic-joy protobuf-c-compiler protobuf-compiler

For using the rqt plugins these dependencies are needed:

$ sudo apt install python-subprocess32 python-protobuf

If you want to run SSL Vision with the Basler cameras installed above the field, you will also need to install the Pylon API. This can be found in the Drive: Software/Cameras/pylon-5.0.5.9000-x86\_64.zip

Note : Check if you have a 64 bit Intel processor

That archive contains a file with instructions.

Finally, you should now be able to run:

$ catkin\_make

Probably you should run it a second time if the first time fails

Now we add the roboteam\_tactics folder back:

"./rts.py get"

"rm -rf src/roboteam\_steering"

Note : if you encounter an error with "unique\_id", run

"sudo apt install ros-kinetic-unique-id"

"uuid" ->

"sudo apt install ros-kinetic-uuid-msgs"

More errors? Run "sudo apt-cache search [yourpackagehere]" and install it

Constant file error? $ source devel/setup.bash

Note : if there is an error with "PROTOBUF\_PROTOC\_EXECUTABLE", run:

"sudo apt install protobuf-compiler"

"which protoc" (remember this. Probably returns "/usr/bin/protoc")

open the file "/usr/share/cmake-3.5/Modules/FindProtobuf.cmake" with sudo rights. Go to the bottom of the file and look for

# Find the protoc Executable

find\_program(PROTOBUF\_PROTOC\_EXECUTABLE

NAMES protoc

DOC "The Google Protocol Buffers Compiler"

PATHS

${PROTOBUF\_SRC\_ROOT\_FOLDER}/vsprojects/${\_PROTOBUF\_ARCH\_DIR}Release

${PROTOBUF\_SRC\_ROOT\_FOLDER}/vsprojects/${\_PROTOBUF\_ARCH\_DIR}Debug

)

Before the ")", add the line returned by "which protoc", except for the "/protoc" part at the end. It should now look something like this:

# Find the protoc Executable

find\_program(PROTOBUF\_PROTOC\_EXECUTABLE

NAMES protoc

DOC "The Google Protocol Buffers Compiler"

PATHS

${PROTOBUF\_SRC\_ROOT\_FOLDER}/vsprojects/${\_PROTOBUF\_ARCH\_DIR}Release

${PROTOBUF\_SRC\_ROOT\_FOLDER}/vsprojects/${\_PROTOBUF\_ARCH\_DIR}Debug

/usr/bin

)

Save the file, and try catkin\_make again

Note : if permission errors, run :

*cd src/roboteam\_tactics/scripts*

*sudo chmod 766 ./\**

This will take a while the first time. If it fails, try running

$ source ./devel/setup.bash && catkin\_make

If that also doesn't work, you'll have to figure out the problem yourself.

Once everything compiles, you can get to work making things even better. Try not to break stuff.

If you want to run RobotHub:

Make sure ModemManager has been eradicated from

your Ubuntu installation. otherwise, upon serial

connection, it will dump about 30 characters of random bytes

into the connection, screwing up the connection.

ModemManager is a program/service of some sort.

<http://askubuntu.com/questions/216114/how-can> z6 -i-remove-modem-manager-from-boot

*sudo apt-get purge modemmanager* + reboot or something should do the trick

Also very important: add the user to the dialout group

*sudo adduser [your\_username] dialout*

Add the code from Appendix 1 to the file "~/.bashrc". This automatically runs the command *source devel/setup.bash* every time you start up a new terminal. It also gives you the functions *rtt\_build* and *rtt\_nuke*, which replace catkin\_make.

Driving the robot:

"source devel/setup.bash"

"roslaunch roboteam\_tactics RTTCore\_serial.launch"

"rosrun roboteam\_input keyboard\_controller"

**Robot Types**

Robots may be controlled differently depending on their "type". The possible types are: proto, arduino, and grsim. The "proto" type is the latest robot of the RoboTeam Twente, and is the default type. The "arduino" type is used for an early prototype version, which uses an arduino and brushed motors for easy testing. The "grsim" type is used when controlling robots in the GRSim simulation environment.

Robot types can be set using global ROS params. For each robot, there is a parameter "robotType" which can be set to either "proto", "arduino", or "grsim" (for robot 3 this would be: robot3/robotType). When this parameter is not set, the robotType used for control will default to "proto".

**Usage of `rtt::get\_PARAM\_&parameter\_name&()`, `ros::param::get()` and `ros::param::getCached()`**

The rtt::get\_PARAM/rtt::set\_PARAM families of functions as well as the other auto-generated stuff in roboteam\_utils/constants.(h/cpp) is a semi-failed attempt to introduce type-safety and typo-safety in ROS parameters. These functions and constants are generated from the String Constants file at the root of the roboteam\_utils package.

ros::param::get and ros::param::set are functions provided by ROS which allow you to access the ROS parameter server. These functions take a parameter name and a variable to store the parameter in or get the value from. They are type-safe on the client side, but bad things will happen if the server has a parameter with that name, but a different type. So make sure you update and read the documentation of all nodes you communicate with. ros::param::getCached is an alternative to ros::param::get which stores the value in a local cache and creates a subscription on the parameter server to monitor changes. It's faster if the value changes less often than you need it, but if you only need the value only rarely it is probably better to just use ros::param::get.

# Appendix 1 - Roboteam "~/.bashrc"

Make sure that the rosconsole.config file exists

### Roboteam Start ###

export ROS\_LANG\_DISABLE=genlisp:geneus:gennodejs

source [PathToYourWorkspace]/devel/setup.bash

alias rts="./rts.py"

export ROSCONSOLE\_FORMAT='[${severity}][${logger}]: ${message}'

export ROSCONSOLE\_CONFIG\_FILE=[PathToYourWorkspace]/rosconsole.config

alias TestX="rosrun roboteam\_tactics TestX"

alias catkin\_nuke="catkin\_make clean && catkin\_make"

alias rosrung="rosrun --prefix \"gdb --args\""

function rtt\_build() {

(

while true; do

if [[ "$PWD" == "/" ]] ; then

break

fi

if [ ! -f ./.catkin\_workspace ]; then

cd ..

else

break

fi

done

if [ -f ./.catkin\_workspace ]; then

# Catkin ws!

eval $1

if [ $? -eq 0 ]; then

notify-send -u critical -i ~/catkin\_ws/rtt.png "Compilation message" "Compilation succeeded ✔"

else

notify-send -u critical -i ~/catkin\_ws/rtt.png "Compilation message" "Compilation failed ❌"

fi

else

printf "No catkin workspace discovered while going up\n"

fi

)

}

export HISTCONTROL=

function try\_rtt\_nuke() {

cmdHistory=$(history 2 | head -n 1)

#printf "$(history)"

#printf "\n\n:::HISTORY:::::\n\n$cmdHistory"

if [[ $cmdHistory == "rtt\_nuke" ]]; then

read -p "Are you sure? " -n 1 -r

echo # (optional) move to a new line

if [[ $REPLY =~ ^[Yy]$ ]]; then

# do dangerous stuff

rtt\_build "catkin\_make clean && catkin\_make"

#printf "REALLY done building"

fi

else

rtt\_build "catkin\_make clean && catkin\_make"

#printf "done building"

fi

}

alias rtt\_make="rtt\_build catkin\_make"

alias rtt\_nuke="try\_rtt\_nuke"

\_GetTestXOptions () {

local cur prev words cword

\_init\_completion || return

COMPREPLY=()

local words=$(TestX show all | grep -E ".?- .?$" | sed -E "s/.?- (.?)$/\1/" )

COMPREPLY=( $( compgen -W '${words}' -- "$cur") )

return 0

}

complete -F \_GetTestXOptions TestX

switchSides () {

side="$(rosparam get our\_side)"

if [ "$side" = "left" ]; then

rosparam set our\_side right

printf "Switched to right side"

elif [ "$side" = "right" ]; then

rosparam set our\_side left

printf "Switched to left side"

else

printf "Could not switch sides"

fi

printf "\n"

}

fortune | cowsay -f duck

### Roboteam Stop ###