Using the Director within the NUClear Framework

Ysobel Sims and Trent Houliston

University of Newcastle, Callaghan 2308, Australia

Abstract. The Director is a behaviour framework for reactive real-time robotic software architectures. We implement it within NUClear, a loosely coupled message-passing framework for robotics. The source code can be found at `github.com/NUbots/DirectorSoccer`. This technical report describes how this implementation can be used to create behaviours.

The Director [2] is a framework and algorithm used to control the flow of robotic behaviour. It allows composable behaviours that can take advantage of transition rules to ensure that the robot performs the right task at the right time. The Director does this by providing a set of new DSL keywords to NUClear [1] that describe what tasks to run, what conditions are required, and how those tasks build on other tasks. It then uses these keywords to make a graph that it will use to decide which tasks to execute in order to achieve the goals of the modules.

The two DSL components that are added are Providers and Tasks.

1 Providers

Providers are implemented as NUClear reactions and live in a NUClear 'on' statement. They are the components which 'provide' the functionality needed to perform a task, either by doing it directly or by running a series of subtasks. They form the basis of the modules that implement behaviours. Each Provider will provide for a specific message type in a specific situation. For example a Provider which executed a walk task, the code might look like:

```
on<Provide<Walk>>().then(...);
```

Providers are triggered whenever a Task needs to be executed by the Director, however they can also be triggered using other NUClear keywords. The Director will ensure that any Provider which is not currently active will not run if it is not active, even if the other triggers would make it run.

For example if the walk engine module wants to update at 10Hz, the 'Every' keyword can be used to achieve this:

```
on<Provide<Walk>, Every<10, Per<std::chrono::seconds>>().then(...);
```

This Provider would then run once every time the Walk Task was updated, as well as running at 10Hz while the Provider is active.

It is possible to run an inactive Provider with the 'Optional' keyword.

```
on<Optional<Provide<Walk>>, Trigger<Sensors>>().then(...);
```

The reaction will run regardless of if there is a Walk task if a Sensors message is emitted. The Walk Task will be empty, and the Provider cannot emit subtasks.

Subtasks must be emitted from the Provider reaction to be considered a child of that Provider. If the Task is emitted from a non-Provider reaction, it will be considered a root task.

1.1 Provider Groups

When there are multiple Providers which provide for a single Task type they are considered a Provider group. For example, both a static walk module and a zero moment point walk module could have 'Provide<Walk>' reactions. These two Providers would be considered a Provider group for the Walk task.

When a Provider group is running, only one of the Providers in that group is allowed to execute at a time.

When a new Task is emitted the Director will determine which of the Providers in a group to run based on the current state of the system as well as any extra DSL keywords that are on the Provider's declaration. If both Providers could run, then the one declared first will run.

If the state of the Director system changes such that the currently active Provider can no longer run, then the Director will attempt to reassign that task to another Provider in the group.

Provider groups operate together as a single unit. Any Task that is emitted from a Provider in a group is considered to be emitted from that group. Considering the walk example, both Providers may emit a LegIK Task. Regardless of which Provider is active, the LegIK Task will be considered as coming from the walk Provider group. If the walk Providers emit different subtasks, then the old subtasks will be considered 'finished' and removed from the graph, and the new subtasks will start running.

1.2 Provider Types

There are three different Provider types that can be used. These three provider types along with the 'Needs', 'When' and 'Causing' DSL words determine the flow of a Provider, executing code as the Director changes which Tasks it executes.

Start: A Start Provider is used to set up the state of a Provider when it starts running. Start is executed when a Provider group has no active Tasks and then is given one, i.e. it gains control.

For example, the walk may need a Start Provider to initialise the state of the walk engine before it starts running.

```
    on<Start<Walk>>().then( { // this is a Start Provider type for the Walk Task
        // Initialise walk
        delta = NUClear::clock::now();
        walk_engine.reset();
        current_orders = Eigen::Vector3d::Zero();
    });
```

Provide: Provide is the normal Provider which executes the functionality for a Task, as described previously. It will run after Start, if a Start Provider exists.

```
on<Provide<Walk>>().then( { // a Provide Provider type for the Walk Task
    // Run the walk
    auto goals = calculate_joint_goals();
    emit<Task>(std::make_unique<LegIK>(goals));
});
```

Stop: Stop Providers run when a Provider group is losing control and will no longer run. That is, when the Provider group has no active Task. This is useful for running any cleanup code.

A Stop Provider cannot run unless either a Start or Provide has run for that Provider group.

Any example of a Stop Provider is:

```
on<Stop<Walk>>().then({ // a Stop Provider type for the Walk Task
    // Log that the walk has stopped
    log<NUClear::TRACE>("Walk Stopped");
});
```

1.3 Needs

The Needs DSL word is used to ensure a Provider can take control of subtasks, otherwise it cannot run.

```
on<Provide<Walk>, Needs<LegIK>>().then(...);
```

In this example, the Walk Provider will only run if it can take over the LegIK Task.

If a higher priority Provider has already taken over the LegIK Task, then this Provider will not run. If the LegIK is not being used by any other Provider, or if this Provider has higher priority than other Providers wanting to control the LegIK Task, then this Provider will take over the LegIK Task.

Without 'Needs', the Provider will still run and can emit subtasks, but the subtasks will not do anything if another Provider has access to them.

1.4 When

A When condition is used to ensure that a Provider will only execute when the specified condition is met. These conditions are provided using a global enum message along with a statement that describes what condition you want. For example, if you wanted a kick engine to only run if the robot is standing then you would write:

```
on<Provide<Kick>,
When<Stability, std::equal_to, Stability::STANDING>>
().then(...);
```

These conditions are updated by using normal NUClear emits. For example if

```
emit(std::make_unique<Stability>(Stability::STANDING));
```

was emitted in another reaction, then the 'When' condition would be met and the Provider would run.

1.5 Causing

A Causing statement is a promise that by running this Provider, a specified desired state will be eventually reached. This is used to allow smooth transitions between modules. For example, when the walk engine stops moving it would be in a 'STANDING' Stability state. This means the walk engine could have a 'Provide' reaction that causes this state:

The implementation for the reaction should result in the desired state being eventually emitted. This would not be the primary Provider for the Walk, but would instead be active when a higher priority Provider wants this state.

For example, if the kick engine only runs

```
on<Provide<Walk>, Causing<Stability, Stability::STANDING>().then({
    // Make the robot stop walking
    set_orders = Eigen::Vector3d::Zero();
    // Check if we have stopped moving
    if (state == STOPPED) {
        // Emit that we are standing
        emit(std::make_unique<Stability>(Stability::STANDING));
    }
});
```

```
When<Stability, std::greater_equal, Stability::STANDING>
```

then it would not be able to run while the walk engine is running normally. However, if the kick Task has a higher priority than the walk Task, then it will push the walk engine into the Causing Provide reaction, which will then cause the Stability state to be 'STANDING' so that the kick engine can run.

2 Tasks

Tasks are the jobs that Providers execute. The are generally in the form of Protobuf messages, and may or may not contain data within them.

Tasks are implemented using NUClear's 'emit' along with a custom scope 'emit<Task>'.

```
// Emit a Task requesting to walk forwards
emit<Task>(std::make_unique<Walk>(Eigen::Vector3d(1.0, 0.0, 0.0)));
```

This walk engine Task contains information on the speed and direction that the robot should walk in. Other Tasks, such as 'GetUpWhenFallen' does not need any data.

Tasks can take four arguments, as described in Table 1.

Argument	Description
data	A pointer to a Protobuf message, describing the Task.
priority	A positive integer. Higher numbers indicate higher priority. If two Tasks have the same priority, then the one emitted first will be chosen. If no priority is specified, then the default priority of 0 is used.
optional	A boolean. If true, then the Task is considered optional. If two Tasks need the same resources and have the same priority but one is optional, then the non-optional Task will be chosen. Optional Tasks do not need to be active for other Tasks from the same Provider to run. If no optional argument is specified, then the default optional value of false is used.
name	A string. If provided, then this string will be used when logging messages. The default is the empty string.

Table 1: Arguments that can be used to define a Director Task in NUClear.

2.1 Root Level Tasks

When a Task is emitted from a reaction which is not itself a Provider, that Task is considered a root level task. These root level tasks are all considered siblings within the priority comparison system at the root level of the graph. When a root level task is emitted with a 'nullptr', that root level task is removed from the Director tree and the current list of Tasks to execute.

```
emit<Task>(std::unique_ptr<GetUpWhenFallen>(nullptr));
```

Note that you must use 'std::unique_ptr' not 'std::make_unique' when using 'nullptr'.

2.2 Subtasks

A subtask is a Task that is emitted by a Provider. When multiple Tasks are emitted by a Provider, each of these are sibling Tasks in the Director graph.

2.3 Priority

Priority in the Director is considered based on the closest common ancestor of the two competing Tasks. For root tasks, the closest common ancestor will be the root element. Once the closest common ancestor is determined, the priority of each Task's branch will determine which Task has higher priority. The winner takes control and becomes active, while the evicted Task will need to watch for an opportunity to take back control.

When a Task's ancestor tree has an optional Task between itself and the common ancestor, then the Task is considered optional. If one Task has an optional parentage and the other does not then the optional Task will automatically lose. If both have optional Tasks in their parentage, then the Tasks will be compared as before. A higher priority root Task's optional elements are more important than a lower priority Task's optional elements.

2.4 Done Tasks

A Done Task is a special Task that can be emitted by a Provider to signal to its parent that it has completed the Task that was assigned to it. The Provider group that created this task will then be re-executed with the knowledge that it was triggered as a done event from below. The Task will not be removed from the Director tree, unless it is a root Task. If it is a root Task, it will be removed as if it was emitted with a 'nullptr'.

```
emit<Task>(std::make_unique<Done>());
```

2.5 Idle Tasks

An Idle Task is a special Task that can be emitted by a Provider to signal to continue running the Tasks it previously requested. For example, if a Provider is running a walk Task, it can emit an Idle Task to continue running the walk Task.

```
emit<Task>(std::make_unique<Idle>());
```

3 Data

Providers can access data about the local state of the Director.

3.1 Uses

The 'Uses' DSL word is used to find out information about subtasks. It contains information on if the subtask is Done and what the current run state of the subtask is.

The 'done' field is a boolean that is true if the subtask is Done, regardless of if this Provider ran it. It is useful when the user wants to know if specific or multiple subtasks are Done.

```
on<Provides<Walk>, Uses<LegIK>>().then([this] (const Uses<LegIK>& leg_ik) {
    if (leg_ik.done) {
        log<NUClear::INFO>("Leg IK is done");
    }
});
```

The 'run_state' field is a 'RunState' enum that indicates the current run state of the subtask. It is useful when the user wants to know if the subtask is running, has been queued to run, or has not been emitted from this Provider.

```
on<Provides<Walk>, Uses<LegIK>>().then([this] (const Uses<LegIK>& leg_ik) {
    if (leg_ik.run_state == GroupInfo::RunState::RUNNING) {
        log<NUClear::INFO>("The LegIK task is running");
    }
    else if (leg_ik.run_state == GroupInfo::RunState::QUEUED) {
        log<NUClear::INFO>("The LegIK task is queued");
    }
    else if (leg_ik.run_state == GroupInfo::RunState::NO_TASK) {
        log<NUClear::INFO>("The LegIK task has not been emitted");
    }
});
```

3.2 Run Reason

The 'RunReason' DSL word is used to find out information about why a Provider was triggered.

```
on<Provides<Walk>>().then([this] (const Walk& walk, const RunReason& reason) {
    if (reason == RunReason::NEW_TASK) {
        log<NUClear::INFO>("A new Walk Task triggered this reaction");
    }
});
```

Table 2 lists possible RunReasons.

4 The Director Graph

In Figure 1, the Root node represents a non-Provider reaction, e.g. an 'on<Startup>()' reaction. It emits a 'Play' Task with 'emit<Task>(std::make_unique<Play>())'.

RunReason	Description
NEW_TASK	The Provider is running because a new Task of the type it provides for was emitted.
STARTED	The Provider is running because it has now become active.
STOPPED	The Provider is running because it is becoming inactive.
SUBTASK_DONE	The Provider is running because one of its subtasks has completed.
PUSHED	The Provider is running because a higher priority module wants its Causing state to be true.

Table 2: RunReason enum types and their meaning.

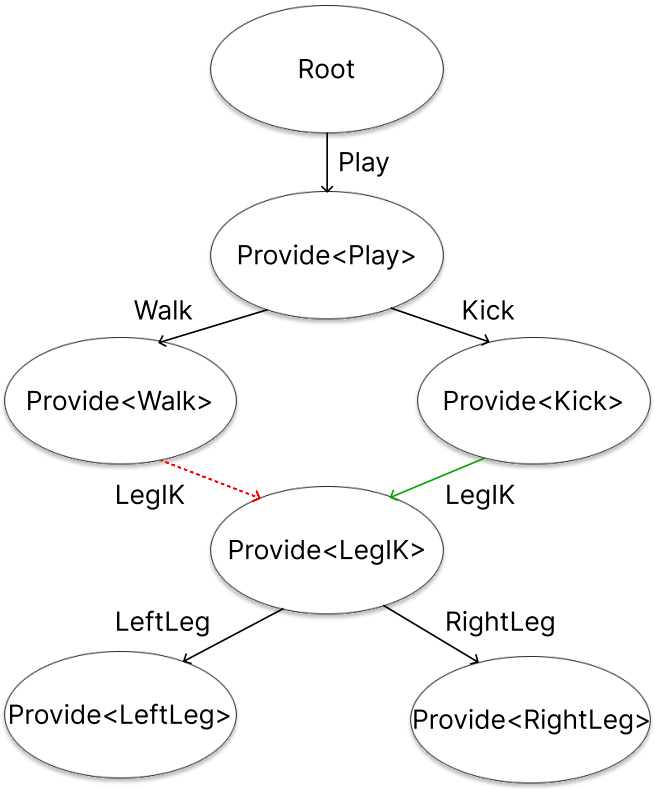

Fig. 1: Example of a Director graph, with the kick blocking the walk.

There is one Provider in this tree for the 'Play' Task, which is the 'Provide<Play>' node. It has two subtasks, 'Walk' and 'Kick'. These are both emitted from the Play Provider. The Kick is given a higher priority than the Walk.

```
emit<Task>(std::make_unique<Walk>());     // 0 priority by default
emit<Task>(std::make_unique<Kick>(), 1); // higher priority number is higher priority
```

The Walk Provider has emitted a LegIK Task, but it is blocked because the Kick Provider has also emitted a LegIK Task. The Kick Provider has a higher priority, so it is the one that has an active LegIK Task.

The Walk Provider's subtask will become unblocked if the Kick Provider is stopped or the Kick Provider stops requesting the LegIK Task.

The LegIK Task emits two subtasks, one for each leg. These in turn would emit subtasks for each servo, but this has been omitted in the graph for clarity.

If the Kick Provider decides it no longer should Kick, such as if the ball is not in front of the robot, it might emit no Tasks. This will remove the LegIK Task from the Kick Provider, allowing the Walk Provider's LegIK Task to run.

```
on<Provide<Kick>>().then([this] (const Kick& kick) {
    if (kick.ball_in_front) {
        emit<Task>(std::make_unique<LegIK>());
    }
});
```

Figure 2 represents the case where the if statement does not run.

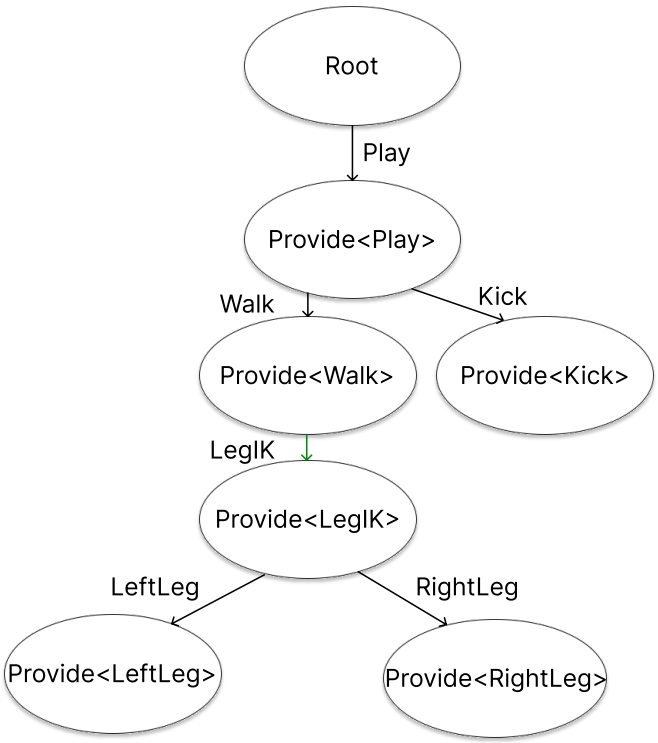

Fig. 2: Example of a Director graph, with the kick not requesting any task, so the walk can run.

This is an instance where multiple instances of the same Task type are emitted. Another situation is where there are multiple Providers for the same Task.

The following example will remove the Kick Provider, and add another Walk Provider. One Walk Provider will run when the robot needs to be precise but slow, such as when there are many obstacles nearby. The other Walk Provider will run when the robot can be less precise and faster.

In this instance, there will be a non-Provider reaction checking for the number of obstacles. If there are many obstacles, the Static Walk Provider will be active. If there are few obstacles, the ZMP Walk Provider will be active. Figure 3 represents the case where there are few obstacles.

```
# Static walk implementation
on<Provide<Walk>, When<Obstacles, std::greater_equal, Obstacles::MANY>>().then({});

# ZMP walk implementation
on<Provide<Walk>, When<Obstacles, std::less, Obstacles::MANY>>().then({});
```

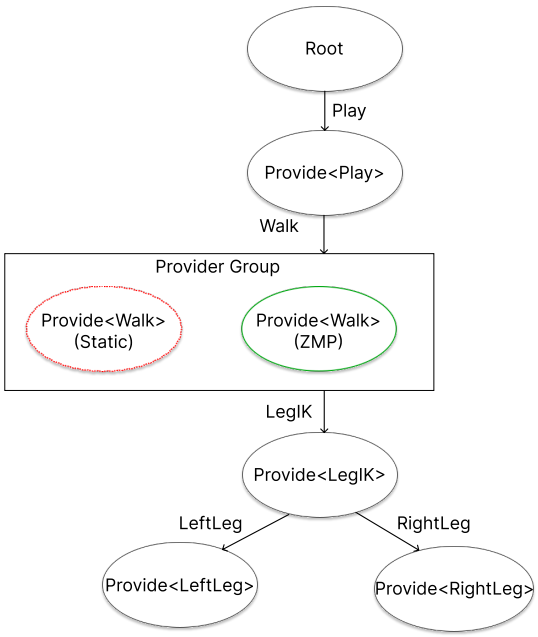

Fig. 3: Example of a Director graph, with two walk Providers in the walk Provider group.

The two Walk Providers form a ProviderGroup. The Director will only allow one Provider from a ProviderGroup to be active at a time. The ZMP Provider was chosen because its When condition was true, and the Static Provider's was not.

5 Behaviour Layout

The Director algorithm and framework is used to implement behaviours in the NUbots codebase. These behaviours can be categorised into multiple layers. The Director does not use the concept of these layers, but they are used to help organise the codebase and make it easier for developers to conceptualise.

There are many modules that exist outside of the behaviour system, such as localisation and vision. These modules are not considered behaviours, but they are used by behaviours to make decisions.

5.1 Purpose

Purpose modules are the highest level modules and will show the 'big picture' view of what is happening in the system. They live in 'modules/purpose' in the source code. They generally will describe the overall goals and purpose of the robot. Some examples of purpose-level modules would be:

- Striker: Combines strategies to walk to the ball and kick it to the opposing goal
- Goalie: Combines strategies for staying near the goal and preventing the opponent from scoring

5.2 Strategy

Strategy modules make high level strategy decisions for the robot, to be used by the purpose modules to achieve their purpose. They live in 'modules/strategy' in the source code. These modules are the strategic elements that provide the 'what' components rather than the 'how' components. They will often combine information from multiple systems in order to decide what action to take. Some examples of strategy level modules would be:

- WalkToBall: Tells the walk path planner to walk to the ball location
- FallRecovery: Tells the falling relax planner and get up planner to recover from any falls

5.3 Planning

Planners are where the system address 'how' the behaviour works. They live in 'modules/planning' in the source code. Planner modules are modules which use skills in order to achieve their goal. They are only interested in a single component of performing an action, such as walking a path or looking around.

- WalkPathPlanner: Determines walk velocity given a target position
- LookPlanner: Controls how the robot is looking for a specific object

5.4 Skill

Skills are the modules that perform the actions specified by the planners. They live in 'modules/skill' in the source code. Some examples are

- Walk: Walks given a desired velocity from the walk path planner
- Kick: Kicks the ball according to the desired direction from the kick planner
- Look: Moves the head to look in a particular location, decided by the look planner

5.5 Actuation

The actuation layer is where the (real or simulated) hardware is controlled. These modules live in 'modules/actuation' in the source code. The end of the actuation layer involves emitting servo commands to be picked up by a platform module, such as HardwareIO or Webots modules. Modules in this layer include

- Kinematics: Takes end effector positions and converts them into servo positions
- Sequences: Takes a sequence of servo positions and emits them in order
- Groups: Takes a group of servos and emits them together, will only emit Done when all servos are done
- Servos: Takes a single servo and emits it, will only emit Done when the time the servo has to move is over

Acknowledgements. This technical report is stored here for archival purposes and is derived from the content found at `https://nubook.nubots.net/system/foundations/director` on the 4th of October, 2023. The authors of the original webpage are the authors of this technical report. The webpage contents are released under the Creative Comments 4.0 license.

References

1. Houliston, T., Fountain, J., Lin, Y., Mendes, A., Metcalfe, M., Walker, J., Chalup, S.K.: Nuclear: A loosely coupled software architecture for humanoid robot systems. Frontiers in Robotics and AI **3** (2016)
2. Sims, Y., Houliston, T.: The director: A composable behaviour system with soft transitions (2023), arXiv preprint arXiv:2309.09248