DSD - Dynamic Stack Decider

A Lightweight Decision Making Framework for Robots and Software Agents

Martin Poppinga[1] · Marc Bestmann[1]

Abstract

We present the *Dynamic Stack Decider* (DSD), a lightweight open-source control architecture. It combines different well-known approaches and is inspired by *behavior trees* as well as *hierarchical state machines*. The DSD allows to design and structure complex behavior of robots as well as software agents while providing easy maintainability. Challenges that often occur in robotics, i.e., a dynamic environment and situation uncertainty, remain well-manageable. Furthermore, it allows fast modifications of the control flow, while providing the state-fullness of a state machine. The approach allows developing software using a simple Domain Specific Language (DSL) which defines the control flow and two types of elements that contain the programmed parts. The framework takes care of executing the demanded portions of the code and gives, due to its stack-like internal representation, the ability to verify preconditions while maintaining a clear structure. The presented software was used in different robotic scenarios and showed great performance in terms of flexibility and structuredness.

Keywords Control architecture · Framework · Behavior · Robots · Agents

1 Introduction

A variety of challenges needs to be tackled in robotics to create software that can produce a complex behavior. A control architecture helps to fulfill these tasks. While some co-routines like the image processing or the walk engine perform scenario-specific, well-defined tasks, high-level planning has to solve more abstract tasks in complex environments. This logical layer, which decides what action needs to be performed, is called the *behavior* in the following.

In earlier years, complex behavior on a high level was often limited, as most systems were designated for a specific task, e.g., cleaning the floor of a room. Due to advancements in robotics, decision-making processes are becoming more complex, especially because in real-world robotic scenarios most often no closed world assumption can be made. For example, in competitive environments with multiple agents

or, in the case of human-robot interaction, different roles and strategies need to be specified, and switched on-demand, as the environment changes quickly and the behavior needs to be adapted to new situations. Similar is true for software agents who deal in uncertain environments, for example, when humans or other agents are involved.

The presented approach consists of two parts. First, the programmed modules of the behavior, which decide and the act, and second, a simple *Domain Specific Language* (DSL) which connects these modules and defines the control flow. This DSL allows fast adaptions in the robot behavior without altering the source code of the modules.

This framework was initially designed for challenges in the RoboCup competition [11] and was developed in the Humanoid Soccer League and used for example on the humanoid robot platform Wolfgang [3] (see Fig. 1). However, it was adapted for other areas of robotics and showed its benefits outside of the RoboCup competition, too. The DSD proved its advantages in several competitions since 2015 and was improved several times, including a clearer formalism in early 2017 under the name *Active Self Deciding Stack* and the addition of the Domain Specific Language in late 2018. We

✉ Martin Poppinga
poppinga@informatik.uni-hamburg.de

Marc Bestmann
bestmann@informatik.uni-hamburg.de

1 Department of Informatics, Universität Hamburg,
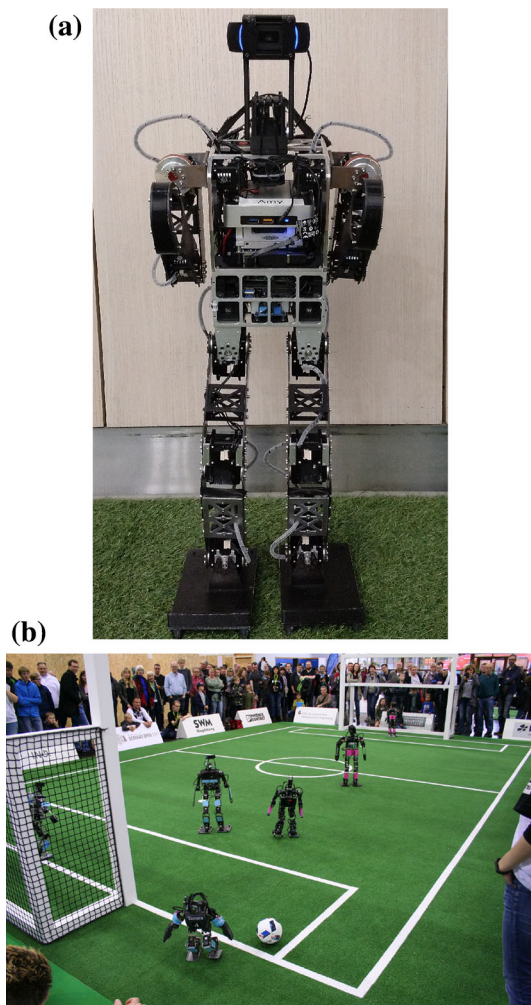Vogt-Kölln-Straße 30, 22527 Hamburg, Germany

(a)

(b)

Fig. 1 The humanoid robot platform Wolfgang (**a**) and a game in the RoboCup competition (**b**) in which the DSD has proven itself

lenges occur in other examples as sensor information is often imperfect and external changes require quick changes in the programming routine. Aside from higher-level decision making, other parts in the robot software require state-full decision making and easily maintainable frameworks for an efficient development process, too.

To enable a feature-rich implementation to handle this complexity in decision making as well as for the developers to adapt and maintain the source code, we extracted several aspects for the here presented framework that need to be considered:

– *States:* For debugging and development it is necessary that developers can easily evaluate in which state the agent currently is. Furthermore, some sub-tasks may have a time component which requires them to stay with their former decision during execution.
– *Reevaluation of previous decisions:* In a dynamic environment preconditions can change rapidly. It is required to reevaluate all such conditions and change the behavior accordingly promptly without getting stuck in an unwanted state.
– *Divisibility:* For structured testing and fast debugging the ability to launch all sub-parts of the behavior separately is beneficial.
– *Maintainability:* Changing or adding parts to the behavior has to be possible at all times without the need for a general restructuring.
– *Code Reuse:* As many routines can occur in completely different states, a mechanism to prevent code duplication is preferable. The means to avoid rewriting specific behavior steps as well as requiring to perform the same check in various modules.
– *Scalability:* The logic needs to express complex behavior for autonomous systems while staying clear and understandable.

This paper is structured as followed. In Section 2 the related work and existing approaches and frameworks for high-level behavior are presented. Our approach is presented in Section 3 and some insight into the implementation is given in Section 4. In Section 5 we evaluate the proposed approach and give a more complex example in Section 6. Finally, in Section 7 we conclude.

While the framework was originally designed for the use in robotics using the ROS Middleware[5], the presented approach and the published software are also suitable for other use-cases outside of robotics. In the following, the term *agent* refers to robots as well as software agents.

use it as high-level tactics behavior[1] as well as an abstraction low-level abstraction mechanism [4][2]. It can be used for compact high-level control of robot behavior and other state-machine-like situations. Furthermore, it was successfully applied in the scenario of a mobile robotic bartender and won the first place in the IROS 2018 Mobile Manipulation Hackathon[3,4].

In the RoboCup Humanoid Soccer competition, only human-like sensors are allowed, and thus object recognition in larger distances provides a difficult task, and decisions are made in an uncertain environment. This challenge increases, as the environment holds multiple robots of which half, the opposing team, do not provide information. Similar chal-

---

[1] https://github.com/bit-bots/bitbots_behavior.

[2] https://github.com/bit-bots/bitbots_motion.

[3] http://iros18-mmh-pal-robotics.com/.

[4] https://github.com/TAMS-Group/tiago_bartender/tree/master/tiago_bartender_stackmachine.

[5] https://ros.org

Fig. 2 Simplified example of the direct acyclic graph (DAG) which is defined by the DSL using the output of the DEs. A chosen path (**blue**) is displayed. The agent decided on being an attacker and going to the ball. These taken decisions and the current action can be seen in the stack (cp. Fig. 3). Since there are multiple ways to reach the GoToBallDirect action, it is necessary to have a history on the stack to retrace the agent's decisions
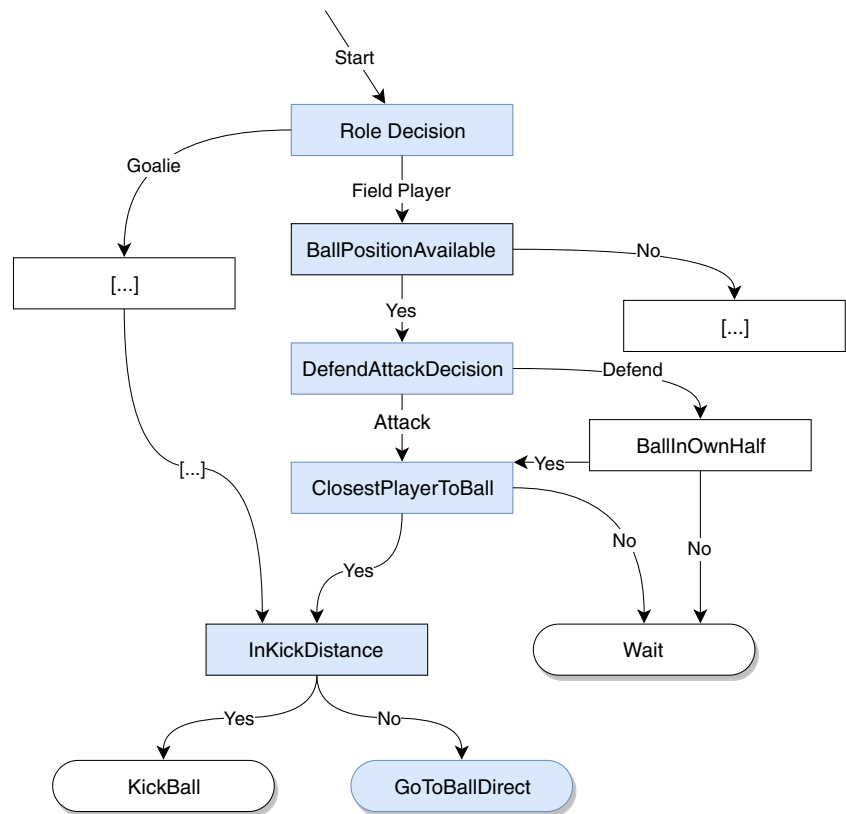
2 Related Work

Due to the importance of control architectures in many fields, several approaches were developed.

Some of them, e.g., finite state machines (FSM) [9], exist for a long time. Others, like behavior trees [6], are more recent and mostly used in the domain of video games. These have already proven to be suitable for complex behaviors, but they are not always ideally suited for the uncertain environments of robotics or in for the use in competitions, where the software needs to adapt constantly. Robot behaviour, while in many scenarios is used for navigation purposes [1], various tasks can be solved using these different approaches [2]. In the following section, the advantages and disadvantages of the most popular approaches are discussed. They are compared with our approach in Fig. 2.

2.1 State Machines

FSM

The *finite state machine* (FSM) [9] is a commonly used approach based on a finite set of discrete states of the agent. Transitions are defined between the states. Typically, actions are performed while entering and leaving a state as well as while the state is active. While this approach is inherently stateful and straightforward to understand, its scalability and maintainability are bad due to the high number of transitions. Furthermore, testing is complicated due to bad divisibility.

HFSM Hierarchical finite state machines (HSMs) [10] are an extension of the FSM. Here, the states are organized in a hierarchical structure. States can be substates of other states which enables the inheritance of behavior from superstates. This way, each substate only needs to focus on its designated tasks and events while its ancestors can handle more general events. This prevents *state explosion* and code duplication in complex systems.

Still, HFSMs are hard to maintain since adding or removing states requires the programmer to reevaluate a large number of transitions [6].

2.2 Planning

For planning sequences of actions and executing them, multiple approaches exist that follow a traditional Sense-Plan-Act manner. The most prominent example is STRIPS [8]. In recent years this area was also influenced by the gaming industry and new approaches arose, e.g., *Goal-Oriented Action Planning* [12].

While these approaches can be used to create sophisticated behaviors, their reactiveness is low, and they often rely on the closed world assumption, making them not applicable in many real-world scenarios.

2.3 Subsumption

The *Subsumption Architecture* (SA) [5] was introduced as a solution to the low reactiveness of classical planning approaches. It divides the task into subtasks that can run in parallel but have a hierarchical order. More important parts can subsume others in the control of the robot's actuators. While this approach has a high reactiveness, it is not stateful and therefore very limited, especially in finding optimal solutions or making decisions based on previous actions.

2.4 Decision Trees

Decision trees (DTs) [13] are a simple concept of connecting multiple conditions into a tree-like structure. They are, among others, applied in the area of artificial intelligence for solving classification problems, but also for decision making. While simple to implement and intuitively understandable, they are stateless. During each iteration, all decisions have to be reevaluated which is expensive. Since the result is only based on a single iteration, it can lead to jumping back and forth between two different results, especially when using noisy sensor data. This makes it challenging to implement non-instantaneous actions. For example, a collision avoidance decision tree might outputs alternating *steer left* and *steer right* when presented with an obstacle right in front. The noise of the sensor will sometimes put it more to the left and sometimes more to the right, maybe resulting in a crash.

2.5 Behavior Trees

Behavior trees (BTs) [7] create a hierarchical tree structure of control elements that result in actions or conditions as leafs. It is possible to define composite tasks which are performed sequentially or in parallel. Due to the possible parallelism and looped conditions, there is not only one point of activity in the tree, but distant parts can be active at the same time. This makes it difficult to see the current state of a tree at a glance. Furthermore, the decision path through the tree is less clear than it is in a decision tree because the conditions are leaf nodes and not internal nodes. This makes a differentiation between actions and conditions more difficult. The implementation of a BT engine is complex since parallelism has to be considered [6]. Since BTs are tick driven, rather than event-driven like most approaches, it requires a change of paradigm [6]. In larger closed-loop systems, many conditions may have to be checked, leading to a high execution time [6].

3 The Dynamic Stack Decider

Our approach combines the simplicity and statefulness of an FSM with the flexibility and scalability of a behavior tree to achieve a light-weighted control architecture that can handle the dynamic and uncertain environment in robotics. The Dynamic Stack Decider (DSD) consists in its central part of a stack-like structure that orders the currently loaded parts of the behavior. Similar to many control systems, the DSD is expected to be called periodically to evaluate its current state and to take actions. A Domain Specific Language (DSL) is specified that defines which elements are pushed on top of the stack depending on the output of the currently executed element and the current position in the tree-like structure. The DSL defines all possible execution paths and creates a directed acyclic graph (DAG).

Using this DSL, further parameters can be passed to the elements. The elements on the stack are divided into *decision* and *action* elements. The stack holds the active components in the current state as well as the history which lead to the current state. This concept is explained in more detail in the following.

3.1 The Elements

Decision Elements

Each *decision element (DE)* capsules one logical decision and has a finite set of possible outputs. It does not control its actors or is in any kind altering the environment. Decisions can be as complex as needed, using if-else clauses for simple cases or, in a more complex situation, for example, neural networks. One example in the RoboCup Soccer domain is the decision whether the agent has sufficient knowledge about the current ball position in its world model. This decision element could be followed by an action to search for the ball or a decision whether the agent should walk towards the ball.

DEs are used in two ways. If they are on top of the stack, the following element is put on top (*pushed*) and this element is executed next. If a DE is inside the stack, it can specify if it wants to be *reevaluated*. When a DE is reevaluated, the outcome of the production is compared to the outcome of the same element in the previous iteration. If it is identical, the stack remains unaltered. When the outcome differs, the stack above the reevaluated DE is discarded, and the new sub-tree is put on top. This method is crucial for validating preconditions, e.g., having sufficient knowledge of the ball location is necessary to be able to go towards it. This mechanism is further explained in 3.3.

Action Elements

The second type of element is the *action element (AE)*. An AE is similar to a state of an FSM, in the meaning that the system stays in this state and executes its logic as long as the

AE is on top of the stack. This is in contrast to a DE which is evaluated instantaneously. An exemplary AE is a kick or a stand-up animation that stays on top of the stack until the animation has finished playing. An AE could also handle going to the ball. In this case, the AE remains on top of the stack until the ball is reached. The AE only makes decisions that are necessary for its own purpose, e.g., some adjustments to the kicking movement. AEs do not push further elements on the stack but control actions on lower-level modules like joint goals. If the AE has finished, it can remove itself from the stack by performing a *pop* command or otherwise remains active until a precondition becomes invalid.

3.2 DSD Description Language

For easy maintainability and visualization, the possible execution flows are described in a designated description language an overview of the symbols is shown in Table 1. This representation is parsed while initialization of the DSD. The DSL determines which return value of a decision element leads to which element executed in the following.

In Listing 1 an example behavior is given, creating the DAG as shown in Fig. 2.

```
1   #Kick
2   $InKickDistance + kick_threshhold:0.1
3       "Yes" —> @KickBall
4       "No" —> @GoToBallDirect
5
6   #Attack
7   $ClosestPlayerToBall
8       "Yes" —> #Kick
9       "No" —> @Wait
10
11  —>SampleBehavior
12  $RoleDecision
13      "FieldPlayer" —> $BallPositionAvailable
14          "No" —> [...]
15          "Yes" —> $DefendAttackDecision
16              "Defend" —> $BallInOwnHalf
17                  "No" —> @Wait
18                  "Yes" —> #Attack
19      "Goalie" —> [...]
20          [...] —> #Kick
```

Listing 1 Example of the used DSL. With Kick and Attack, two subtrees are defined. SampleBehavior defines the starting point. Depending on the output of the $RoleDecision different paths are executed in the control flow which leads to the corresponding actions.

This structure allows fast changes of connections as well as parameters (e.g., thresholds, speeds) in one place, giving a good overview without the requirement of searching in long lists of parameters or directly in the code. Due to the semantic naming of decision results, transition changes can be performed locally.

Table 1 These symbols are used in the DSL. They assemble the DAG as shown in Fig. 2

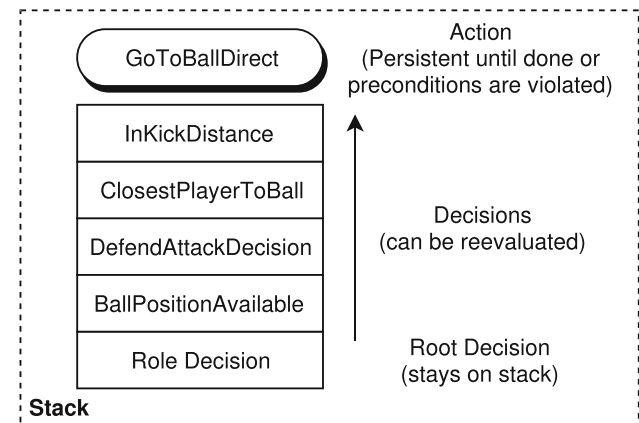|	
$	$Name$ Name of a decision element
@	@Name$ Name of an action element
-->	"ReturnValue" $--> \$, @Name$ Defining the following element
#	#Name$ Defining or using a sub-tree
+	$\{\$, @\}Name + param : p\_value$ Gives initial parameters to the element

Fig. 3 Example of a stack for a field player after taking the decisions as displayed in Fig. 2. Each time step all DEs which should be re-evaluated are called from bottom to top. If no precondition changed, the AE on top of the stack is executed

3.3 Call Stack

The built stack follows a similar logic as call stacks in various programming environments (e.g., Python, Java, .NET), providing the current call stack as a path to the finally executed module. However, in this architecture, each element on the stack can remain active.

The control workflow of the DSD is sketched in Fig. 4 and described in the following: The DSD is initialized with one decision on the stack (Fig. 3) as the root decision. This can be any node in the DAG, allowing simple execution of subgraphs / sub-behaviors. Each iteration, all DEs on the stack can be reevaluated from bottom to top to see if any preconditions have changed. For this, each DE is checked if it requires to be reevaluated. If it has to be reevaluated, the DE is re-performed, and the outcome is compared to the output of the previous iteration. If it has changed, the stack above the current DE is cleared, and the new outcome is pushed. Otherwise, the next element is checked to be reevaluated. After every element has been reevaluated, or after the first different outcome, the reevaluation phase is finished. Then, the topmost element is executed. If the topmost element is

Fig. 4 The control flow of the DSD
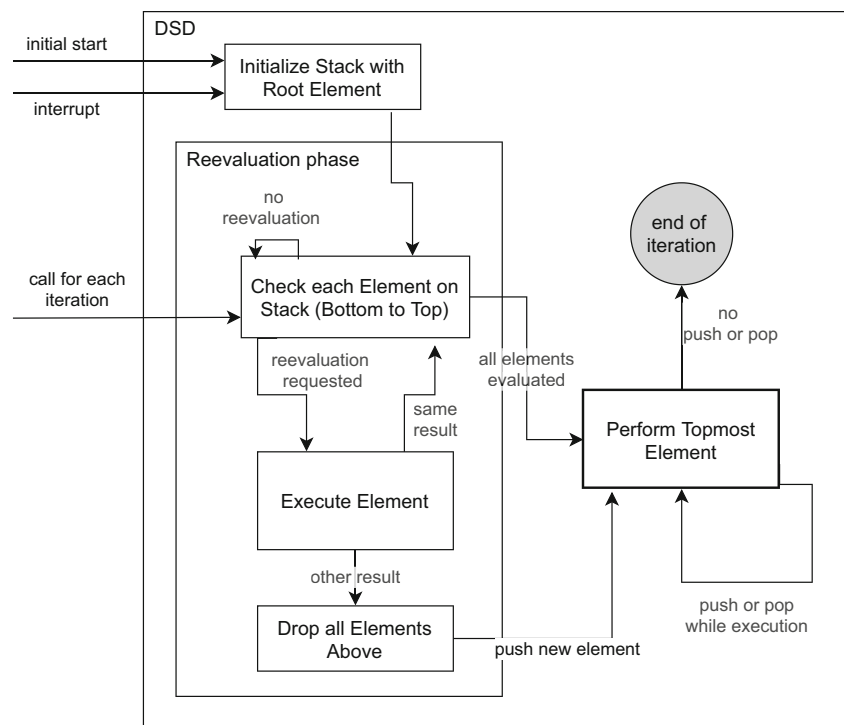
a DE, it is executed, and its outcome defines, according to the DSL, which element or elements is or are pushed next on the stack. If an element pushes one or multiple elements on the stack, the topmost it is directly executed, enabling the evaluation of multiple DEs in one cycle. If the topmost is an AE, the AE is executed. If the AE pops itself from the stack, the new topmost element is executed. If the topmost element is an AE and performs no pop, it stays on top of the stack, and the DSD has finished its iteration.

This way the AE is executed each iteration until either a precondition changes (as checked in reevaluation) or the AE pops itself after completion. At any time an external interrupt can clear the stack and start again with the root element.

By using this stack-like structure, it is always traceable which action the agent tries to perform and which decisions were made.

3.4 Extended Features

A few additional features are added to the base structure to facilitate the use of the DSD:

Interrupts

An interrupt is an event from outside of the structure, which clears the complete stack to reset the behavior as displayed in Fig. 4. In the particular case of RoboCup, we use it when the game state changes, for example, if a goal was scored. However, it can also be used, for example, if the

agent is kidnapped or paused. After the interrupt, the stack is recreated, starting at the root element.

Reevaluate

Each decision element may define a *reevaluate* criteria. If the corresponding method returns true, the element will be executed during the reevaluation phase even if it is in the middle of the stack. This way a precondition can be checked constantly or periodically. As an example, every iteration it could be checked if the ball position is known to the agent with a given certainty. If the now pushed element is different from the element which is currently above in the stack, meaning that the decision changed, the whole stack above will be dropped and the newly selected element executed. Additionally, actions can specify set a *do_not_reevaluate* flag which prevents all decision elements from being reevaluated. Semantically, this allows to specify actions that can not be interrupted, e.g. stopping a dynamic motion animation on a bipedal robot would lead to a fall.

Passing arguments

It is possible to pass arguments to decision and action elements directly in the DSL. This improves the generalization of stack elements and allows to use the same elements in multiple parts of the DAG. For example, it is possible to implement a *Move* Action that takes an argument on the direction to move rather than implementing *MoveForward* and *MoveBackwards*. An example of how this is specified in the DSL is shown in Listing 1.

Actions Sequences

It is possible to perform a sequence of actions. For this, a DE can push multiple AEs at once on top of the stack. In this case, at first, only the topmost AE is executed.

If the element is popped from the stack, the next action is directly executed. This way for example movement patterns can be executed over multiple iterations unless any preconditions change. Alternatively, an AE can be pushed, which does only one task for a single iteration, e.g. sending a message to other agents and then popping itself, followed by a movement in the second AE.

4 Implementation

The reference implementation is written in Python. Using Python [14] allows fast development due to its well readable syntax and the absence of the need to recompile the code on changes. Further, feature rich libraries, like *scipy* or *tensorflow* can be used if needed. As decision making in the presented context is not crucial to milliseconds, the reduced performance in contrast to an optimized C++ variant is negligible. However, the same patterns of a DSD would also apply to runtime-sensitive languages, as i.e. C++.

Listing 2 shows an example of a simple decision element. Each element inherits from an abstract class.

```python
from DSD import DecisionElement

class InKickDistance(DecisionElement):
  def perform(self, data, param):
    if data.world_model.get_ball_dist() <
        ↪ param["kick_threshhold"]:
      return "Yes"
    else:
      return "No"

  def get_reevaluate(self):
    return True
```

Listing 2 An example of a simple decision element. It checks every iteration whether the ball is currently close to the agent.

Data Exchange

Each stack element holds its instance variables as long it is in the stack. The data is discarded when the element is removed from the stack. If persistent data is required, it can be written to a shared *blackboard*. Data required by the elements is structured in different scopes, which encapsulate different parts of the agent's knowledge. For example, information coming from the vision is handled separately from the information of the inter-agent communication. This data can be stored locally or published to other processes. For this purpose, a data handler in the form of a python object can be passed to the framework at startup.

The *getter* may define default values if no data is existing. *Setters* are only existing where it is necessary to publish information. In most cases, the decisions use the *getters* to obtain information from other parts of the system, and only the actions publish commands and data using the *setters* for external modules.

Running Multiple Instances

If two or more independent behaviors are required, it is possible to create multiple independent behavior stacks. For example, in robot soccer, the behavior of the head is often controlled semi-independently from the body behavior. There, the head, equipped with a camera, collects data while the body may request to obtain certain information. As these are independent processes, no common blackboard is accessible for communication between the different DSD instances. In this case, for example, communication is handled by ROS messages.

Reusing Modules

In some cases, it is required to use the stack elements several times but with modified decisions or other outcomes. In these cases, a module can be reused and given a different path in the description file and other parameters can be passed.

Visualization

When integrated into *ROS* it is possible to create a real-time view of the active action and elements on the stack, cp. Fig. 5. As well as alternative paths in the DAG.

Future Work on Implementation

We plan to further improve the usability of our implementation by providing a GUI to create DSDs, which are then saved in the DSL. This improves the overview and lowers the entry barrier. Furthermore, we want to introduce automatic sanity checks, that verify, for example, if all elements are part of the same graph and if all outcomes of a DE lead to another element.

5 Evaluation

In the following, we evaluate different aspects of the DSD compared to other frameworks, especially to behavior trees. An overview can be seen in Table 2.

Hierarchical organization

The DSD allows clear hierarchical ordering of importance based on the decision element's position on the stack. During the reevaluation phase, DEs which are closer to the bottom of the stack are performed earlier and are therefore more important. This is also visible in the DSL description of the DSD since elements specified earlier will end up lower on the stack. The ordering of importance is thereby clearly visible and changeable before and during run time.
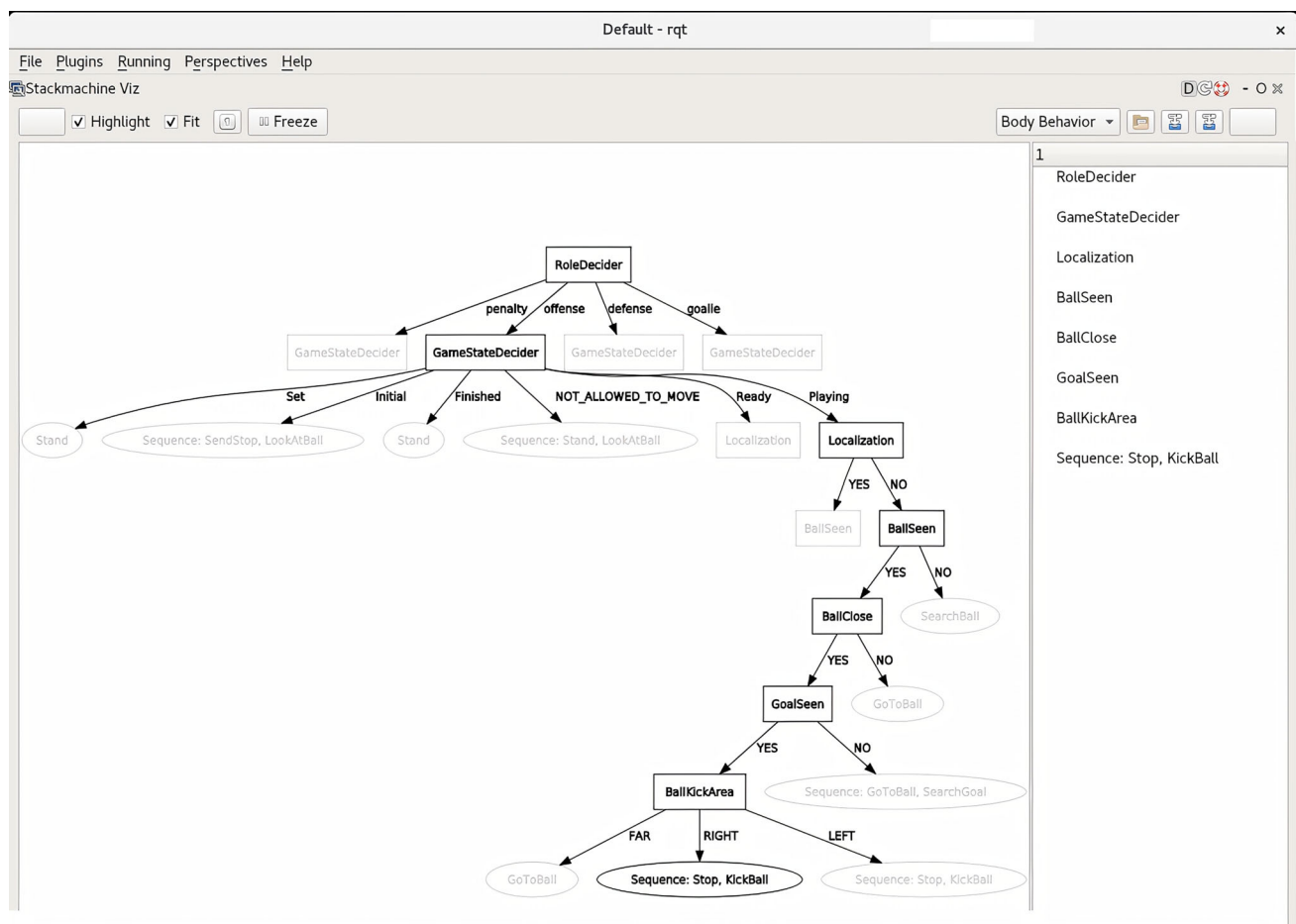
Fig. 5 Live visualization of an exemplary stack DAG inside *rqt*

Table 2 Comparison of different control architectures. Partly following [6, Chapter 2]

	FSM	HSM	SA	DT	BT	DSD
Hierarchical org.	-	+	++	+	+	+
Code reuse	-	○	○	+	+	++
Maintainability	–	○	-	-	+	+
Human readable	-	○	-	+	○	++
Stateful	+	+	-	-	+	+
Scalability	–	○	-	○	++	+
Concept compre.	+	+	○	++	-	-

The other frameworks perform mostly similar in this regard, with the exception of the FSM, which does not allow such an ordering.

Code reuse

Reusing existing code parts, inside one behavior and from previous ones, can significantly decrease development time. The DSD performs especially well in this regard, due to the clear separation of decision and action elements. Further-more, the used DSL allows simple and fast creation of a behavior when reusing existing decision and action elements.

In behavior trees, there is only separation between actions and conditions. The decisions are encoded by a combination of conditions and control flow nodes. Therefore it is often necessary to take subtrees when reusing code. Furthermore, a simple reordering with a DSL is there not possible.

Maintainability

Changes that are introduced after the initial implementation should only require modifications to the corresponding part. The DSD performs well on this as each element can be modified without the need to touch others. Furthermore, the introduction of a new element into the DAG is simple and only requires a local change in the DSL description file.

Behavior trees perform similar well in this regard. In comparison, in an FSM the introduction of a new state may require adding transitions to all other states.

The *Dynamic Stack Decider* allows to explicitly define the required behavior and enables fast replacements of code parts. Since the root decision can be defined at the start, it is

easy to run only a part of the behavior or even just a single action.

Human readable

While programming and debugging it is necessary to have a clearly arranged graphical representation that is easy to understand by the developer. The DSD performs well in this regard due to the clear separation between actions and decisions, the semantic labeling of decision outcomes, and its tree-like structure.

While behavior trees do also provide a clear graphical representation, they are more difficult to read since different parts can be active at the same time and the transitions have no semantic labeling. A typical negative example is the FSM where the high number of transitions for complex systems results in cluttered graphs.

Stateful

The state of a DSD is defined by all elements on its stack. The current active action element is clearly visible since it is the top-most element in the stack. The remaining elements on the stack provide information on the previously taken decisions but are also part of the state since they influence how the DSD will act in the future, e.g., by reevaluation.

In a behavior tree, the state is defined by its currently active nodes. Naturally, the reactive approaches SA and DT have no defined state.

Scalability

While the DSD does generally scale well to complex problems, it has no concept of parallel execution and is therefore limited in this regard. It is possible to get around it by running multiple DSDs concurrently, but this only works well if they are mostly independent of each other.

Behavior trees scale better since they can handle parallelism easily with their tick concept.

```
1   —>Waiter
2   $CustomersWaiting
3       "None" —> $ContinousRoomCheck
4           "Clean" —> @CleanFloor
5           "Check" —> @CheckRoom + room:1 , @CheckRoom +
                ↪ room:2 , @CheckRoom + room:3
6       "AtLeastOne" —> $CustomerDistance
7           "Far" —> @GoToCustomer
8           "Near" —> $SpeakWithCustomer
9               "WantsToOrder" —> @TakeOrder
10              "BringBill" —> @BringBill
11              "Complains" —> @FetchManager
```

Listing 3 Corresponding DSL for the example DSD displayed in Figure 6.

Concept comprehensibility

The amount of time which is spent to understand how a framework works should be as low as possible to keep the development time to a minimum. Some concepts, i.e. BT and DSD, need more time to understand, as they are more complex and less intuitive. Other more simplistic frameworks, for example, ones based on Decisions Trees, are faster to understand but tend to not have the same usability for larger projects.

6 Example

To better demonstrate how the DSD works and its benefits, we consider a robot waiter scenario. The robot works in a three-room restaurant where it should serve customers but also clean when possible. It may find waiting customers randomly while cleaning, but to keep the waiting time low, it should stop cleaning and search for customers in all three rooms every three minutes. When a customer is found, the robot needs to drive towards it, and then it should deal with all wishes of this customer before going to another one. An overview of the resulting DSD is displayed in Fig. 6 and the corresponding description in the DSL is shown in Listing 3.

In the beginning, the robot is cleaning. After some time, the *ContinousRoomCheck* is reevaluated and an action sequence is pushed on the stack. It consists of three times the same action, but with different arguments. The robot checks the first room and the corresponding action is popped. During the check of the second room, two customers are found. Therefore the root decision is reevaluated and further room checks are aborted. The robot drives to the first customer and sees what he wants. After bringing the bill, the customer complains. Only after completely serving this customer the robot drives to the second and takes its order. Afterward, it continues cleaning.

The DSD implementation of this scenario stays small (four decisions and six actions) and is easily readable due to its semantic decisions. The adaptable reevaluation mechanism allows constant checking for new customers on current camera images while moving into other rooms to search for customers is only performed periodically, as this interrupts the cleaning process. While the robot serves a customer, it uses *DoNotReevaluate* to make sure it is not interrupted. The reevaluation mechanism allows giving a clear hierarchical organization of the different tasks by specifying which tasks can interrupt others.

The checking of the three rooms is utilizing the ability to provide arguments to actions and is formed by using an action sequence. If the robot should be expanded, e.g., by a *BringOrder* action, this can be done easily by adding another action. The current stack of the DSD provides a clear state of the robot's behavior, which could, for example, be used by a human supervisor.
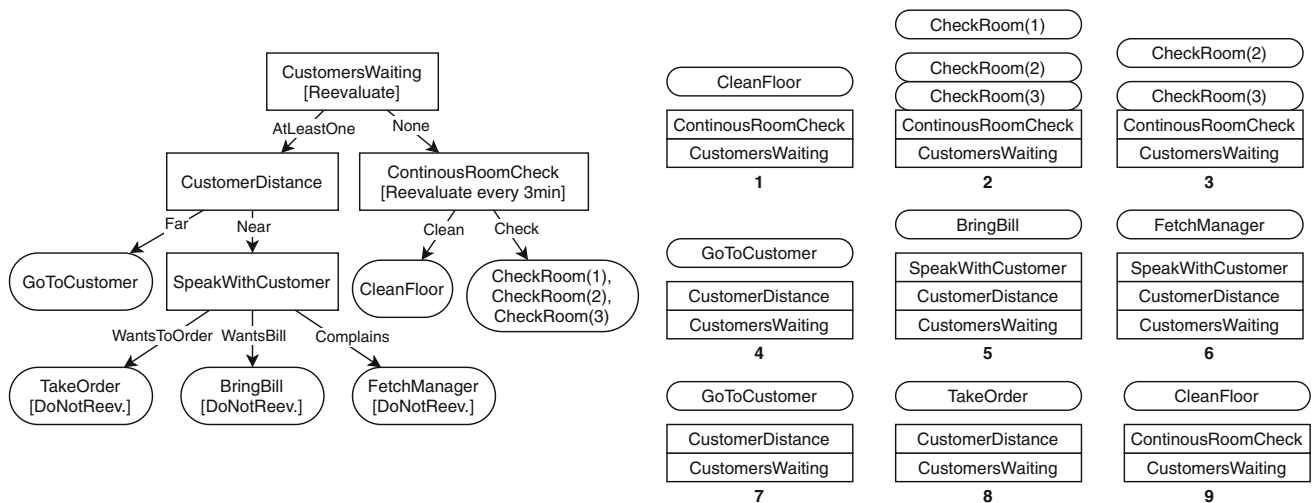
Fig. 6 Example of a DSD for a robot waiter. The DAG (**left**) shows the structure of the DSD with annotations displaying the reevaluation behavior. The stacks (**right**) show steps in an exemplary scenario

7 Conclusion

We presented a lightweight, open-source control architecture for robots and software agents. The presented approach, called *Dynamic Stack Decider* (DSD), gives several benefits over existing techniques. Most importantly, the current state of the behavior, as well as the decisions leading to this state, are clear at all times. Furthermore, decisions and actions are clearly differentiated, reusable, and replaceable. The DSD can easily be created using its own domain-specific language, a feature no other framework provides.

In Section 5 we showed the benefits over existing approaches, especially in readability and maintainability. Our model focuses on more complex behaviors with high readability and good maintainability. Further, it shows its benefits in the development process, as it allows easy code reuse and offers good human readability. For this, we made a trade-off which limited the possibilities of parallelism and the simplicity of the concept.

Finding an objective measurement for the best framework is hard, if not impossible. Further, the scenario where the system is used determines how well-suited an approach is. However, we are confident that the proposed framework works very well for a lot of use cases. It provides a great approach for writing easy understandable and maintainable software.

The RoboCup context was chosen as an example, as this provides a convenient scenario, showing a complex environment while maintaining understandable and showing an alternative application aside from the often shown examples in research, as robot navigation and related behaviors.

We provide a reference implementation [6], which has been used in RoboCup competitions for several years and was further extended and improved over the years. The approach with this implementation was also used in social domains on mobile and non-mobile platforms (see Section 1) and will be used for further research. The written software is provided as a ROS package and is therefore easy to integrate into existing systems. It can utilize many tools and libraries provided by ROS, as rqt or the logging environment. While it is ready to work with ROS, it can still be used ROS agnostic in every environment which supports Python.

Declarations

[6] https://github.com/bit-bots/dynamic_stack_decider

References

1. Adouane L (2016) Autonomous vehicle navigation: from behavioral to hybrid multi-controller architectures. CRC Press . 00018
2. Arkin RC (1998) Arkin. Behavior-Based Robotics. p 04358 MIT press, R.C
3. Bestmann M, Brandt H, Engelke T, Fiedler N, Gabel A, Güldenstein J, Hagge J, Hartfill J, Lorenz T, Heuer T, et al (2019) Hamburg bit-bots and wf wolves team description for robocup 2019 humanoid kidsize. In: RoboCup Symposium
4. Bestmann M, Zhang J (2020) Humanoid control module: An abstraction layer for humanoid robots. In: 2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC), pp 263–268. IEEE
5. Brooks R (1986) A robust layered control system for a mobile robot. IEEE J Robot Autom 2(1):14–23
6. Colledanchise M, Ögren P (2017) Behavior Trees in Robotics and AI: An Introduction . https://doi.org/10.1201/9780429489105
7. Dromey G (2003). From requirements to design: formalizing the key steps. https://doi.org/10.1109/SEFM.2003.1236202. https://doi.org/10.1109/SEFM.2003.1236202
8. Fikes RE, Nilsson NJ (1971) Strips: a new approach to the application of theorem proving to problem solving. Artif Intell 2(3–4):189–208
9. Gill a introduction to the theory of finite-state machines (1962). https://doi.org/10.1109/PROC.1963.2548
10. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274. https://doi.org/10.1016/0167-6423(87)90035-9
11. Kitano H, Asada M, Kuniyoshi Y, Noda I, Osawa E RoboCup The robot world cup initiative. Tech. rep. https://doi.org/10.1145/267658.267738
12. Orkin J (2003) Applying goal-oriented action planning to games. AI game programming wisdom 2 pp. 217–227 . http://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf
13. Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106. https://doi.org/10.1007/BF00116251
14. Sanner MF (1999) Python: a programming language for software integration and development. J Mole Gr Modell 17(1):57–61. https://doi.org/10.1016/S1093-3263(99)99999-0

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH ("Springer Nature").

Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users ("Users"), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use ("Terms"). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.

These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.

We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.

While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;

2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;

3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;

4. use bots or other automated methods to access the content or redirect messages

5. override any security feature or exclusionary protocol; or

6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.

These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.

To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.

Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.

If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com