DSD - Dynamic Stack Decider
A Lightweight Decision Making Framework for Robots and Software Agents

Martin Poppinga, Marc Bestmann *

August 15, 2019

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 which 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 which hold 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 a great performance in terms of flexibility and structuredness.

*Martin Poppinga and Marc Bestmann are with the Department of Informatics, Universität Hamburg, 22527 Hamburg, Germany [1popping,bestmann]@informatik.uni-hamburg.de

1 Introduction

A variety of challenges needs to be tackled in robotics to create software which can 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, the 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 environments 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 [7] and was developed in the Humanoid Soccer League. However, it was adapted for other areas of robotics and showed its benefits outside of the RoboCup competition, too. The DSD proved its advantages over the last years in several competitions and was improved several times. 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 [1].

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 challenges 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 handle this complexity in decision making, we extracted several aspects for the 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

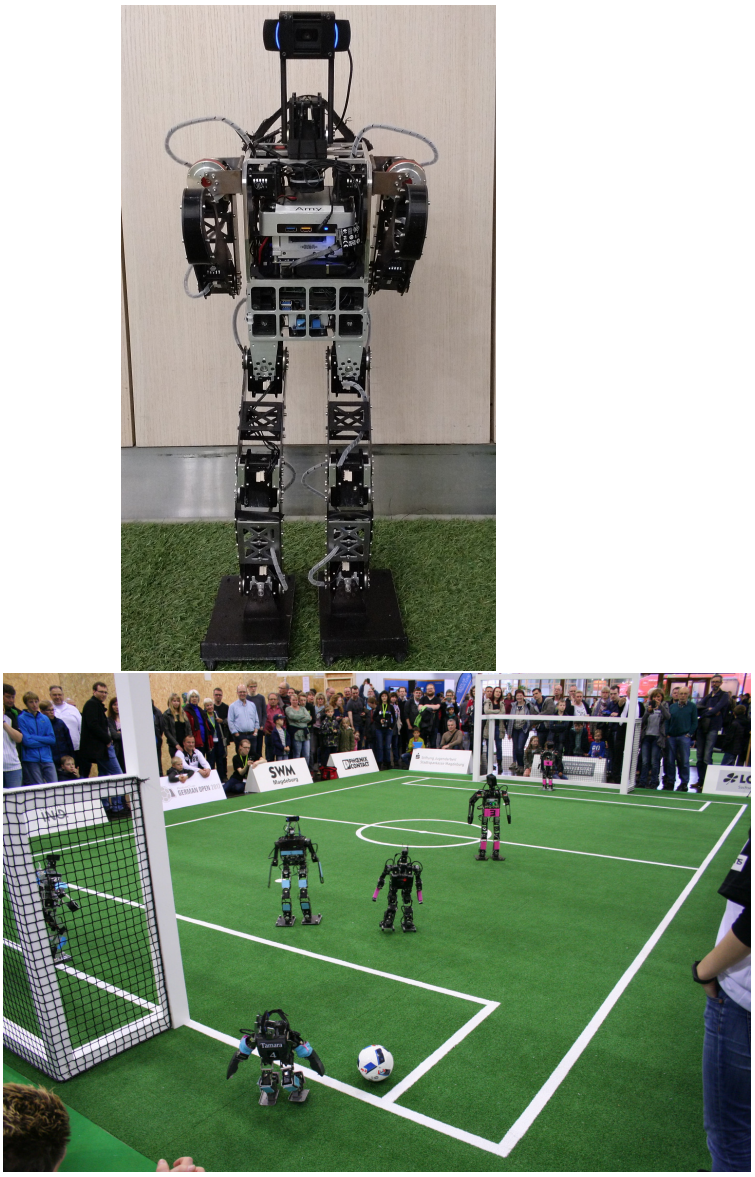

Figure 1: The humanoid robot platform Wolfgang and a game in the RoboCup competition in which the DSD has proven itself.

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

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. Finally, in Section 5 we conclude and discuss future work.

While the framework was originally designed for the use in robotics using the ROS Middleware[2], 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.

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) [5], exist since a long time. Others, like behavior trees [2], 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 software needs to adapt constantly. In the following section, the advantages and disadvantages of the most popular approaches are discussed.

[2] `https://ros.org`

2.1 State Machines

FSM The *finite state machine* (FSM) [5] 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 is bad due to the high number of transitions. Furthermore, testing is complicated due to bad divisibility.

HFSM *Hierarchical finite state machines* (HSMs) [6] 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 [2].

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 [4]. In recent years this area was also influenced by the gaming industry and new approaches arose, e.g., *Goal-Oriented Action Planning* [8]. 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* [1] was introduced as a solution to the low reactiveness of classical planning approaches. It divides the task into subtasks which 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 [9] 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) [3] 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 [2]. Since BTs are tick driven, rather than event-driven like most approaches, it requires a change of paradigm [2]. In larger closed-loop systems, many conditions may have to be checked, leading to a high execution time [2].

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 which 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 which 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 which 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 Figure 2. This representation is parsed while initialization of the DSD. It defines which return values of an decision element elements lead to which following executed element.

$	$\$Name$ 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

Figure 2: These symbols are used in the DSL. They assemble the DAG as shown in Figure 3.

In Listing 1 an example behavior is given. Creating the DAG as shown in Figure 3

```
#Kick
$InKickDistance + kick_threshhold:0.1
    "Yes" --> @KickBall
    "No" --> @GoToBallDirect

#Attack
$ClosestPlayerToBall
    "Yes" -> #Kick
    "No" -> @Wait

-->SampleBehavior
$RoleDecision
    "FieldPlayer" -->
        ↪ $BallPositionAvailable
         "No" --> [...]
         "YES" --> $DefendAttackDecision
              "Defend" --> $BallInOwnHalf
                   "No" --> @Wait
                   "Yes" --> #Attack
    "Goalie" --> [...]
         [...] --> #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 lead to the corresponding actions.

This structure allows fast changes of connections

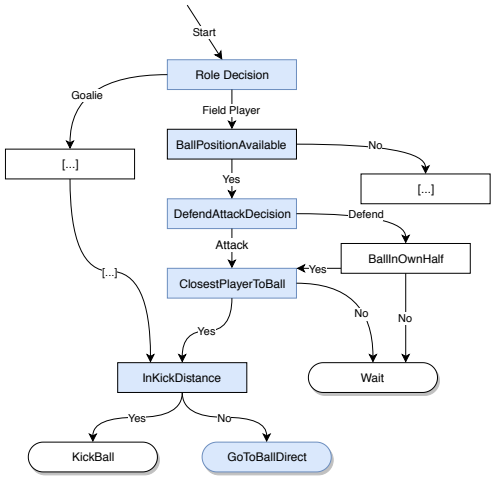

Figure 3: Simplified example of the direct acyclic graph (DAG) which is defined by the DSL using the output of the DEs. A chosen path (**blue**) is displayed. The agent decided on being an attacker and going to the ball. These taken decisions and the current action can be seen in the stack (cp. Figure 4). 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.

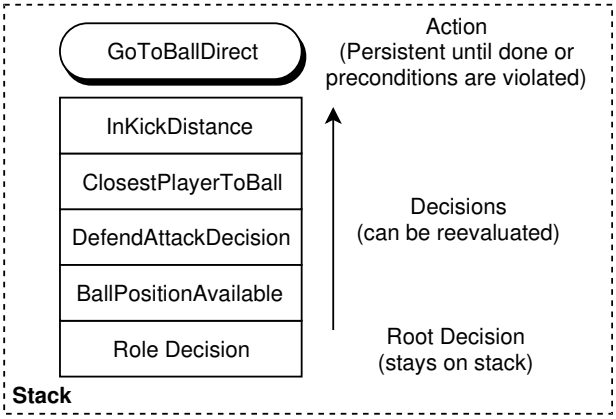

Figure 4: Example of a stack for a field player after taking the decisions as displayed in Figure 3. Each time step all DEs which should be re-evaluated are called from bottom to top. If no precondition changed, the AE on top of the stack is executed.

as well as parameters (e.g., thresholds, speeds) at 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.

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 as described in the following.

The control workflow of the DSD is sketched in Figure 5. The DSD is initialized with one decision on the stack (Figure 4) as the root decision. This can be any node in the DAG, allowing simple execution of subgraphs / sub-behaviors. Each cycle, all DEs on the stack are re-evaluated from bottom to top to see if any preconditions have changed. If this is not the case, the topmost element is executed. If an element pushes another element on the stack, it is directly executed, enabling the evaluation of multiple DEs in

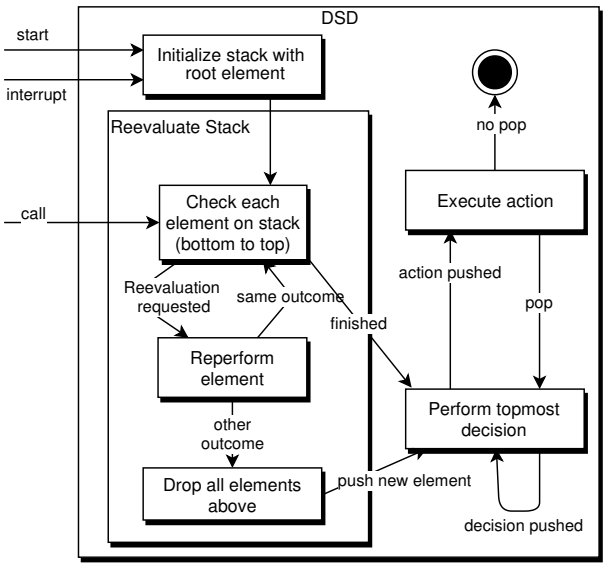

Figure 5: The control flow of the DSD. When started, the root element is pushed on the stack. Then the normal cycle of the DSD is performed, as it is also done when the DSD is called. Each DE is checked if it requires to be reevaluated. If it has to be reevaluated, the DE is re-performed, and the outcome is compared to the previous one. If it changed, all the stack above the current DE is cleared, and the new outcome is pushed. Else 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. Now the topmost DE is performed, and its outcome defines, according to the DSL, which element is pushed next on the stack. The new element is directly executed until an AE is on top of the stack. Then this AE is executed. If it pops itself from the stack, the uppermost DE is performed as before. If the AE performs no pop, it is left on top of the stack, and the DSD has finished its cycle. It waits now to be called again. At any time an external interrupt can reset the stack to the start.

one cycle. At some point, an AE is pushed, which is a leaf in the DAG, and therefore no further pushes are done during its execution, ending this iteration of the DSD. The AE is then called each cycle until either a precondition changes or the AE pops itself after completion. Then the topmost DE is executed, and the process restarts.

By using this 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 Figure 5. 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. In the following iteration, the stack is recreated, starting at the root element.

Actions Sequences It is possible to perform a sequence of actions by pushing multiple AEs at once on top of the stack. In this case, after the uppermost action is done the element is popped from the stack, and the next action is directly executed.

4 Implementation

The reference implementation made by our team is written in Python. Using Python [10] 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.

```
1  from DSD import DecisionElement
2
3  class InKickDistance(DecisionElement):
4    def perform(self, data, param):
5      if
           ↪ data.world_model.get_ball_dist()
           ↪< param["kick_threshhold"]:
6        return "Yes"
7      else:
8        return "No"
9
10   def get_reevaluate(self):
11     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 information of the inter-agent communication. This data can be stored locally or published to other processes. For this purpose, a data handler in 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. The head, equipped with a camera, collects data while the body may requests 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.

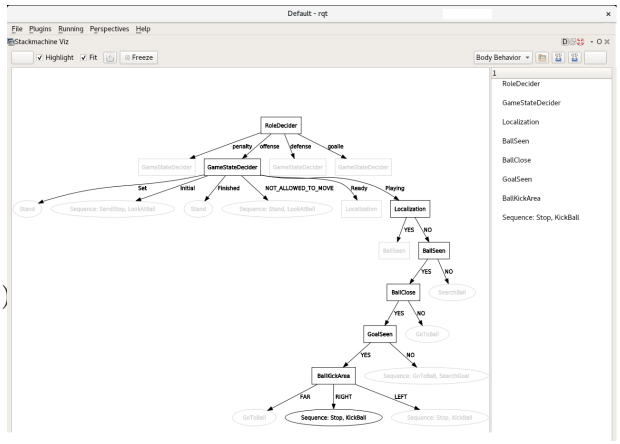

Figure 6: Live visualization of an exemplary stack DAG inside *rqt*.

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. Figure 6. As well as alternative paths in the DAG.

Future Work on Implementation We plan to further improve the usability of our implementation by providing an 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 Discussion & Conclusion

A lightweight, open source control architecture for robots and software agents was developed. The presented approach 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 and reusable.

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. This is more difficult for FSM and GOAP. Parts of the implementation can easily be reused in other places of the behavior.

Furthermore, it allows specifying which of the decision elements need to be reevaluated, in contrast to decision trees. This can be important if decisions are expensive to compute. While this is possible to do in behavior trees, this would lead to a more complicated structure compared to the DSD. Thus the programmer can think on a more local level similar to specifying tasks in GOAP but without the need of defining a cost function. In relation to BTs, the DSD is more lightweight and therefore simple to implement. The resulting graph is easier to understand since the decisions have semantic outcomes, instead of the more abstract tick system of BTs. Furthermore, only a single part of the graph is active at the same time, giving a better overview of the current state of the agent.

The presented approach allows an easily maintainable and flexible way of defining decision making for robots and software agents. The flexibility is especially beneficial in fast-changing and uncertain environments. Its advantages compared over currently used approaches have been shown.

The implementation provided[3] has been used in RoboCup competitions for several years and was further extended and improved over the years. It is ready to work with ROS, but can still be used ROS agnostic in every environment which supports Python2 or Python3.

Acknowledgments.

Thanks to Nils Rokita for helping to implement the DSD, thanks to Finn-Thorben Sell for implementing the visualization tool, thanks to Timon Engelke for cleaning up the implementation and thanks to the Hamburg Bit-Bots for the support.

References

[1] Brooks, R.: A robust layered control system for a mobile robot. IEEE journal on robotics and automation **2**(1), 14–23 (1986)

[2] Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction (2017). URL `https://arxiv.org/pdf/1709.00084.pdf`

[3] Dromey, G., Research Online, G., Dromey, R.G.: From Requirements to Design: Formalizing the Key Steps Author From Requirements to Design: Formalizing the Key Steps (2003). DOI 10.1109/SEFM.2003.1236202. URL `http://hdl.handle.net/10072/32937`

[4] Fikes, R.E., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial intelligence **2**(3-4), 189–208 (1971)

[5] Gill, A.: Introduction to the theory of finite-state machines (1962). URL `http://agris.fao.org/agris-search/search.do?recordID=US201300488343`

[6] Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987).

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

DOI 10.1016/0167-6423(87)90035-9. URL https://www.sciencedirect.com/science/article/pii/0167642387900359

[7] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E.: RoboCup: The Robot World Cup Initiative. Tech. rep. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.7511&rep=rep1&type=pdf

[8] Orkin, J.: Applying Goal-Oriented Action Planning to Games. AI Game Programming Wisdom 2 pp. 217–227 (2003). URL http://www.jorkin.comhttp://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf

[9] Quinlan, J.R.: Induction of decision trees. Machine Learning **1**(1), 81–106 (1986). DOI 10.1007/BF00116251. URL http://link.springer.com/10.1007/BF00116251

[10] Sanner, M.F.: Python: a programming language for software integration and development. Journal of molecular graphics & modelling **17**(1), 57–61 (1999). DOI 10.1016/S1093-3263(99)99999-0. URL http://www.python.org/doc/Comparisons.html