# Behavior Trees in SCCharts

Yorik Timo Hansen

Bachelor's thesis
September 2024

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
Dr. Alexander Schulz-Rosengarten

**Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

# Abstract

Behavior Trees (BTs) offer simplicity and modularity for modeling reactive systems where the focus is on responding to environmental changes. In contrast, Sequentially Constructive Statecharts (SCCharts) provide a state-driven execution flow, allowing synchronous execution with a high level of abstraction. This thesis explores different approaches for translations of Behavior Trees into SCCharts, aiming to combine the advantages of both models. It presents a systematic approach to translating Behavior Trees into SCCharts while preserving their modularity and reactivity. The research analyzes challenges in mapping implicit state behaviors from Behavior Trees and optimizing SCChart models for readability and scalability. The presented results demonstrate both the benefits and limitations of this translation, suggesting future improvements such as the incorporation of dataflow mechanisms and enhanced visualization techniques to better manage complexity.

## Acknowledgements

I would like to express my gratitude to Prof. Dr. Reinhard von Hanxleden and the Real-Time and Embedded Systems Group for their assistance in addressing my questions and for offering a welcoming and supportive environment within the working group. Special thanks go to my advisor, Dr. Alexander Schulz-Rosengarten, for his guidance and support throughout the process.

I am also deeply grateful to Finn Evers, Merlin Felix, and Tokessa Hamann for their valuable feedback and companionship, especially during the final days of writing.

Lastly, I would like to extend my appreciation to Dr. h. c. Marit Hansen for taking the time to proofread this thesis.

# Contents

# List of Algorithms

# List of Figures

# List of Tables

Chapter 1

# Introduction

Behavior Trees (BTs) and state machines are two widely used models for controlling the behavior of systems. Behavior Trees, initially popularized in the gaming industry, have since been adopted across various domains, including robotics and automation, due to their simplicity, modularity, and ability to manage complex tasks in a reactive manner. They model system behavior using a hierarchical structure of nodes that determine the flow of execution based on the state of the environment. On the other hand, Finite State Machines (FSMs) and Hierarchical State Machines (HSMs) are foundational in modeling deterministic processes where system behavior is dictated by distinct states and transitions.

With Sequentially Constructive Statecharts (SCCharts), a programming language for reactive HSMs using the Sequentially Constructive Model of Computation (SC MoC), there exists a formalism that allows for the modeling of complex systems in a synchronous environment while maintaining a high level of abstraction.

This thesis explores different methods for translating Behavior Trees into SCCharts, aiming at preserving the core benefits of Behavior Trees while leveraging the structured, state-driven execution flow provided by SCCharts. By analyzing the structural and functional similarities between Behavior Trees and HSMs, this work identifies systematic translation patterns that maintain semantic equivalence, modularity, and reactivity. Furthermore, the research examines the challenges posed by Behavior Trees' implicit state and the potential for optimizing SCCharts to reduce complexity and improve scalability.

The objective of this thesis is to evaluate different translation possibilities between Behavior Trees and SCCharts. Through the development of a proof-of-concept translation tool, this work provides a practical implementation of the proposed translation strategies, enabling the generation of SCChart models from Behavior Trees Extensible Markup Language (XML) files. The results highlight both the advantages and limitations of the approach, including the potential for maintaining modularity and responsiveness as well as the visual complexity introduced by the translation process. This study also outlines potential future improvements, such as incorporating dataflow mechanisms and blackboard variables to enhance modularity and control within SCCharts.

Chapter 2

# Preliminaries

In this chapter, a foundational understanding of the key concepts and models relevant to this thesis is provided. To contextualize the translation of Behavior Trees into SCCharts, the concept of the Behavior Tree and its structure are first introduced, followed by a discussion on FSMs and HSMs. Additionally, the SCChart language, is introduced. These preliminary concepts form the basis for the analysis and translation processes that are explored later in this thesis.

## 2.1 Behavior Trees

Behavior Trees are a way to model the behavior of a system. They are a tree structure, where the behavior is controlled by a small set of node types that each return one of the three values SUCCESS, FAILURE or RUNNING to its parent node. A Behavior Tree works with ticks. A tick is a discrete time step in which the whole tree is evaluated. A tick starts at the root node, which then ticks its single child node which ticks its child nodes recursively. In the end of a tick, the nodes return their value to their parent node. The root node then returns the value of the whole tree to the caller. The tree is ticked again in the next time step.

The few available node types that are depicted in Table 2.1 can be differentiated into leaf nodes, control flow nodes and decorator nodes, where the leaf nodes represent the actual behavior of the system while the control flow nodes decide which child to tick next. Decorator nodes are a type of control flow node that, depending on the implementation of the Behavior

**Table 2.1.** The return values of the different control flow nodes in a Behavior Tree with $N$ children (abbreviated as *ch*). This table is adapted from [MCS+14, Tbl. III] and [CÖ17, Tab. 1.1].

| **Node Type** | **Symbol** | **Returns SUCCESS** | **Returns FAILURE** | **Returns RUNNING** |
|---|---|---|---|---|
| Root | $\varnothing$ | Tree succeeds | Tree fails | Tree is running |
| Sequence | $\rightarrow$ | All *ch* succeed | One *ch* fails | One *ch* is running |
| Fallback | ? | One *ch* succeeds | All *ch* fail | One *ch* is running |
| Parallel | $\rightrightarrows$ | $\geqslant M$ *ch* succeed | $> N - M$ *ch* fail | Otherwise |
| Decorator |  | Custom | Custom | Custom |
| Conditon |  | If true | If false | Never |
| Action |  | On completion | Impossible | Not yet completed |

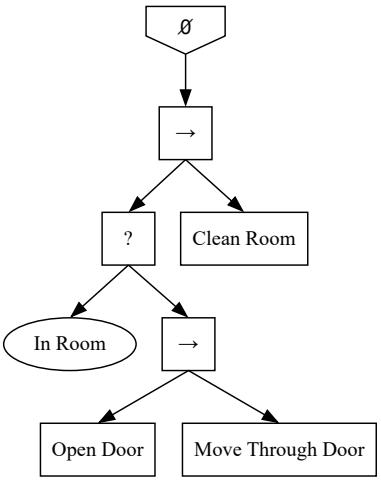



**Figure 2.1.** A simple Behavior Tree for a cleaning robot. The robot first checks if it is in the room. If it is, it cleans the room. If it is not, it opens the door and moves through it to clean the room.

Tree framework, can be used to influence the behavior of the tree as well.

If not stated otherwise, the following definitions are based on [CÖ17].

### 2.1.1 Leaf Nodes

Leaf nodes are either condition nodes or action nodes. They do not have any child nodes. A condition node returns SUCCESS if the condition is met and FAILURE otherwise. An action node returns SUCCESS if the action was successful, FAILURE if the action failed, and RUNNING if the action did not finish in the current tick.

In the example Behavior Tree in Figure 2.1, the leaf nodes are the condition node *In Room* and the action nodes *Open Door*, *Move Through Door*, and *Clean Room*.

### 2.1.2 Sequence and Fallback Nodes

The sequence node as well as the fallback a.k.a. selector node are control flow nodes. Sequence (fallback) nodes tick their child nodes in order from left to right until one of the children returns FAILURE (SUCCESS) or RUNNING. If all children return SUCCESS (FAILURE), the sequence (fallback) node returns SUCCESS (FAILURE). If a child returns RUNNING, the sequence (fallback) node also returns RUNNING.

This means, that the Behavior Tree in Figure 2.1 will first tick the sequence node. This node will check with a fallback node, if the robot is in the room. If it is, the fallback node returns SUCCESS and the sequence node will tick the Clean Room action. If the robot is not in the room, the fallback node ticks its second child, which is a sequence node. This sequence node will first tick the Open Door action and if that is successful, the Move Through Door action. If any of these actions fail, the sequence node will return FAILURE. As there is no third

child in the fallback node, it will propagate the FAILURE to the sequence node, which will then return FAILURE to the root node.

### 2.1.3 Parallel Node

The parallel node is a control flow node that ticks all of its children. After all child nodes have been ticked, the parallel node calculates the return value based on a threshold value $M$. If at least $M$ children return SUCCESS, the parallel node returns SUCCESS. If on the other hand more than $N - M$ children return FAILURE, the parallel node returns FAILURE. If neither of the two conditions are met, the parallel node returns RUNNING.

### 2.1.4 Decorator Nodes

Decorators are a special type of nodes that allow a lot of flexibility in the Behavior Tree, as they are customizable. Every decorator has exactly one child node. The decorator can map the return value of the child node to another value and can also control the number of times the child node is ticked. Some examples of decorators are the inverter decorator, which inverts the return values SUCCESS and FAILURE, and the repeater decorator, which ticks the child node a fixed number of times.

Depending on the framework, this customization can be done programmatically during runtime or by defining the Behavior Tree in a configuration file. This allows for a degree of flexibility that blurs the line between leaf and control flow nodes, as decorators may be used to execute actions while controlling the flow of the tree. In some implementations like the Unreal Engines Behavior Tree framework, it is recommended to use decorators instead of condition nodes[1].

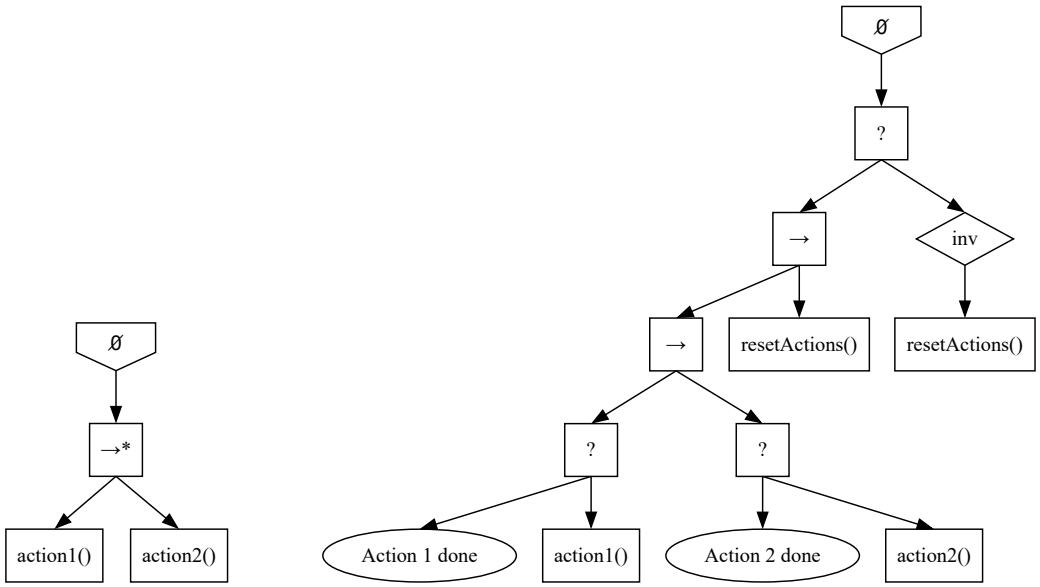
### 2.1.5 Shared Memory with Blackboards

Shared memory in Behavior Trees is generally not defined. Many frameworks solve this issue by extending Behavior Trees with blackboards. The blackboard is a globally shared memory where data is stored in key-value pairs. This allows for nodes to read and write data to the blackboard and thus share information between nodes. Some definitions of Behavior Trees allow for locally scoped blackboard variables (i.e.: [SAC+24]). Other definitions explicitly forbid scopes other than global [GBJ+20].

### 2.1.6 Sequence with Memory

Another common extension are sequence with memory nodes. The sequence with memory node is a sequence node that remembers the child that was last ticked. If it returns RUNNING, it will skip the previous child nodes in the next iterations and tick the child that was ticked

[1] https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-tree-in-unreal-engine---overview?application_version=5.4




**(a)** A sequence with memory node. **(b)** The same behavior by using implicit memory.

**Figure 2.2.** Two Behavior Trees that implement the same behavior. The tree on the left uses a sequence with memory node, while the tree on the right uses implicit memory by using a sequence and a blackboard. Adapted from [CÖ17, Fig. 1.8] with the addition that the Behavior Tree here also resets the memory after the sequence has been completed.

last. If it returns SUCCESS, it will tick the next child in the sequence until it reaches the end of the sequence or a child returns FAILURE. Sequence with memory nodes can be implemented using blackboards and normal sequence and fallback nodes as shown in Figure 2.2. Some frameworks also offer fallback with memory nodes, which work accordingly.

Some implementations of Behavior Trees like *Panda BT* use sequence with memory nodes as the default sequence node[2]. This is contradicting to the philosophy of Behavior Trees as it leads to less reactivity and is not as expressive as using implicit memory in blackboards to store the currently active child, but it makes the trees smaller and more readable [BZS21].

## 2.2 Finite State Machines and Hierarchical State Machines

A FSM is a computational model used to represent and control execution flow based on a finite number of states. It consists of states, transitions between these states, inputs, and actions. In an FSM, the system resides in one state at a time, and transitions between states occur in response to specific inputs. FSMs are commonly used in control systems, parsers, and sequential logic, as they provide a structured approach for modeling deterministic processes where the system's future behavior is fully determined by its current state and input.

[2] https://www.youtube.com/watch?v=W7p34qhBux8

A HSM extends the concept of an FSM by allowing states to be organized hierarchically. This introduces the concept of superstates and substates, where superstates can encapsulate common behaviors across multiple substates, reducing redundancy and improving scalability in complex systems. HSMs allow for more efficient modeling of systems with nested or overlapping states, making them useful in applications where complex state dependencies or transitions occur.

## 2.3 Sequentially Constructive Statecharts

SCChart is a synchronous language that extends the concept of HSMs in the SC MoC. It is a textual language that generates graphical HSM models. As a synchronous language, SCCharts are executed in ticks where the system is in a well-defined state at the beginning and end of each tick [Mot17].

SCCharts have a large set of features that make them suitable for modeling complex systems, as shown in Figure 2.3. The Features are split into two categories: Core SCCharts and Extended SCCharts. Core SCCharts are the basic features of the language, while Extended SCCharts are syntactic sugar to simplify the modeling process. This allows for a high level of abstraction while maintaining a clear and structured representation of the system's behavior. Some of the key features of SCCharts that are used for this work include:

*During actions* Actions that are executed while a state is active, enabling continuous monitoring or adjustments during a state.

*History transitions* Retain the most recent active substate within a superstate, allowing a system to resume from where it last left off.

*Immediate transitions* These enable state transitions to occur within the same tick, allowing for rapid responses to events.

*Local declaration of signals* Enables signals to be declared within a local scope, facilitating modular design and reducing the complexity of signal management.

*Parallel regions* Allow for concurrent execution of states, enhancing the modeling of systems with simultaneous activities.

*Transition priority* Ensures that when multiple transitions are possible, they are executed in a defined order of importance.

*Terminations and weak aborts* Terminations signal the end of a state while weak aborts allow for interrupting ongoing actions under specific conditions without losing intermediate results.

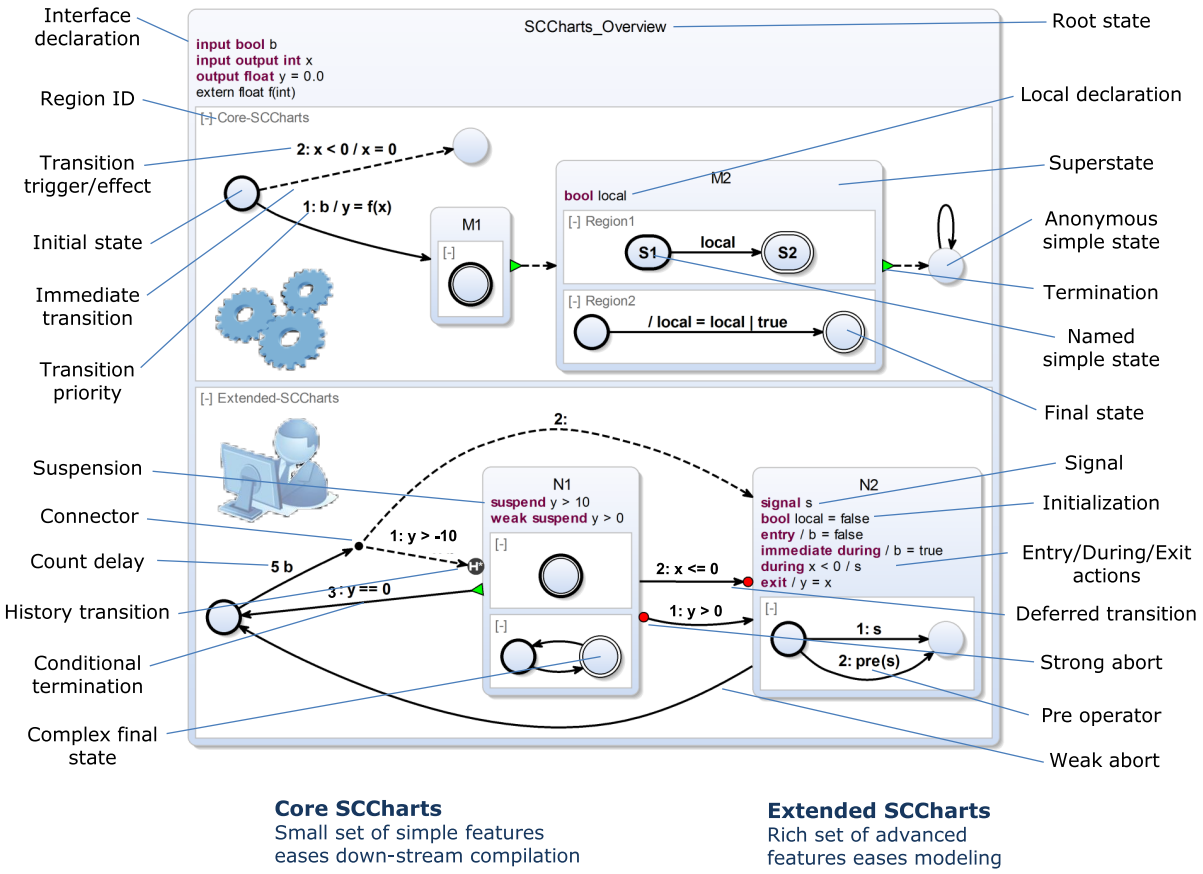



**Figure 2.3.** SCChart visual syntax overview [Mot17, Figure 3.2.1].

Chapter 3

# Related Work

The synthesis of FSMs from Behavior Trees was first explored by Kim et al. [KMW+12]. They introduced a formal transformation from Behavior Tree models to UML state machines, providing a pathway from natural language requirements to executable models. While their work was pioneering in linking Behavior Trees and FSMs, they have used Behavior Trees as described by Dromey [Dro03] while this work focusses on the kind of Behavior Trees that is also used in the video game industry.

Further advancements in the relationship between Behavior Trees and formal languages were demonstrated by Schulz-Rosengarten et al. [SMA+23], who established a representation of Behavior Trees in the synchronous programming language Esterel. This work showcased how Behavior Trees can function within synchronous languages, expanding their applicability to fields such as industrial automation and robotics.

In more recent studies, Ahmad [Ahm23] pointed out the need to extend Behavior Trees with explicit dataflow modeling to handle data dependencies within Behavior Trees, addressing potential issues like non-determinism. Building on this, Schulz-Rosengarten et al. [SAC+24] proposed integrating dataflow mechanisms within Behavior Trees using the Lingua Franca polyglot coordination language, improving modularity and reuse.

A notable body of research has focused on comparing FSMs and Behavior Trees, as well as HSMs. Colledanchise and Ögren [CÖ17] introduced foundational concepts for translating between FSMs/HSMs and Behavior Trees, contributing to the growing understanding of their similarities and differences. Iovino et al. [IFF+24] conducted a practical comparison of FSMs and Behavior Trees in robotics, highlighting the benefits of Behavior Trees, especially in complex task management, in terms of modularity and maintainability over FSMs.

Additionally, Ghzouli et al. [GBJ+23] provided a comprehensive study comparing Behavior Trees and state machines in robotic applications, analyzing their usage within domain-specific languages and open-source projects. Their work demonstrates an increasing trend toward adopting Behavior Trees in real-world systems and suggests significant overlap in the design concepts of both behavior modeling languages.

Finally, Klöckner [Klö15] proposed methods for encoding internal states directly and explicitly within Behavior Trees, offering solutions for tasks requiring memory. By embedding FSMs in Behavior Trees tasks and incorporating memory tasks, they provide hybrid solutions for working with stateful tasks and Behavior Trees.

Chapter 4

# Concept

The objective of this thesis is to evaluate different translation methods for converting Behavior Trees into SCCharts. To achieve this, it is necessary to analyze the structural relationship between these two formalisms. This chapter will explore patterns within Behavior Trees that can be systematically mapped to SCCharts, aiming to identify a translation process that is both general and adaptable to various Behavior Trees implementations. Additionally, the chapter will investigate how implicit states within Behavior Trees can be detected and effectively utilized during the translation to SCCharts.

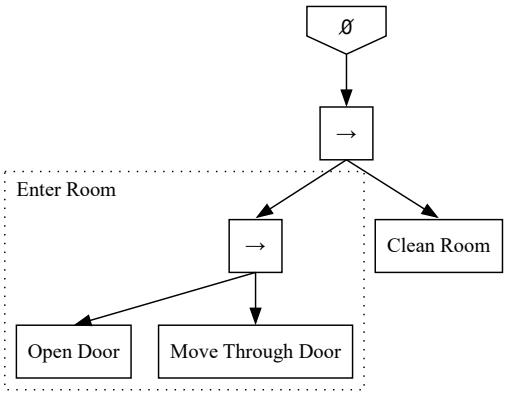
## 4.1 The Structure of Behavior Trees

The strength of Behavior Trees lies in their simplicity and their reactiveness. The simplicity comes from the fact that Behavior Trees only use a few different types of nodes and that the control flow is controlled using only three return types. The reactiveness is achieved by evaluating the tree on every tick from the root node. The hierarchical structure of Behavior Trees enables modelling and modularizing complex behavior in simple ways. The upwards propagation of return types allows actions to easily change the control flow of the current tick.

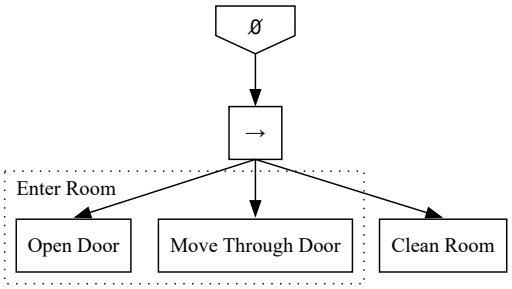
**Algorithm 1:** Pseudocode of a parallel node with $N$ children and a success threshold $M$ [CÖ17].

```
1 for i ← 1 to N do
2     childStatus(i) ← Tick(child(i))
3 if Σ_{i:childStatus(i)=Success} 1 ⩾ M then
4     return Success
5 else if Σ_{i:childStatus(i)=Failure} 1 > N − M then
6     return Failure
7 return Running
```

Colledanchise and Ögren provide a pseudo code snippet (as seen in Algorithm 1) that defines the behavior of parallel nodes. According to this definition, parallel nodes in Behavior Trees are executed in a order defined by the programmer. This ensures determinicity but loses some advantages of parallel execution as it does only allow for limited shared resources and does not facilitate processes that wait for each other in a single tick. The return value




**(a)** A Behavior Tree where a robot enters a room before cleaning it. The "Enter Room" subtree is reusable for other Behavior Trees.




**(b)** A Behavior Tree where a robot enters a room before cleaning it. It is less modular than the tree in Figure 4.1a.

**Figure 4.1.** Comparison of two different Behavior Trees that model the same behavior. The tree in Figure 4.1a is more modular than the tree in Figure 4.1b, but also larger.

of a parallel node is checked after every child node has terminated. Therefore action nodes are prevented from aborting the execution of other child nodes. These limitations ensure that subtrees of parallel nodes are not interfering with each other. Similar definitions can be found in many scientific publications on Behavior Trees [MCS+14; CN22; Ögr]. Recently, other implementations of parallel nodes have been proposed that are more flexible [SMA+23; SAC+24].

The core of Behavior Trees is very modular, as the tree structure makes it simple to reuse subtrees in different Behavior Trees. This modularity sometimes results in larger trees. In Figure 4.1, two different Behavior Trees are shown that model the same behavior. A robot enters a room before cleaning it. Entering the room requires opening a door and passing through it. While this is modeled with its own sequence node in Figure 4.1a, the tree in Figure 4.1b does not contain this subtree. This flattens the tree but makes it harder to reuse the behavior of entering a room. Generally, Behavior Trees can be flattened by removing sequence nodes that have a sequence node as a parent as well as removing fallback nodes that have a fallback node as a parent. This can be done recursively until the tree is flat, without changing the behavior of the tree. This often makes the tree harder to maintain as the modularity is lost, but in certain cases with small Behavior Trees it can be beneficial to reduce the size of the tree and make it thereby easier to understand.

As Behavior Trees are designed to easily model complex behaviors of systems, they are often modeled on a high level of abstraction. For example the Behavior Tree in Figure 4.2 shows the model for a waiter robot. Actions like `FindBottle()` are a black box that can be implemented in many different ways. This high level of abstraction makes the Behavior Tree easy to understand and to design, although it decreases the expressiveness of the model [BZS21].

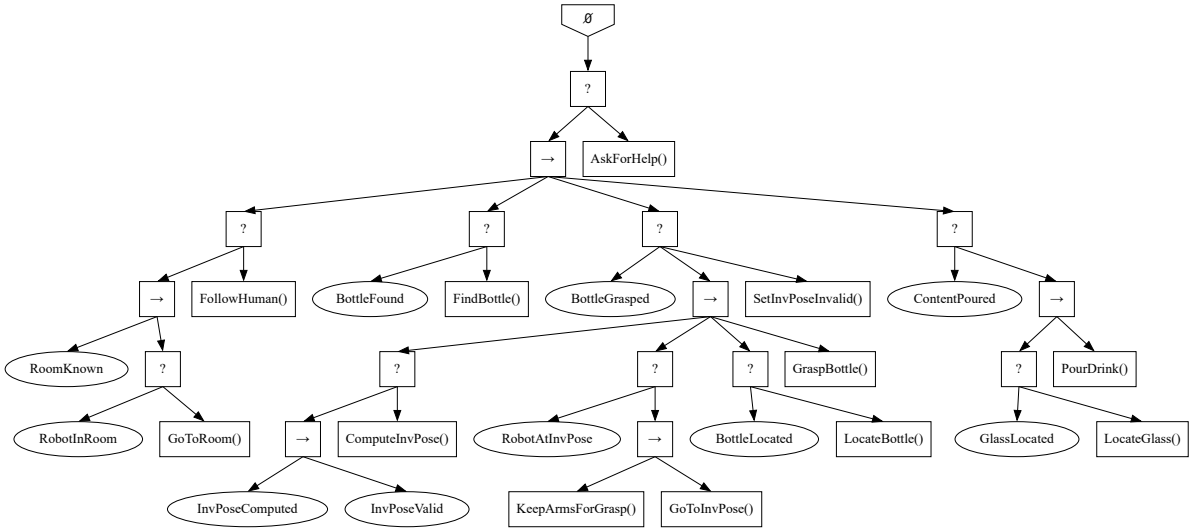



**Figure 4.2.** A Behavior Tree for a waiter robot. Based on Yarp-SmartSoft-Integration[1].

## 4.2 Categorizing the Concepts of Behavior Trees and Comparing Them to Hierarchical State Machines

This section provides an overview of the key concepts of Behavior Trees and compares them with the corresponding concepts in HSM. The table below presents a detailed side-by-side comparison, highlighting similarities and differences in aspects such as control flow, modularity, and concurrency.

Table 4.1: Comparison of the concepts of Behavior Trees and Hierarchical State Machines.

| **Concept** | **Behavior Trees** | **Hierarchical State Machines** |
|---|---|---|
| Root | The root node of a Behavior Tree is ticked every time step. | The initial state of an HSM is the first state ticked. Subsequent ticks process the current state. |
| Conditions | Condition nodes read the state of the environment. | Conditions serve as guards for transitions between states. |

Continued on next page

[1] https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/

Table 4.1: Comparison of the concepts of Behavior Trees and Hierarchical State Machines. (Continued)

| **Concept** | **Behavior Trees** | **Hierarchical State Machines** |
|---|---|---|
| Actions | Action nodes define system behavior, with multiple actions executed in the same tick. | Actions execute on transitions, with some frameworks allowing execution during state entry, exit, or presence. Multiple actions per tick can be achieved via immediate transitions. |
| Control Flow | Sequence and fallback nodes control the tree's flow, relying solely on the environment. | Control flow in HSMs is governed by transitions between states, depending on both the environment and the current state. |
| Modularity | Behavior Trees are inherently modular, allowing subtrees to be designed and reused independently. | HSMs are also modular, with state machines used as substates, and scoped variables enabling better information control across states. |
| Concurrency | In scientific publications parallel nodes are often ticked in a deterministic predefined order while frameworks often come with concurrent execution of child nodes that introduce race conditions and unpredictable behavior [CN22]. | Depending on the framework, HSMs may support parallel regions through multithreading or deterministic scheduling, such as in SCCharts, preventing race conditions. |
| Blackboard | As discussed in Section 4.6.3, Behavior Trees often use a blackboard for storing information globally, effectively treating it as part of the environment. There exist other implementations that allow for scoped variables [SAC+24]. | HSMs utilize scoped variables for sharing information between states, offering more controlled information management. SCCharts provide deterministic semantics. |

Continued on next page

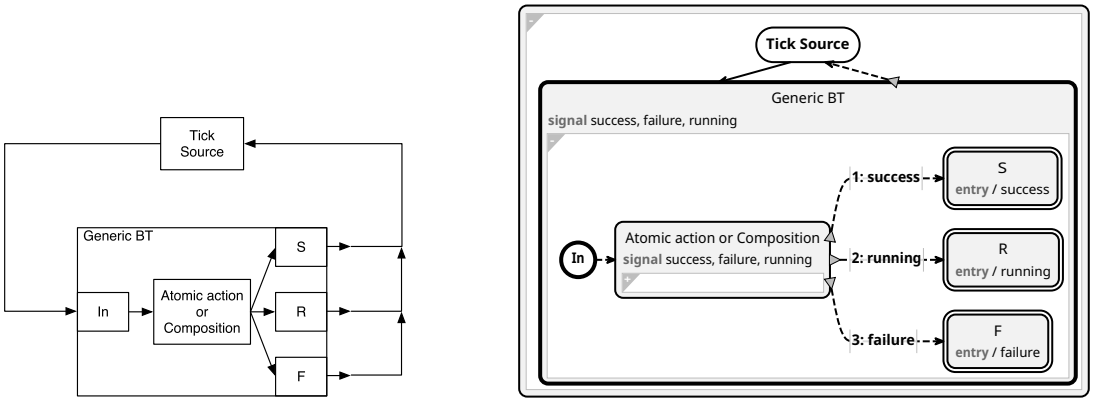
Table 4.1: Comparison of the concepts of Behavior Trees and Hierarchical State Machines. (Continued)

| **Concept** | **Behavior Trees** | **Hierarchical State Machines** |
|---|---|---|
| Responsiveness to Changes in the Environment | Behavior Trees actions are driven solely by the environment state. | HSMs actions depend on both the environment and the current state, with more reactive behavior enabled by state abortion without needing full transitions. |
| Expressiveness | Behavior Trees provide a high-level abstraction for modeling system behavior, offering simplicity for complex tasks. | According to [CÖ17], HSMs offer a lower-level abstraction, typically closer to implementation, and can be more complex. This depends on the framework. For example with their extended feature set SCCharts offer a more readable, but less expressive model that works on a higher abstraction level, but core SCCharts operate on a lower abstraction level. A detailed view on the expressiveness of different Behavior Trees and HSMs can be found in [BZS21]. |

## 4.3 Encoding the Control Flow of Behavior Trees in SCCharts with Immediate Transitions

Colledanchise and Ögren proposed a way to translate Behavior Trees to HSMs that keeps the reactiveness and modularity of the Behavior Trees, by preserving the original hierarchy of the Behavior Tree and using immediate transitions to switch between the different states [CÖ17]. Exactly one delayed transition is used to tick the inner state machine. Figure 4.3a shows the proposed translation and how a similar approach could translate a Behavior Tree to an SCChart side by side. The following differences can be observed:

- Colledanchise and Ögren did not explicitly use a specific framework for their translated HSMs. Their proposed translation does not differentiate between immediate and delayed transitions. As Behavior Trees evaluate the whole tree in every tick, the SCChart must use only transient states. This means that the state machine does not maintain any persistent internal state across execution cycles and purely relies on immediate transitions. Every state (except the "AwaitTick" state) is exited in the same tick it is entered.
- In SCCharts, the SUCCESS, FAILURE and RUNNING states are final states as this makes it easier




**(a)** An FSM behaving like a Behavior Tree [CÖ17, Fig. 2.5]. **(b)** An SCChart behaving like a Behavior Tree.

**Figure 4.3.** Comparison of the state machines of Colledanchise and Ögren and the SCChart translation. The SCChart uses signals to communicate the output values of the nodes.

to leave the inner state machine only when it traversed the whole Behavior Tree. As it is not possible to connect inner states and outer states with transitions, the output value is read from local signals that are scoped to the inner state machine.

▻ Colledanchise and Ögren completely ignore guards on their transitions. In SCCharts the guards are used to map the output value to the corresponding final state. They therefore check the signal emitted by the inner state machine and transition to the corresponding final state.

As a proof of concept of the following work, this approach is implemented in a python script that takes a Behavior Tree in the form of an XML file (compatible with the BehaviorTree.CPP framework[2]) and generates an SCChart file. This shows that it is possible to systematically map the control flow of Behavior Trees to executable SCCharts using immediate transitions. The script can be found in the in Appendix A.

The most basic nodes, the leaf nodes, are a black box in translated SCCharts. As long as a action node emits one of the three output signals or the condition node emits SUCCESS or FAILURE, the corresponding states can do anything to interact with the environment. It is therefore possible to use SCCharts as action and condition nodes.

The control flow nodes get translated by transitioning between nodes of the same hierarchy. For example a basic sequence node as shown in Figure 4.4 gets translated into an SCChart as being presented in Figure 4.5. The initial state is connected to the first child node. If a child node terminates with SUCCESS, the state machine transitions into the next child node. If any child node terminates with FAILURE, the state machine transitions into the FAILURE state, and if any child node terminates with RUNNING, the state machine transitions into the RUNNING state. The SUCCESS state is only reached if all child nodes terminate with SUCCESS.

[2] https://www.behaviortree.dev/

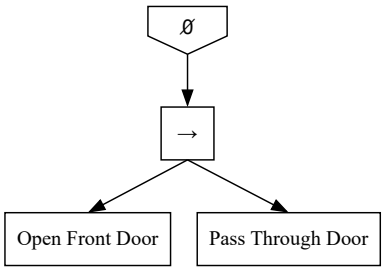



**Figure 4.4.** An example for a sequence node in a Behavior Tree. The robot opens a door and only passes through if opening the door was successful [CÖ17, Fig. 2.8].

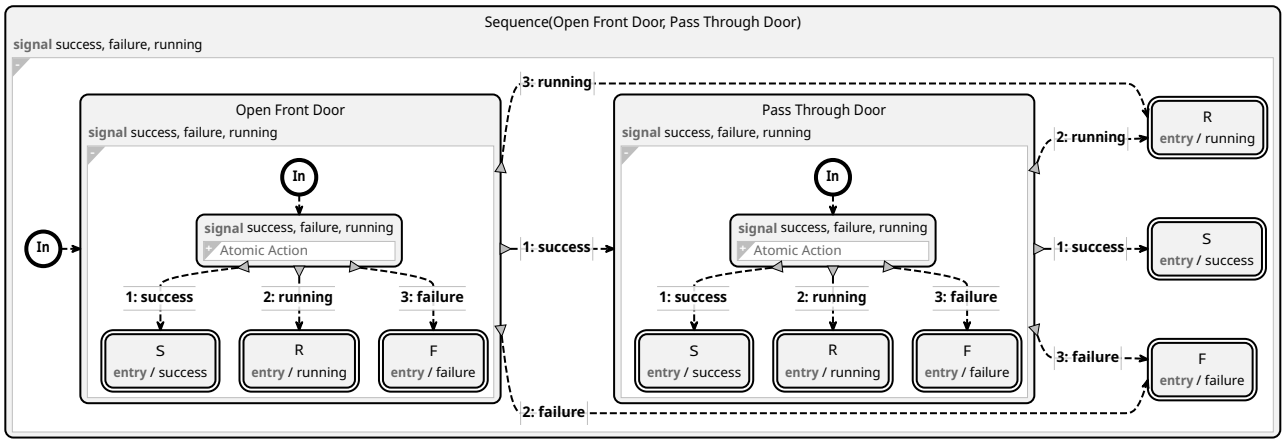



**Figure 4.5.** The sequence node from Figure 4.4 as an SCChart. The state machine transitions between the child nodes depending on their output values.

The fallback nodes are generated accordingly. As the hierarchy level of the SCChart corresponds to the depth of the Behavior Tree, nested trees are translated to nested state machines. As a result, the more complex Behavior Tree in Figure 4.6 generates the SCChart in Figure 4.7. The structure of the original Behavior Tree is preserved in the SCChart but is less readable due to the large number of states and transitions. The SCChart starts evaluating with the outermost "In"-state and traverses the inner state machines as the Behavior Tree would. The outer state simulates a fallback node while the two inner states each simulate sequence nodes.

## 4.4 Extending the Approach

In their work, Colledanchise and Ögren do not define how to handle parallel nodes or decorators. Following the pattern of simulating the control flow with immediate transitions, the pattern of the previous section can be extended to handle parallel nodes and decorators as described in this section.

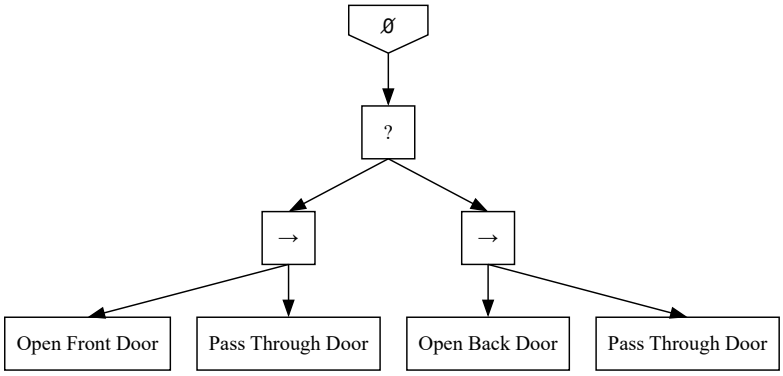



**Figure 4.6.** A more complex Behavior Tree with nested trees [CÖ17, Fig. 2.10].

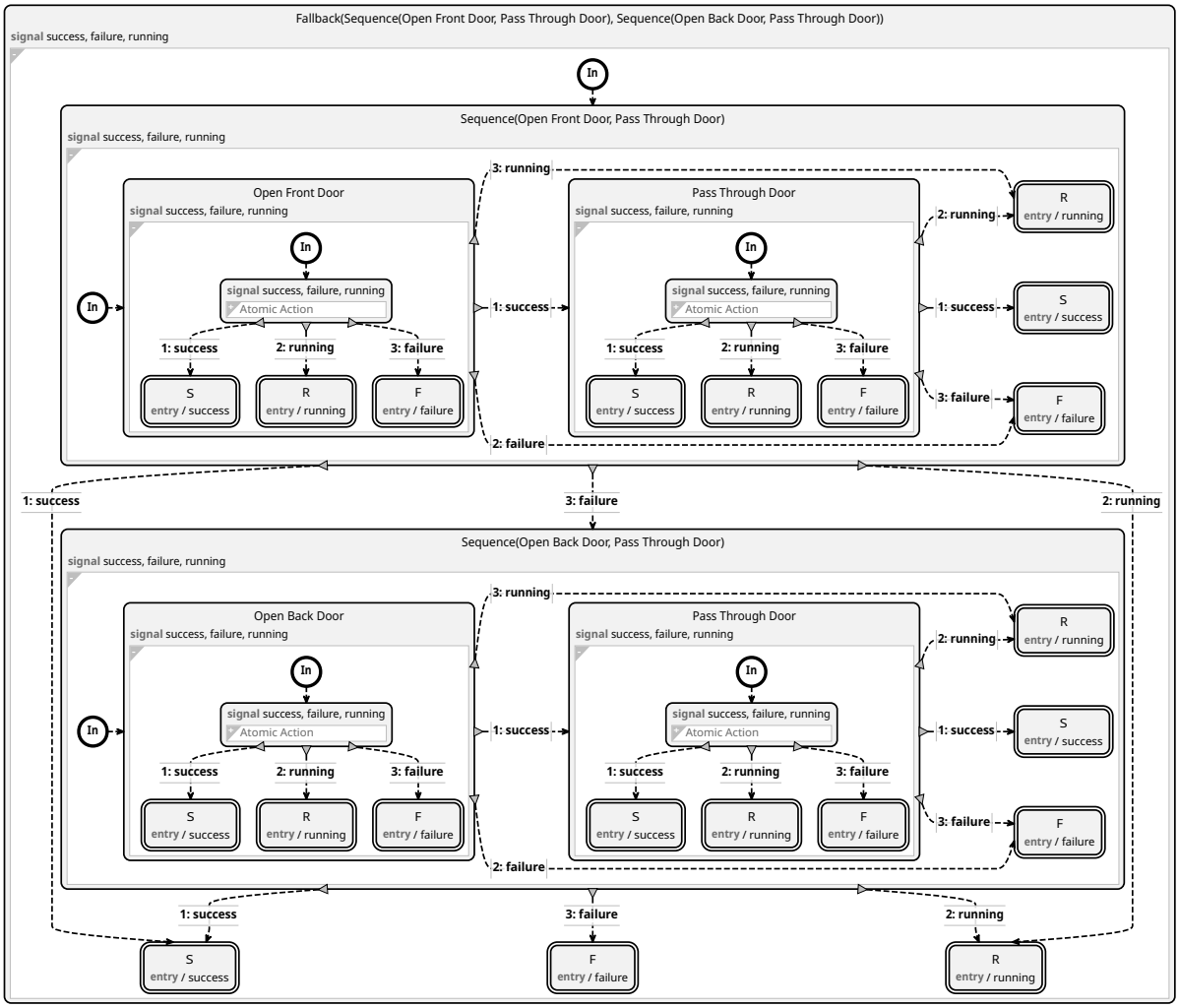



**Figure 4.7.** The Behavior Tree from Figure 4.6 as an SCChart. The generated SCChart is more complex and less readable due to the amount of transitions and generated states.

### 4.4.1 Parallel Nodes

One challenge with parallel nodes is that they must wait for all child nodes to return a value before deciding on the output value depending on the number of returned SUCCESS and FAILURE values. In FSM this could be achieved by building a product automaton. However, the number of states grows exponentially with the number of child nodes as can be seen in Figure 4.9. This FSM works without variables by storing the number of child nodes that have terminated with SUCCESS and FAILURE in the state itself. In the example, the threshold $M$ is set to 1, which means that the parallel node terminates with SUCCESS if at least one child node terminates with SUCCESS, RUNNING or FAILURE. For example, the *Action 2* state from Figure 4.9 has three copies representing the numbers of the return values of the previous child nodes. The comparison with the threshold $M$ and the number of child nodes $N$ is also directly encoded in the transitions. As $M$ is 1, the state machine will always transition to the SUCCESS state if the first child node terminates with SUCCESS or any of the second child nodes terminates with SUCCESS. It will only return FAILURE if both child nodes terminate with FAILURE. Changing this value would require the generation of a new FSM.

Since SCCharts allow the use of variables, the number of states can be reduced by using counter variables to keep track of the number of child nodes that have terminated with SUCCESS or FAILURE. As an example Figure 4.8 shows a parallel node with two child nodes and a threshold $M$ of 1. The SCChart in Figure 4.10 keeps track of the number of child nodes that terminated with SUCCESS and FAILURE with `successCounter` and `failureCounter`. After a termination of a child node, the corresponding counter is incremented. If the child node returns RUNNING, the control flow continues with the next child node without changing any counter. If the last child node has terminated, the output state is determined by checking the counter variables. If the number of children that have terminated with SUCCESS is greater or equal to the threshold $M$, the state machine transitions into the SUCCESS state. If the number of children that have terminated with FAILURE is greater than $N - M$, the state machine transitions into the FAILURE state. Otherwise the state machine transitions into the RUNNING state. This approach does not only use variables to reduce the number of states, but encodes the number of child nodes $N$ and the threshold $M$ as constants in the SCChart. This makes the SCChart more readable and easier to maintain.

Building on the idea of using variables, ticking the child nodes in parallel is possible by using parallel regions. As discussed in Section 4.1, this is not semantically equivalent to the definition by Colledanchise and Ögren, but it fulfills the general definition of ticking every child node and deciding on the output value after all child nodes have returned a value. Figure 4.11 shows that this strategy still needs variables to keep track of the return values of the child nodes. Additionaly, it is necessary to use join transitions to ensure that the child nodes have all terminated before deciding on the output value. Using parallel regions comes with the benefits of the SC MoC such as being able to handle shared resources in a single tick. The compiler schedules the execution of the different parallel regions in a deterministic order, where write processes are executed before read processes. This is not real concurrency, as the execution order is pre defined and the program is running on a single thread. As synchronous

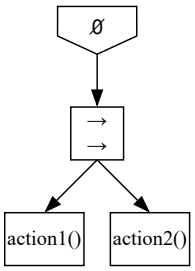



**Figure 4.8.** A parallel node with two child nodes and a threshold of 1.

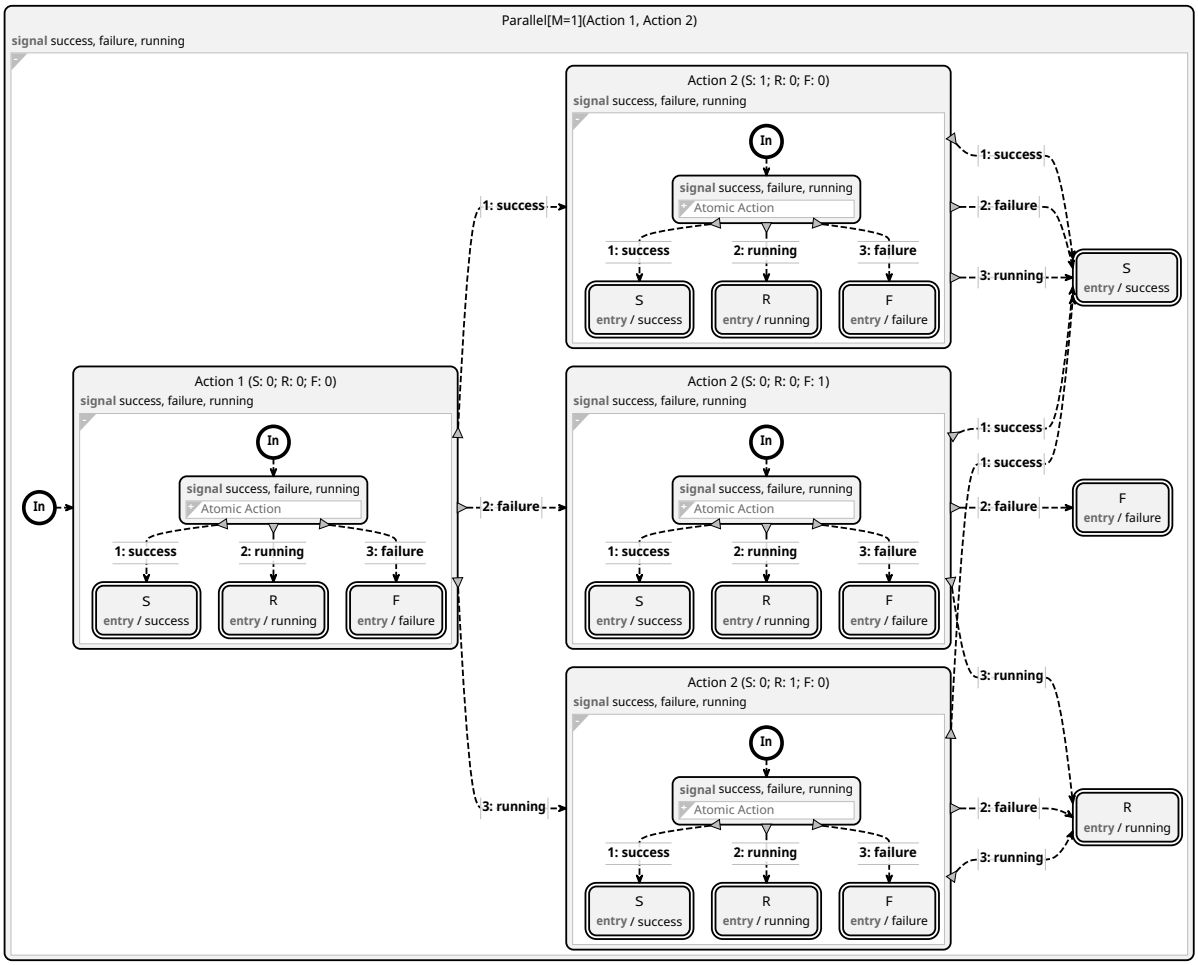



**Figure 4.9.** The parallel node from Figure 4.8 as a product automaton.

languages work under the assumption that atomic actions and computations do not take time, this is not a problem [BG92].

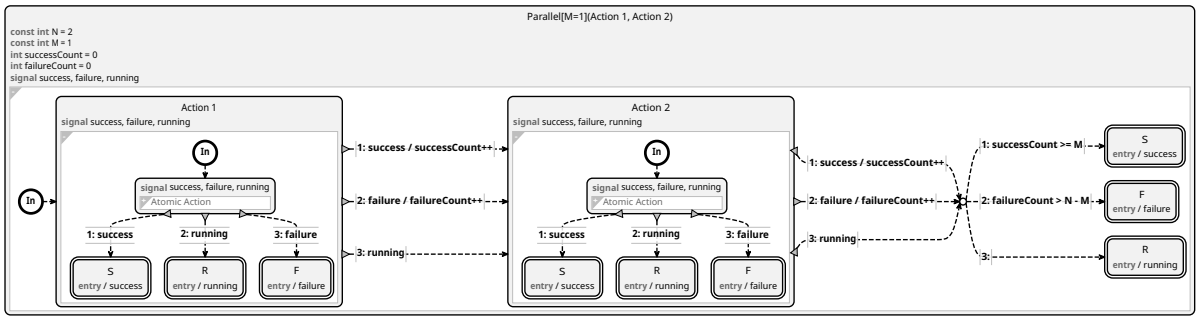



**Figure 4.10.** The parallel node from Figure 4.8 as an SCChart with variables.

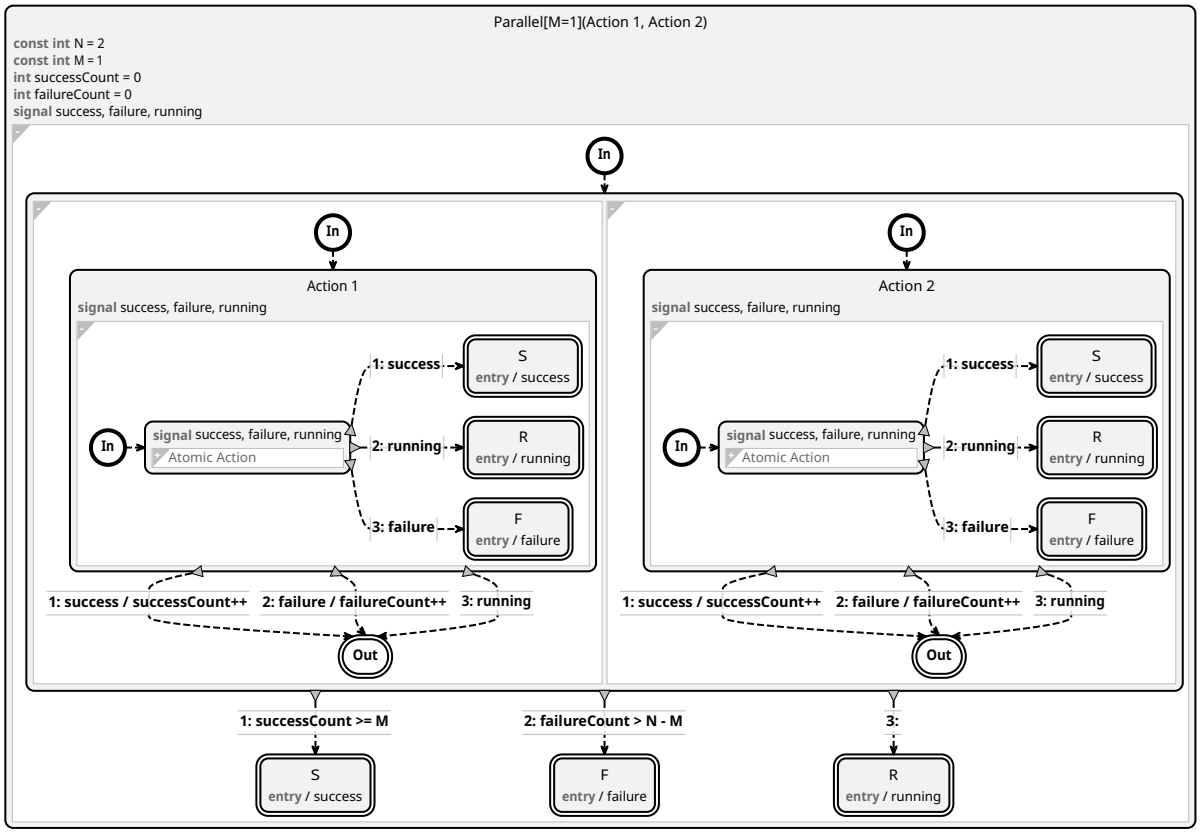



**Figure 4.11.** The parallel node from Figure 4.8 as an SCChart with parallel regions.

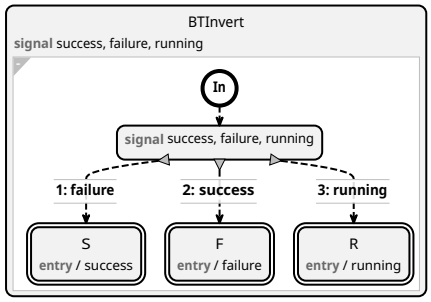



**Figure 4.12.** The inverter decorator as an SCChart.

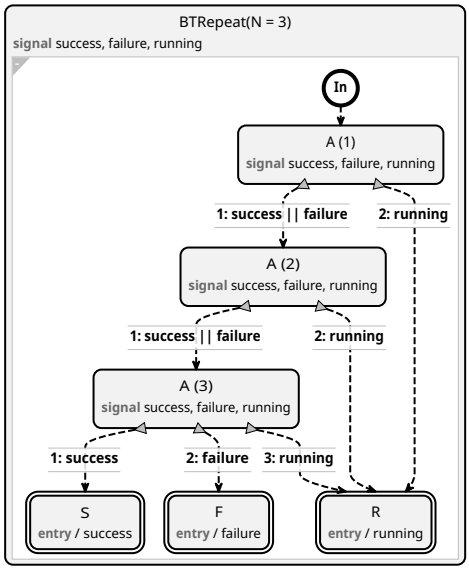



**Figure 4.13.** The repeat $N$ times decorator as an SCChart with $N = 3$.

### 4.4.2 Decorators

The translation of decorator nodes depends on the type of decorator. Remapping the output types of a child node is straightforward, as the inverter decorator in Figure 4.12 illustrates. Changing the number of times a child node is ticked is not as trivial, because deterministic state machines do not work with immediate loops. If the number of times a child node is ticked is known at compile time, the child node can be duplicated and the output states of the child node can be mapped to the output states of the decorator. This is shown in Figure 4.13. The output of the child node can be changed by mapping the output states of the child node to the output states of the decorator.

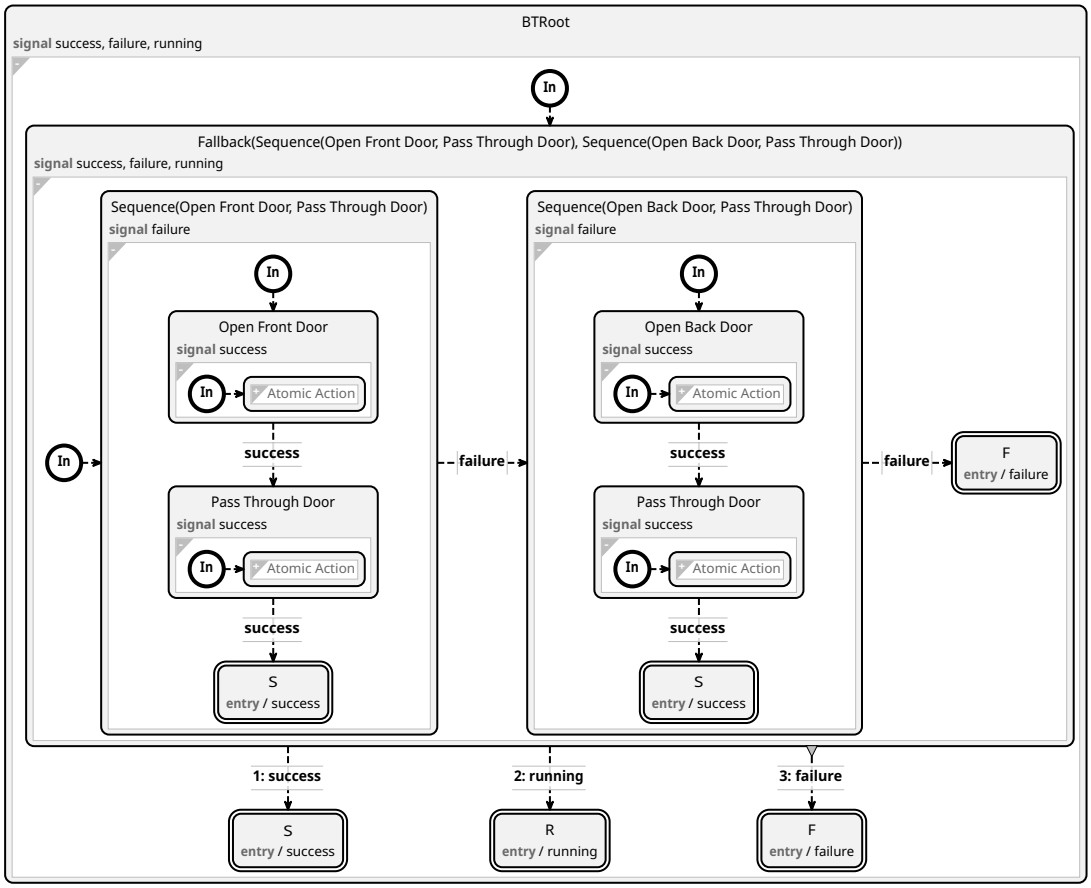



**Figure 4.14.** The Behavior Tree from Figure 4.6 as an SCChart with scoped signals.

## 4.5 Optimizing for SCCharts

The approach presented in Section 4.3 is very verbose and generates large state machines with many transitions. For example RUNNING is propagated upwards unless the parent node is a parallel node. This means that every inner state machine has to have a RUNNING state, a corresponding signal for that state and transitions to the state.

Using properly scoped signals and utilizing aborts can reduce the size of the generated SCChart and improve the readability. Using the previous example of Figure 4.7 in Figure 4.14, signals (and states) can be reduced. Signals only need to be defined in the scope in which they are relevant for a reaction. This means that the RUNNING signal is only needed in the outermost scope or in specific decorators. For children of sequence (fallback) nodes, the output signals can be scoped so that the FAILURE (SUCCESS) signal of the child is directly connected to the FAILURE (SUCCESS) signal of the parent node.

## 4.6 Detecting State in Behavior Trees

The previous sections showed how the control flow of Behavior Trees can be translated into SCCharts. The objective of this thesis is to evaluate different translations especially for utilizing the state of the Behavior Tree in the SCChart. As responsiveness to the environment is a key feature of Behavior Trees, detecting state in Behavior Trees is unintuitive and non trivial due to the lack of explicit state in Behavior Trees [Klö15]. There are different possibilities of how state could be encoded in Behavior Trees and how this state could be detected and used in the translation to SCCharts, as elaborated in the following subsections.

### 4.6.1 The State Pattern

In their work, Colledanchise and Ögren do not only explore a way to create FSMs from Behavior Trees, but also a process to perform a translation the other way around. They propose an approach where the state of the FSM is encoded by a state variable and depending on its value, the Behavior Tree executes different actions [CÖ17]. As seen in Figure 4.15, the pattern consists of a fallback node with sequence nodes as children. Each sequence nodes models a state. The sequence nodes start with a condition that checks the state variable. After that, it is checked if transitions between the states should be taken. If so, the state variable is updated. If the state variable is not updated, the actions of the state are executed. This can be abstracted into a pattern where a fallback node combined with many sequence nodes which start with conditions that check the state variable can be used to simulate a state machine. This and similar patterns can be found in many Behavior Trees (e.g. in Figure 4.2).

As the pattern is expressive, it is detectable and theoretically even possible to extract a state machine from it.

This pattern is sometimes obfuscated by the use of modular subtrees, so before detecting this pattern the Behavior Tree must be flattened. Additionally, Figure 4.16 shows that the pattern does not always use a single level of hierarchy to encode the state or explicitly switches state. Here the state condition *In Room* can be inverted and the actions *Enter Room* and *Clean Room* can be swapped to see the pattern more clearly. The state in this example is changed by the environment and the actions of the robot. Detecting transitions between states is not possible in this example.

### 4.6.2 The Memory Nodes

As discussed in Section 2.1.6, there is a Behavior Tree extension for sequence with memory nodes. In their paper, Ghzouli et al. [GBJ+23] analyze the quantity of different node types in Behavior Trees. Unfortunately, they grouped memory nodes with their memoryless counterparts and did not provide a detailed analysis of the usage of memory, but analyzing their publicly available dataset[3] shows that 30% of the examined Behavior Trees used memory nodes to encode state. Additionally, some frameworks only provide memory nodes to store

[3] https://bitbucket.org/easelab/behaviortrees

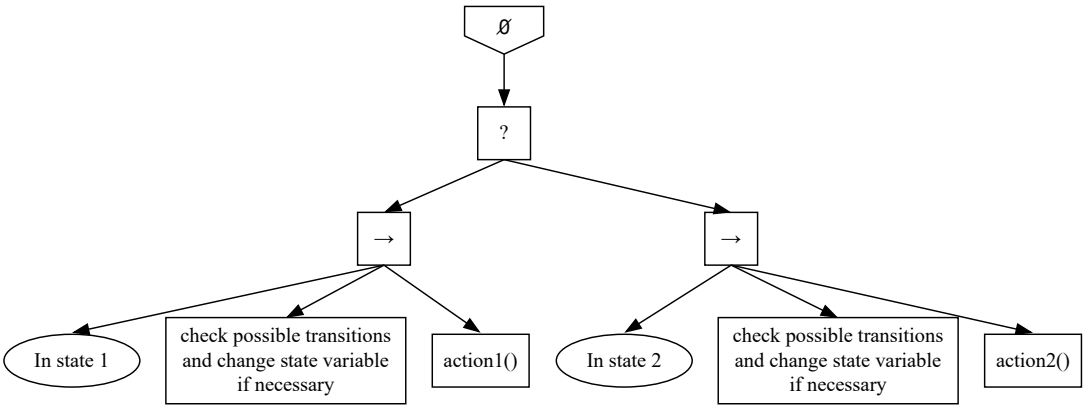



**Figure 4.15.** The state pattern in a Behavior Tree. Based on [CÖ17, Fig. 2.13].

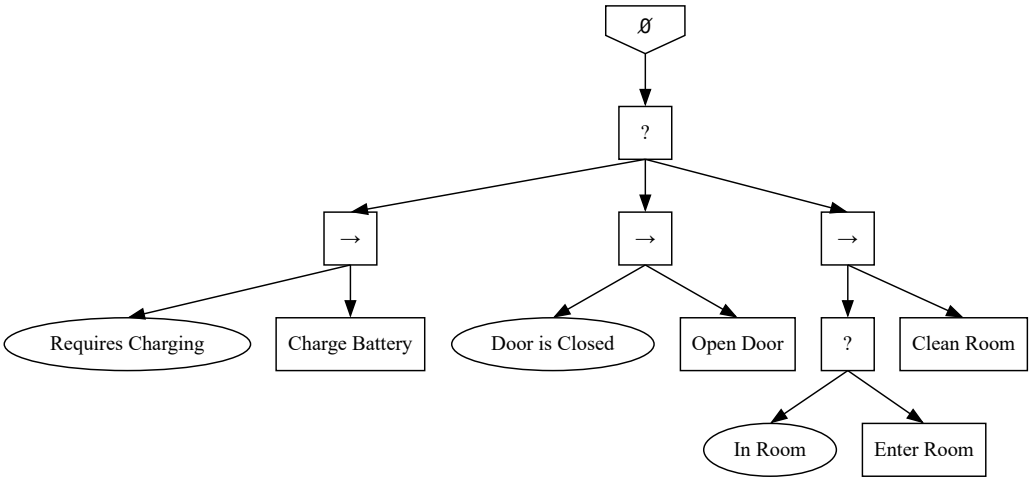



**Figure 4.16.** A Behavior Tree where the state is changed by the environment and the actions of the robot. The state is not explicitly encoded in the Behavior Tree.

state or do this by default. This implies that a non-negligible amount of Behavior Trees use memory nodes to explicitly store state. As the state is stored in the memory nodes, it is possible to extract the state from the Behavior Tree and potentially use it in the translation to the SCChart.

Additionaly there are some decorators like the "Run until Success" decorator, where the state can also be translated directly to the SCChart. It is important to note that there might be other decorators that store state internally, as they are custom for different Behavior Trees and therefore it is not possible to generalize this state extraction method.

The attached python script converts the memory nodes to normal sequence nodes and introduces state variables (as described in Section 2.1.6) to keep track of the state. This solution adds many states and transitions to the SCChart and is therefore not optimal.

A different approach can be seen in Figure 4.17. The sequence with memory is encoded as

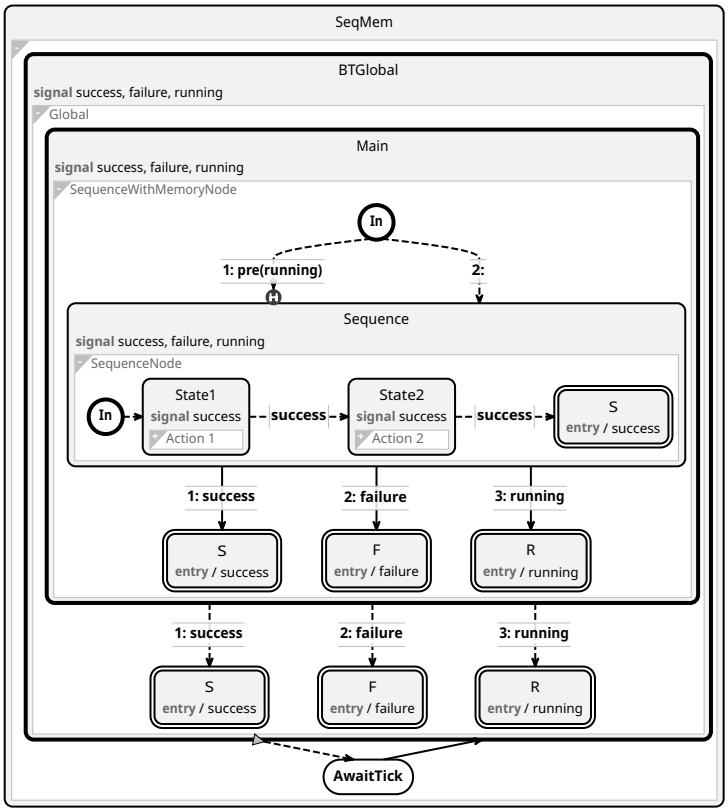



**Figure 4.17.** The sequence with memory node from Figure 2.2a as an SCChart with shallow history transitions.

a wrapper around the sequence node. If the sequence node terminated with RUNNING in the previous tick, the sequence with memory node uses a shallow history transition to continue with the last child node in the sequence. This approach is more readable but did not compile successfully during testing.

### 4.6.3 States on the Blackboard

Memory nodes can be substituted by using only pure Behavior Tree nodes and storing information on the blackboard. This is the most expressive way to store state in Behavior Trees [BZS21], but is also very hard to detect as the blackboard might contain all kinds of information.

This can be seen in Figure 4.18. The Behavior Tree models a robot that takes an order, prepares it and handles the payment. These steps will always be executed in the same order. The robot stores the current step on the blackboard in the `orderStep` variable. The `productId` is only an identifier and does not encode a sensible state. It is not trivial and in general not possible to detect which variables encode state and which do not, as systems can use the blackboard for many different purposes.

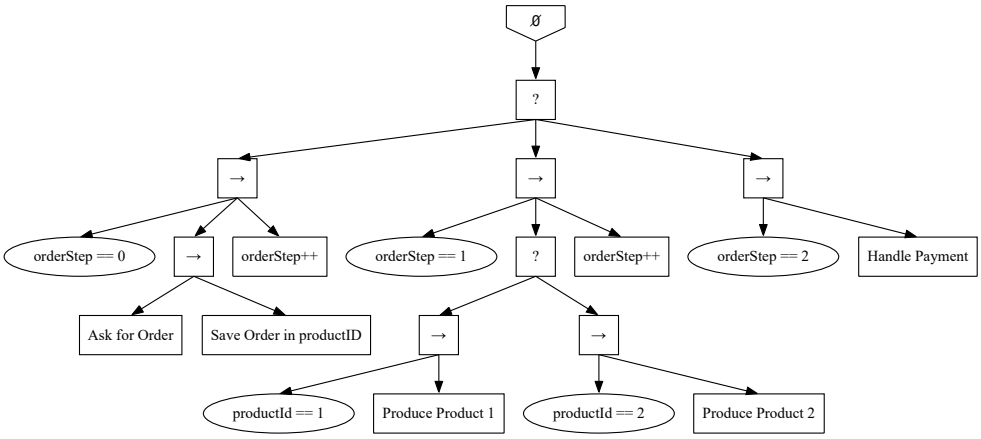



**Figure 4.18.** A Behavior Tree with a blackboard where `orderStep` encodes a sensible state while `productId` is only an identifier.

A solution to this problem could be a special node or a decorator that marks variables on the blackboard as state variables and transition actions as state transitions. It is to be noted that this would only work for specific Behavior Trees and would decrease the responsiveness to changes in the environment.

### 4.6.4 State in Actions

As Behavior Trees are used to control the actions of a system on a higher abstraction level, the state of the system is often encoded in the actions themselves. This means that the Behavior Tree is only used to determine which action to take next and the actions keep track of the progress that has been made. The Behavior Tree in Figure 4.19 shows an example of this. The Behavior Tree models a pacman Artificial Intelligence (AI). The actions are on a high level of abstraction. The Behavior Tree includes no information about the environment around the pacman, and keeps track of the state of the game in the actions, as this is encoded in the actions themselves. The Behavior Tree only determines which action to take.

Detecting this state by only analyzing the Behavior Tree is not possible, but when the actions are SCCharts themselves, the state of the system would be directly encoded in the translated SCChart.

## 4.7 Ideas for Other Approaches

This section explores other ideas for translating Behavior Trees to SCCharts. These ideas are not as well elaborated as the previous ones and are therefore only briefly discussed.

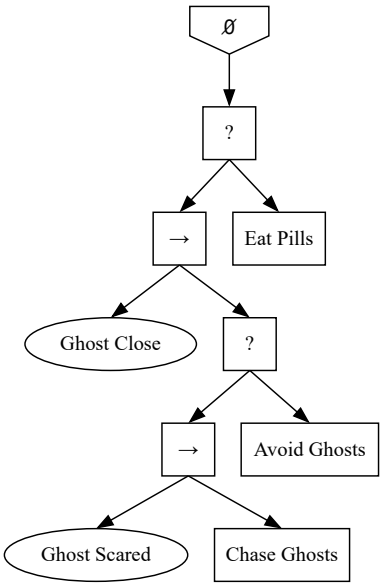



**Figure 4.19.** A Behavior Tree for a pacman game. The state of the game is encoded in the actions. Based on [CÖ17, Fig. 1.11].

### 4.7.1 Semantically Different RUNNING Actions

If a child of the sequence or fallback nodes returns RUNNING, the signal will be propagated to the parent node until it reaches the root or a parallel node. Under the assumption that a node that did not terminate with SUCCESS or FAILURE should be ticked again in the next tick, nodes that return RUNNING could be viewed as states that the program stays in until proper termination. This could result in more efficient SCCharts, but makes it a lot harder to detect the different reasons why a node should not be ticked again but aborted. This would therefore only work for specific Behavior Trees.

### 4.7.2 Potential for Dataflow Integration

The approach in Section 4.3 depends purely on immediate transitions between states. This does not make use of the advantages of an HSM, but is very similar to data flow diagrams. As SCCharts support data flow, building the data flow diagram of the Behavior Tree is a viable approach to translate Behavior Trees to SCCharts. This was already done in lingua franca [SAC+24]. As this work focuses on the relationship between state machines and Behavior Trees, this approach has not been further investigated.

Chapter 5

# Evaluation

This chapter evaluates the SCChart representation of Behavior Trees by discussing utilization of SCChart features, responsiveness to the environment, modularity, and readability. The aim is to explore the strengths and limitations of translating Behavior Trees into SCCharts while considering the trade-offs between semantic equivalence and model complexity.

## 5.1 Challenges in State Detection

Extracting states from Behavior Trees to create SCCharts poses several challenges. In Behavior Trees, states are often hidden within the action nodes. Additionally, responding to unexpected events in the environment requires multiple actions to be executed in a single tick. This requires the use of immediate transitions in the SCChart model to ensure that the system promptly reacts to changes in the environment. As a result, the proposed strategy does not make use of non-transient states. Unlike traditional FSMs and SCCharts, where states retain memory across ticks, all states in the generated SCChart model are transient. This means that the state machine does not maintain any persistent internal state across execution cycles. Each state is exited after a single tick, and the system transitions immediately to the next state based on the environment.

This work shows that detecting states of memory nodes is straightforward, as they are explicitly defined in the SCChart model. However, utilizing this information in combination with the otherwise non-transient states remains a challenge.

## 5.2 Modularity and Reuse

One of the key strengths of Behavior Trees is their inherent modularity. Subtrees in a Behavior Tree can be designed and tested independently, allowing for easy reuse and scalability in complex systems. This property is preserved in the SCChart representation, as the hierarchy of the Behavior Tree is directly translated into the hierarchy of SCCharts. This is not only beneficial for the development process and the maintainability of the system, but it also improves the performancy of the compilation process [Lüd21].

## 5.3 Readability and Visual Complexity

Despite the advantages of modularity, the generated SCCharts tend to suffer from readability issues. As more transitions are added to preserve responsiveness and transient states are used for every node, the visual complexity of the model increases. A large number of transitions and states make the SCChart difficult to comprehend, especially in cases where multiple environmental conditions have to be handled within a short span of time. This can lead to a cluttered and confusing model, which may hinder the understanding of the system's behavior.

## 5.4 Semantical Equivalence

The proposed translation strategy aims to preserve the semantics of the Behavior Tree in the generated SCChart model, as the strategies for the different node types are designed to reflect the control flow of the original Behavior Tree. Parallel nodes would benefit from the use of parallel regions as their semantics are well defined in SCCharts.

Chapter 6

# Conclusion & Future Work

This chapter presents a summary of the research conducted in this thesis and provides directions for future work.

## 6.1 Summary

The SCChart representation of Behavior Trees offers a promising method for combining the reactive nature of Behavior Trees with the formal structure of state machines. Extracting state from Behavior Trees is inherently challenging due to the Behavior Tree philosophy of avoiding internal states to maximize responsiveness to environmental changes. Despite this challenge, Behavior Trees can be translated directly into Sequentially Constructive Statecharts, preserving key behavioral properties such as modularity and semantic equivalence.

Using Behavior Trees in Sequentially Constructive Statecharts enables high-level abstraction of behavior while allowing for detailed implementation of actions within state charts. This combination provides a powerful tool for modeling systems that require both high reactivity and formal state management. However, this approach comes with trade-offs, notably the use of transient states and a high number of transitions, which can lead to significant visual complexity.

Although Sequentially Constructive Statecharts preserve the modularity and responsiveness of Behavior Trees, managing the resulting models can become challenging due to their complexity. Future work could mitigate these issues by incorporating dataflow mechanisms and exploring new visualization techniques to improve scalability and readability.

## 6.2 Future Work

Future research will focus on several areas to further enhance the integration of Behavior Trees with SCCharts. One key direction is the development of a Domain-Specific Language (DSL) for Behavior Trees that compiles directly into SCCharts. This would streamline the process of generating SCCharts models from Behavior Trees and facilitate their integration into the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) framework. Such a DSL could provide an abstraction layer that simplifies the modeling and implementation process, making the combined use of Behavior Trees and SCCharts more accessible and scalable.

Additionally, the use of blackboard variables, which are commonly employed in Behavior Trees, presents an opportunity for improved information sharing within SCCharts. By exploring

how blackboard variables can be effectively incorporated into SCChart models, future work can enable more efficient handling of shared memory between states, ensuring better modularity and controlled access to state information.

Finally, the implementation of these proposed approaches in the KIELER framework will be a significant step forward. This will allow for the practical evaluation of the methods discussed, testing their efficiency and scalability in real-world applications. The integration of these tools into KIELER will provide a practical foundation for further experimentation and refinement of the combination of Behavior Trees and SCCharts.

Appendix A

# Proof of Concept: Python Script

```
from typing import Optional, Tuple, Set, List
import xml.etree.ElementTree as ET
import sys

# Global variable to indicate if the environment interface has actions
#  (if it provides <action>Success, <action>Failure, <action>Running signals)
interface_has_actions = False


def sequence_node(
    children: List[ET.Element],
    listens_to: str = 'success, failure, running',
    imports: Optional[Set[str]] = None,
    state_vars: Optional[Set[str]] = None
) -> Tuple[str, Optional[Set[str]], Optional[Set[str]]]:
    """
    Generates SCChart code for a sequence node with multiple children.

    Args:
        children (List[ET.Element]): The child nodes of the sequence node.
        listens_to (str): Signals the node listens to
            (default: 'success, failure, running').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
        Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
            generated SCChart code, imports, and state variables.
    """
    out = f'''
/* Sequence node with {len(children)} children */
signal {listens_to}
```

```
region SequenceNode {{
    initial state In
    immediate go to {'State1' if len(children) else 'S'}

    final state S {{
        entry do success
    }}
'''

    for i, child in enumerate(children):
        parsed_tree, imports, state_vars = parse_tree(
            child, 'success', imports, state_vars)
        out += f'''
    state State{i + 1} {{
        {parsed_tree}
    }}
    immediate if success go to {
        'State' + str(i + 2) if i + 1 < len(children) else 'S'
    }
'''

    out += '''
}
'''

    return out, imports, state_vars


def fallback_node(children: List[ET.Element],
    listens_to: str = 'success, failure, running',
    imports: Optional[Set[str]] = None,
    state_vars: Optional[Set[str]] = None
) -> Tuple[str, Optional[Set[str]], Optional[Set[str]]]:
    """
    Generates SCChart code for a fallback node with multiple children.

    Args:
        children (List[ET.Element]): The child nodes of the fallback node.
        listens_to (str): Signals the node listens to
            (default: 'success, failure, running').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
```

```
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
        Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
            generated SCChart code, imports, and state variables.
    """
    out = f'''
/* Fallback node with {len(children)} children */
signal {listens_to}

region FallbackNode {{
    initial state In
    immediate go to {'State1' if len(children) else 'F'}

    final state F {{
        entry do failure
    }}
'''

    for i, child in enumerate(children):
        parsed_tree, imports, state_vars = parse_tree(
            child, 'failure', imports, state_vars)
        out += f'''
    state State{i + 1} {{
        {parsed_tree}
    }}
    immediate if failure go to {
        'State' + str(i + 2) if i + 1 < len(children) else 'F'
    }
'''

    out += '''
}
'''
    return out, imports, state_vars


def parallel_node(
    children: List[ET.Element],
    failure_count: int,
    success_count: int,
```

```
    listens_to: str = 'success, failure, running',
    imports: Optional[Set[str]] = None,
    state_vars: Optional[Set[str]] = None
) -> Tuple[str, Optional[Set[str]], Optional[Set[str]]]:
    """
    Generates SCChart code for a parallel node with multiple children.

    Args:
        children (List[ET.Element]): The child nodes of the parallel node.
        failure_count (int): Number of failures allowed before the parallel
            node fails.
        success_count (int): Number of successes required for the parallel
            node to succeed.
        listens_to (str): Signals the node listens to
            (default: 'success, failure, running').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
        Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
            generated SCChart code, imports, and state variables.
    """
    if failure_count < 0:
        failure_count = len(children) + failure_count
    if success_count < 0:
        success_count = len(children) + success_count

    out = f'''
/* Parallel node with {len(children)} children */
signal {listens_to}

int maxFailureCount = {failure_count}
int maxSuccessCount = {success_count}

region ParallelNode {{
    initial state In {{
        signal running

        int failureCount = 0
        int successCount = 0
'''
```

```
    for i, child in enumerate(children):
        parsed_tree, imports, state_vars = parse_tree(
            child, 'success, failure, running', imports, state_vars)
        out += f'''
region Region{i + 1} {{
    initial state In {{
        {parsed_tree}
    }}
    immediate if success do successCount++ join to Out
    immediate if failure do failureCount++ join to Out
    immediate if running join to Out

    final state Out
}}
'''
    out += '''
    }
    immediate if failureCount >= maxFailureCount join to F
    immediate if successCount >= maxSuccessCount join to S
    immediate join to R

    final state S {{
        entry do success
    }}

    final state F {{
        entry do failure
    }}

    final state R {{
        entry do running
    }}
}
'''
    return out, imports, state_vars


def action_node(
    code: str,
    listens_to: str = 'success',
```

```
    imports: Optional[Set[str]] = None,
    state_vars: Optional[Set[str]] = None
) -> Tuple[str, Optional[Set[str]], Optional[Set[str]]]:
    """
    Generates SCChart code for an action node.

    Args:
        code (str): The action code to execute.
        listens_to (str): Signals the node listens to (default: 'success').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
        Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
            generated SCChart code, imports, and state variables.
    """
    if interface_has_actions:
        return f'''
/* Action node */
signal {listens_to}

region ActionNode {{
    initial state In
    immediate do {code} go to C

    connector state C
    immediate if {code}Success go to S
    immediate if {code}Failure go to F
    immediate if {code}Running go to R
    immediate go to F

    final state S {{
        entry do success
    }}

    final state F {{
        entry do failure
    }}

    final state R {{
        entry do running
```

```
    }}
}}
''', imports, state_vars
    else:
        return f'''
/* Action node */
signal {listens_to}

region ActionNode {{
    initial state In
    immediate do {code} go to S

    final state S {{
        entry do success
    }}
}}
''', imports, state_vars


def condition_node(
    condition: str,
    listens_to: str = 'success, failure',
    imports: Optional[Set[str]] = None,
    state_vars: Optional[Set[str]] = None
) -> Tuple[str, Optional[Set[str]], Optional[Set[str]]]:
    """
    Generates SCChart code for a condition node.

    Args:
        condition (str): The condition to check.
        listens_to (str): Signals the node listens to
            (default: 'success, failure').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
        Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
            generated SCChart code, imports, and state variables.
    """
    return f'''
/* Condition node */
```

```
signal {listens_to}

region ConditionNode {{

    initial state In
    immediate if {condition} go to S
    immediate go to F

    final state S {{
        entry do success
    }}

    final state F {{
        entry do failure
    }}
}}
''', imports, state_vars


def sctx_node(import_name: str,
              chart_name: str,
              mapping: str,
              listens_to: str = 'success, failure, running',
              imports: Optional[set] = None,
              state_vars: Optional[set] = None):
    """
    Generates SCChart code for an SCTX node (A node that embeds an external
        SCChart).

    Args:
        import_name (str): Name of the imported module.
        chart_name (str): Name of the embedded SCChart.
        mapping (str): Mapping of signals between the parent and embedded
            SCChart.
        listens_to (str): Signals the node listens to
            (default: 'success, failure, running').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
        Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
```

```
            generated SCChart code, imports, and state variables.
    """

    listens_to_mapping = ''.join([f', {sig} to {sig.upper()}' for sig in [
                                ’success’, ’failure’, ’running’]])
    if imports:
        imports.add(import_name)
    else:
        imports = set([import_name])

    state_var = chart_name[0].lower() + f’{chart_name}CurrentState’[1:]
    if state_vars:
        state_vars.add(state_var)
    else:
        state_vars = set([state_var])

    state_var_mapping = f’, {state_var} to currentState’
    return '''
/* SCTX node */
signal ''' + listens_to + '''

region SCTXNode {
    initial state In
    immediate go to Action

    state Action is ''' + chart_name + \
        '''('''  + mapping + listens_to_mapping + state_var_mapping + ''')
    immediate join to Out

    final state Out
}
''', imports, state_vars


def sequence_with_memory_node(
    children: List[ET.Element],
    listens_to: str = ’success, failure, running’,
    imports: Optional[set] = None,
    state_vars: Optional[set] = None
) -> Tuple[str, Optional[set], Optional[set]]:
    """
```

```
Generates SCChart code for a sequence with memory node.
This implementation uses global variables to store the state of the sequence.

Args:
    children (List[ET.Element]): The child nodes of the sequence with
        memory node.
    listens_to (str): Signals the node listens to
        (default: 'success, failure, running').
    imports (Optional[Set[str]]): Set of imported modules (default: None).
    state_vars (Optional[Set[str]]): Set of state variables (default: None).

Returns:
    Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
        generated SCChart code, imports, and state variables.
"""

state_vars = state_vars or set()
state_var = 'statevar' + str(len(state_vars))
state_vars.add(state_var)

modified_children = []
for i, child in enumerate(children):
    fallback_element = ET.Element('Fallback')
    sequence_element = ET.Element('Sequence')
    sequence_element.extend([
        child,
        ET.Element('Script', attrib={
            'code': f'{state_var}++'
        })
    ])
    fallback_element.extend([
        ET.Element('ScriptCondition', attrib={
            'code': f'pre({state_var}) > {i}'
        }),
        sequence_element
    ])
    modified_children.append(fallback_element)
modified_children[-1][-1][-1].attrib['code'] = f'{state_var} = 0'

sequence_element = ET.Element('Sequence')
sequence_element.extend(modified_children)
```

```
    return parse_tree(sequence_element, listens_to, imports, state_vars)


def convert_bt_to_scchart(xml_file: str, output_file: str, name: str) -> None:
    """
    Converts a Behavior Tree (BT) XML file to an SCChart.

    Args:
        xml_file (str): Path to the input XML file.
        output_file (str): Path to the output SCChart file.
        name (str): Name of the SCChart.
    """
    tree = ET.parse(xml_file)
    root = tree.getroot()

    if root[0].tag == 'BehaviorTree':
        root = root[0]

    with open(output_file, 'w') as f:
        parsed_tree, imports, state_vars = parse_tree(root[0])

        imports = imports or set()
        state_vars = state_vars or set()

        f.write(
            '''import "''' + name.lower() + '''.environment"
''' + '\n'.join(f'import "{i}"' for i in imports) + '''

scchart ''' + name + '''BT extends ''' + name + '''Environment {
''' + '\n'.join(f'int {var} = 0' for var in state_vars) + '''

    initial state BTGlobal {
        signal success, failure, running

        region Global {
            initial state Main {
                ''' + parsed_tree + '''
            }
            immediate if success go to S
            immediate if failure go to F
```

```
            immediate if running go to R

            final state S {
                entry do success
            }

            final state F {
                entry do failure
            }

            final state R {
                entry do running
            }
        }
    }
    join to AwaitTick

    state AwaitTick
    go to BTGlobal
}
'''')


def parse_tree(
    node: ET.Element,
    listens_to: str = 'success, failure, running',
    imports: Optional[Set[str]] = None,
    state_vars: Optional[Set[str]] = None
) -> Tuple[str, Optional[Set[str]], Optional[Set[str]]]:
    """
    Parses an XML node representing part of a Behavior Tree (BT) and generates
        SCChart code.

    Args:
        node (ET.Element): The XML node to parse.
        listens_to (str): Signals the node listens to
            (default: 'success, failure, running').
        imports (Optional[Set[str]]): Set of imported modules (default: None).
        state_vars (Optional[Set[str]]): Set of state variables (default: None).

    Returns:
```

```
    Tuple[str, Optional[Set[str]], Optional[Set[str]]]: A string of
        generated SCChart code, imports, and state variables.
"""

imports = imports or set()
state_vars = state_vars or set()

match node.tag:
    case 'Sequence':
        return sequence_node(
            list(node),
            listens_to=listens_to,
            imports=imports,
            state_vars=state_vars
        )
    case 'Fallback':
        return fallback_node(
            list(node),
            listens_to=listens_to,
            imports=imports,
            state_vars=state_vars
        )
    case 'Parallel':
        return parallel_node(
            list(node),
            int(node.attrib['failure_count']),
            int(node.attrib['success_count']),
            listens_to=listens_to,
            imports=imports,
            state_vars=state_vars
        )
    case 'ParallelAll':
        return parallel_node(
            list(node),
            int(node.attrib['max_failures']),
            0,
            listens_to=listens_to,
            imports=imports,
            state_vars=state_vars
        )
    case 'Script':
```

```
            return action_node(
                node.attrib['code'],
                listens_to=listens_to,
                imports=imports,
                state_vars=state_vars
            )
        case 'ScriptCondition':
            return condition_node(
                node.attrib['code'],
                listens_to=listens_to,
                imports=imports,
                state_vars=state_vars
            )
        case 'AlwaysSuccess':
            return condition_node(
                'true',
                listens_to=listens_to,
                imports=imports,
                state_vars=state_vars
            )
        case 'AlwaysFailure':
            return condition_node(
                'false',
                listens_to=listens_to,
                imports=imports,
                state_vars=state_vars
            )
        case 'SCTX':
            return sctx_node(
                node.attrib['import'],
                node.attrib['chartName'],
                node.attrib['mapping'],
                listens_to=listens_to,
                imports=imports,
                state_vars=state_vars
            )
        case 'SequenceWithMemory':
            return sequence_with_memory_node(
                list(node),
                listens_to=listens_to,
                imports=imports,
```

```
                state_vars=state_vars
            )
        case _:
            raise Exception('Unknown node type: ' + node.tag)


if __name__ == '__main__':
    if len(sys.argv) != 4:
        print('Usage: python bt2scchart.py input.xml output.sctx name')
        print('\tinput.xml: the input behavior tree XML file')
        print('\toutput.sctx: the output SCChart file')
        print('\tname: the name of the SCChart')
    else:
        convert_bt_to_scchart(sys.argv[1], sys.argv[2], sys.argv[3])
```

# Bibliography


[Ahm23] Akash Ahmad. "A DSL for Behavior Trees in Lingua Franca". Bachelor's Thesis. Christian-Albrechts-Universität zu Kiel, Sept. 2023. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aah-bt.pdf.

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(92)90005-V. URL: https://www.sciencedirect.com/science/article/pii/016764239290005V.

[BZS21] Oliver Biggar, Mohammad Zamani, and Iman Shames. "An expressiveness hierarchy of Behavior Trees and related architectures". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5397–5404. DOI: 10.1109/LRA.2021.3074337.

[CN22] Michele Colledanchise and Lorenzo Natale. "Handling concurrency in Behavior Trees". In: *IEEE Transactions on Robotics* 38.4 (2022), pp. 2557–2576. DOI: 10.1109/TRO.2021.3125863.

[CÖ17] Michele Colledanchise and Petter Ögren. "Behavior Trees in robotics and AI: an introduction". In: *CoRR* abs/1709.00084 (2017). arXiv: 1709.00084. URL: http://arxiv.org/abs/1709.00084.

[Dro03] R. G. Dromey. "From requirements to design: formalizing the key steps". In: *First International Conference on Software Engineering and Formal Methods.* 2003, pp. 2–11. DOI: 10.1109/SEFM.2003.1236202.

[GBJ+20] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wąsowski. "Behavior Trees in action: a study of robotics applications". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 196–209. ISBN: 9781450381765. DOI: 10.1145/3426425.3426942. URL: https://doi.org/10.1145/3426425.3426942.

[GBJ+23] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. "Behavior Trees and state machines in robotics applications". In: *IEEE Transactions on Software Engineering* 49.9 (2023), pp. 4243–4267. DOI: 10.1109/TSE.2023.3269081.

[IFF+24] Matteo Iovino, Julian Förster, Pietro Falco, Jen Jen Chung, Roland Siegwart, and Christian Smith. *Comparison between Behavior Trees and finite state machines*. 2024. arXiv: 2405.16137 [cs.RO]. URL: https://arxiv.org/abs/2405.16137.

[Klö15] Andreas Klöckner. "Behavior Trees with stateful tasks". In: *Advances in Aerospace Guidance, Navigation and Control*. Ed. by Joël Bordeneuve-Guibé, Antoine Drouin, and Clément Roos. Cham: Springer International Publishing, 2015, pp. 509–519. ISBN: 978-3-319-17518-8.

[KMW+12] Soon-Kyeong Kim, Toby Myers, Marc-Florian Wendland, and Peter A. Lindsay. "Execution of natural language requirements using state machines synthesised from Behavior Trees". In: *Journal of Systems and Software* 85.11 (2012), pp. 2652–2664. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.06.013. URL: https://www.sciencedirect.com/science/article/pii/S0164121212001690.

[Lüd21] Gavin Lüdemann. "Modular code generation for SCCharts". Bachelor's Thesis. Christian-Albrechts-Universität zu Kiel, Sept. 2021. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/glu-bt.pdf.

[MCS+14] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. "Towards a unified Behavior Trees framework for robot control". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 5420–5427. DOI: 10.1109/ICRA.2014.6907656.

[Mot17] Christian Motika. *SCCharts – language and interactive incremental compilation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Kiel University, Germany. Department of Computer Science, 2017. ISBN: 9783746009391. DOI: 10.21941/kcss/2017/02.

[Ögr] Petter Ögren. "Increasing modularity of UAV control systems using computer game Behavior Trees". In: *AIAA Guidance, Navigation, and Control Conference*. DOI: 10.2514/6.2012-4458. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2012-4458. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2012-4458.

[SAC+24] Alexander Schulz-Rosengarten, Akash Ahmad, Malte Clement, Reinhard von Hanxleden, Benjamin Asch, Marten Lohstroh, Edward A. Lee, Gustavo Quiros Araya, and Ankit Shukla. *Behavior Trees with dataflow: coordinating reactive tasks in Lingua Franca*. 2024. arXiv: 2401.09185 [cs.PL]. URL: https://arxiv.org/abs/2401.09185.

[SMA+23] Alexander Schulz-Rosengarten, Michael Mendler, Joaquin Aguado, Malte Clement, and Reinhard von Hanxleden. *Trapping Behavior Trees in Esterel*. 2023. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/fdl23.pdf.

# List of Abbreviations

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **BT** | Behavior Tree |
| **DSL** | Domain-Specific Language |
| **FSM** | Finite State Machine |
| **HSM** | Hierarchical State Machine |
| **KIELER** | Kiel Integrated Environment for Layout Eclipse Rich Client |
| **SC MoC** | Sequentially Constructive Model of Computation |
| **SCChart** | Sequentially Constructive Statechart |
| **XML** | Extensible Markup Language |