Verification of Behavior Trees using Linear Constrained Horn Clauses

Thomas Henn[1](✉), Marcus Völker[1], Stefan Kowalewski[1], Minh Trinh[2], Oliver Petrovic[2], and Christian Brecher[2]

[1] Informatik 11 - Embedded Software, RWTH Aachen University, Aachen, Germany
{henn,voelker,kowalewski}@embedded.rwth-aachen.de

[2] Laboratory for Machine Tools and Production Engineering, RWTH Aachen University, Aachen, Germany
{m.trinh,o.petrovic,c.brecher}@wzl.rwth-aachen.de

Abstract. In the field of industrial production the usage of Behavior Trees sparks interest due to their modularity and flexibility. Considering Behavior Trees are used in a safety-critical domain, there is increased interest for methods to verify a Behavior Tree's safety. Current approaches for Behavior Trees are only semi-automatic since they require manually added low-level details about the action's behavior.

In this paper, we describe an automatic verification method for safety properties on Behavior Trees using Linear Constrained Horn Clauses (LCHCs). Our approach encodes all components of the verification task as CHCs, that is, the structure and semantics of the Behavior Tree, the implemented actions in the leaf nodes and the safety property itself. These clauses are then solved by a state-of-the-art SMT solver, leading to an efficient algorithm for Behavior Tree verification, which we evaluate by comparing our method against a general purpose verification framework.

Keywords: behavior tree · formal verification · constrained horn clauses · software safety

1 Introduction

Behavior Trees describe the executions of agents and systems. One of the major advantages is their modularity [10]. Complex tasks are composed of simpler tasks, without further knowledge about the implementation of the simple tasks, since all nodes share a common interface. This advantage and the visualization of Behavior Trees (e.g., see Fig. 1) contribute to their popularity and helps to design, develop, and test Behavior Trees. At first, Behavior Trees were used to characterize the behavior of non-player characters (NPCs) in video games

[14]. Since then, other communities, like the robotics [5,6,12,13] and artificial intelligence communities [9,11], have used Behavior Trees to model their agents.

This usage of Behavior Trees in safety-critical environments leads to an increasing interest in the application of formal methods on Behavior Trees. However, the clear and intuitive graphical representation of Behavior Trees is achieved by defining the control flow implicitly. Therefore, execution paths can easily be overlooked.

Previous work focuses mostly on defining a clear syntax and semantics, since no common standard, for representing Behavior Trees, exists [2–4]. A first approach to verify Behavior Trees looks promising, but still needs additional input in form of logical formulas from the user about the low-level behavior [1].

In this paper, our contribution is the demonstration of a viable approach for an automatic (i.e., no further input about the Behavior Tree is required) verification of Behavior Trees. Our approach is based on a logical encoding of the Behavior Tree's semantics. We utilize Linear Constrained Horn Clauses (LCHCs), because solving LCHCs has been proven to be efficient [7] and their successful usage in software verification has been presented in [8,9].

The paper is structured as follows: Sect. 2 shows the current state-of-the-art concerning the verification and analysis of Behavior Trees. In Sect. 3, we introduce the notion of a Behavior Tree and Constrained Horn Clauses. In Sect. 4, we present the encoding of Behavior Trees using CHCs. The presented approach is then evaluated on several verification tasks and compared to a general purpose verification framework in Sect. 5. We finish with a summary and outlook in Sect. 6.

2 Related Works

In several other works, alternative encodings of Behavior Trees which could be used for verification purposes, are presented. In [2], the authors show how Behavior Trees can be encoded as Communicating Sequential Processes (CSP). The motivation behind this work is to provide a more precise formalization for Behavior Trees since there is no standardized formalism or rigorous semantics for Behavior Trees. CSPs are no intended to be used as a control architecture but is suited for verifying and specifying concurrent systems. To use the CSP formalism for verification purposes was left for future work.

Another approach which encodes Behavior Trees in a description logic is presented in [3]. The shown encoding in description logic could be utilized for a runtime verification that checks whether a proper action is executed. The extension of the approach was left for future work as well as the verification whether a given Behavior Tree is guaranteed to execute successfully.

In [21] another approach for runtime monitoring is presented. Behavior Trees are translated into a communication channel system. The, in the paper, introduced Behavior Trees only model a subset of the "classical" Behavior Trees since parallel nodes are omitted. Also the environment is not part of the formal model and therefore the properties can only be analyzed in a simulation or in a real world scenario.

The authors of [4] present an correct-by-construction approach. Linear Temporal Logic (LTL) formulas are used to define the correct behavior which the synthesized Behavior Tree has to exhibit. The approach does not allow the verification of already existing Behavior Trees.

In [1] a verification approach is presented which is based on the transformation of (sub-)trees to a collection of LTL formulas. These constructed LTL formulas, representing the semantics of the Behavior Tree, are then checked against properties encoded as other LTL formulas. The need of LTL formulas, given by the user, which describe the semantics of the action and condition nodes, prevent an automatic usage of the verification algorithm. The necessary level of detail differs from property to property and should be adapted for every verification run.

3 Preliminaries

In Sect. 3.1, we introduce the structure as well as the semantics of behaviour trees. Afterwards, we give a short introduction to Constrained Horn Clauses in Sect. 3.2.

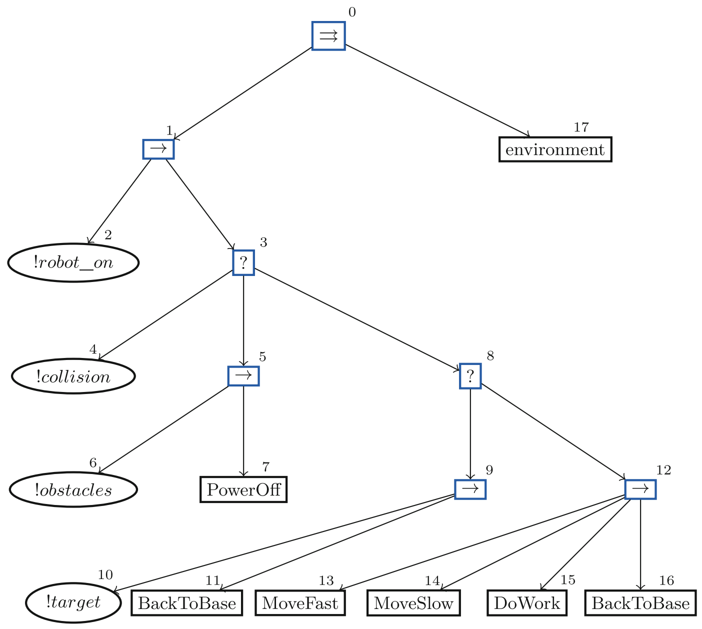

Fig. 1. Example behavior tree with collision and obstacle detection.

3.1 Behavior Trees

A Behavior Tree, as depicted in Fig. 1, is a directed acyclic graph with a distinguished root node [10]. It describes the control flow between the possible actions. All nodes in a Behavior Tree have the same interface when it comes to the execution. A node starts its execution when it receives a *tick*. Each node returns one of the following three statuses: `SUCCESS`, `FAILURE` and `RUNNING`. The status indicates that the subtree performed its task successfully, unsuccessfully or that the task is still in execution, respectively. The possibility, to return `RUNNING`, shows that Behavior Trees are not an extension of (hierarchical) finite state machines since a Behavior Tree does not stay in a node until the execution is complete. The whole tree is executed by ticking the root node. Usually, this is done in an infinite cycle, i.e. the root node is ticked again as soon as it returns `SUCCESS`, `RUNNING` or `FAILURE`.

The leaves encode *actions* (drawn as a box) and *conditions* (drawn as a ellipse). Condition nodes only return `SUCCESS` or `FAILURE` and check the condition of the system, described by the Behavior Tree, or the environment. They also have no side effects (i.e. do not alter variables or the state of the system). Ticking an action node corresponds to a function call which triggers the action to be performed. If the execution of the action is not finished, `RUNNING` is returned. Otherwise the successful or failed execution is reported to the parent node.

The inner nodes, also called composite nodes, of a Behavior Tree are responsible for the control flow. Based on the returned status of their subtrees, they decide which subtrees to tick next or to return a status to their parent. The children of an inner nodes are ordered from left to right (i.e. the first child is depicted as the leftmost child). A *sequence* node (represented by $\rightarrow$) executes is children consecutively. When a sequence node is ticked it propagates the tick to the first child. If a child returns `SUCCESS` the next child is ticked. If the child is the last one the sequence node returns `SUCCESS` instead since the whole sequence was executed successfully. Whenever a child returns `FAILURE` or `RUNNING` the sequence node stops ticking the other child nodes and returns `FAILURE` or `RUNNING`, respectively.

Complementary to the sequence node is the *selector* node (represented by ?). A selector node also executes its children from left to right, but stops the ticking of other child nodes whenever a child returns `SUCCESS` or `RUNNING` and returns the same value to its parent. If a child returns `FAILURE` the next child in order is ticked or `FAILURE` is returned from the selector node if the child is the last one. Selector and sequence node have the same behavior; only the roles of `SUCCESS` and `FAILURE` are switched.

The third type of composite node is the *parallel* node (represented by $\rightrightarrows$). A parallel node executes its children in parallel and returns a value based on the accumulated return values of its children. Parallel nodes are parametrized with a variable $m \in \mathbb{N}$ which is less than or equal to the number of children. `SUCCESS` is returned when at least m children finished their execution with `SUCCESS`. If $n - m + 1$ children returned `FAILURE` the parallel node returns `FAILURE`. In all other cases `RUNNING` is returned.

Behavior Trees composed of these node types are sometimes called *classic* Behavior Trees. In practice there are custom nodes and extensions since no standard exists. The idea, that sequences do not necessarily start from the beginning, but from a child which returned `RUNNING`, is incorporated in *sequence with memory* nodes (represented by $\rightarrow_m$). These nodes behaves similar to the regular sequence nodes except that if the last value returned was `RUNNING`, the corresponding child is ticked instead of the first child when the sequence with memory node is ticked again. The same extension exists for selector nodes which are called *selector with memory* (represented by $?_m$).

The BT in Fig. 1 corresponds to assembly robot which performs at task (DoWork) if no collisions & obstacles are detected and a target is selected.

3.2 Constrained Horn Clauses

Constrained Horn Clauses (CHCs) are a structure for clauses from a first-order logic [8]. Given sets of predicates $\mathcal{P}$, functions $\mathcal{F}$, and variables $\mathcal{V}$ a Constrained Horn Clause is defined as formula of the following structure:

$$\forall \mathcal{V}.\mathbf{p_1}(\overrightarrow{X_1}) \wedge \cdots \wedge \mathbf{p_k}(\overrightarrow{X_k}) \wedge \varphi \rightarrow \mathbf{h}(\overrightarrow{X}), \quad k \geq 0 \tag{1}$$

where $\mathbf{p}$ are predicates, $\overrightarrow{X_i} \subseteq \mathcal{V}$ are subsets of variables, φ is a quantifier-free formula over $\mathcal{X}$ and $\mathcal{F}$, and $\mathbf{h}$ can be either a predicate or a quantifier-free formula. A Constrained Horn Clause is called linear if $k \leq 1$.

A set of CHCs is satisfiable when there exist an interpretation of all predicates such that all implications hold. Since all variables are universal quantified, we omit the quantifier and in the style of logic programming languages we replace $\wedge$ by comma and reverse the implication:

$$\mathbf{h}(\overrightarrow{X}) \leftarrow \mathbf{p_1}(\overrightarrow{X_1}), \cdots, \mathbf{p_k}(\overrightarrow{X_k}), \varphi \tag{2}$$

4 Encoding of Behavior Trees

In this section, we present our approach how the Behavior Trees semantics can be encoded in linear Constrained Horn Clauses. Section 4.1 explains the general idea and introduces a common interface and some auxiliary definitions to simplify further explanations. The following sections propose how every node type can be encoded using only the knowledge of their direct children which creates a logical representation of the Behavior Tree which is as modular and flexible as the Behavior Tree itself. After we presented the encoding of the Behavior Tree, we show how safety properties and the environment is transformed in linear Horn Clauses in Sect. 4.9 and 4.10 respectively.

4.1 Idea

The approach of encoding procedures with Constrained Horn Clauses, presented in [8,9], is based on creating uninterpreted predicates which corresponds to program locations. The SMT solver finds an over-approximation of variable valuations which are valid at these specific program locations. E.g., the interpretation of a predicate loc_1 with $x > 0$ characterizes all states at location loc_1 where x is positive.

To identify the different nodes, which could have the same type, we assign an index $i \in \mathbb{N}$ to each node where the root node always has the index 0. The number of children of node i is denoted with n_i and the index of the j-th child of node i is the result of the auxiliary function $child(i, j)$. The parameterized threshold for parallel node i is given as m_i.

We also introduce two vectors of variables X and X' where X is a vector containing all program variables as well as all variables introduced by our encoding. X' is a primed copy of the vector X which is used to distinguish variables before and after some changes. E.g., the formula $y' = y + 1$ encodes the increment of the variable y by 1.

For every node i, we add the following predicates: $tick_i(X)$, $success_i(X)$, $failure_i(X)$ and $running_i(X)$. These predicates represent the states when a node is ticked and when the node returns `SUCCESS`, `FAILURE` or `RUNNING`.

Since these four predicates exist for all nodes and the behavior of the composite nodes only depends on the return value of their children, we can use these predicates as a means to encode the semantics with CHCs.

4.2 Action Node

As mentioned before, an action node corresponds to a function in a program. These functions are represented as Control Flow Automata (CFA) which are directed graphs. The nodes (in the CFA) are called locations and the edges correspond to the instructions which are performed in order to move from one location to the next location. We omit a detailed definition of CFAs since more information can be found in the literature [20]. These CFA have four designated locations for the entry and exit. One entry location l_0 and one for each return value and exit location named $l_{success}, l_{running}, l_{failure}$.

In [20] is shown how CFAs can be encoded using Constrained Horn Clauses. We use presented approach for the encoding of CFAs: e.g., the clause $l_i(X') \leftarrow x' = x + 1, l_j(X)$ encodes the transition from location j to location i which is the labeled with `x = x + 1`.

The predicates used for the location representation need to be connected with the predicates for the action node. The semantics of an action node i are encoded by the following clauses:

$$\begin{aligned} l_0(X) &\leftarrow tick_i(X) \\ success_i(X') &\leftarrow l_{success}(X) \\ running_i(X') &\leftarrow l_{running}(X) \\ failure_i(X') &\leftarrow l_{failure}(X) \end{aligned}$$

Intuitively, the first clause states that if the action node i is ticked with the variables X the execution continues at the initial location of the corresponding

CFA. The remaining clauses propagate the state reaching one of the exit location of the CFA to the predicates of the BT.

In order to model asynchronous function calls to external libraries, we allow the use of nondeterministic values. This method is also used for modeling the environment which is explained in Sect. 4.10.

4.3 Condition Node

Condition nodes are represented by functions in the same way as action nodes. Therefore, they can be encoded in the same way as in Sect. 4.2 and we can construct predicates and clauses for the CFA of condition node i. Note, that we only have two exit locations for condition nodes, since condition nodes never return `RUNNING`.

The clauses for encoding a condition node i are the following:

$$\begin{aligned} l_0(X) &\leftarrow tick_i(X) \\ success_i(X') &\leftarrow l_{success}(X) \\ running_i(X') &\leftarrow l_{running}(X), false \\ failure_i(X') &\leftarrow l_{failure}(X) \end{aligned}$$

The boolean condition *false* encodes that $l_r(X)$ is not reachable.

4.4 Sequence Node

Clause 3 encodes the propagation of a tick from a sequence node i to its first child.

$$tick_{child(i,1)}(X) \leftarrow tick_i(X) \tag{3}$$

When a child returns `FAILURE` or `RUNNING` the value is propagated to the parent of the sequence node. Since the sequence stops its execution independent from the child node which returns `FAILURE` or `RUNNING`, we use a clause for each child to propagate the return value, as shown in clauses 4 and 5.

$$failure_i(X) \leftarrow failure_{child(i,j)}(X) \quad \forall 1 \leq j \leq n_i \tag{4}$$

$$running_i(X) \leftarrow running_{child(i,j)}(X) \quad \forall 1 \leq j \leq n_i \tag{5}$$

The successful execution of a child triggers the tick of the next child in the sequence which is encoded in the set of clauses 6. Only if the last child returns `SUCCESS` the value is propagated to the parent of the sequence node (see clause 7).

$$tick_{child(i,j+1)}(X) \leftarrow success_{child(i,j)}(X) \quad \forall 1 \leq j < n_i \tag{6}$$

$$success_i(X) \leftarrow success_{child(i,n_i)}(X) \tag{7}$$

The clauses generated for node 1 from Fig. 1 are shown in the following enumeration. Since the control flow is determined directly by the return values, there is no modification of variables.

$$\begin{aligned}
tick_2(X) &\leftarrow tick_1(X)\\
tick_3(X) &\leftarrow success_2(X)\\
running_1(X) &\leftarrow running_2(X)\\
failure_1(X) &\leftarrow failure_2(X)\\
success_1(X) &\leftarrow success_3(X)\\
running_3(X) &\leftarrow running_3(X)\\
failure_3(X) &\leftarrow failure_3(X)
\end{aligned}$$

4.5 Sequence Node with Memory

A sequence node with memory needs to keep track which of its children needs to be ticked when the sequence node itself is ticked next time. We introduce a fresh variable $next_i$ for every sequence node i with memory to store the information. The variable is initialized with the index of the first child to ensure that the first time the sequence node with memory is ticked, it starts from the beginning.

Since every child can return RUNNING, we encode the propagation of the tick with clauses 8. In contrast to clause 3, the propagation of the tick is no longer unconditional, but we enforce that the value of the variable $next_i$ is the same as the index of the child being ticked.

$$tick_{child(i,j)}(X) \leftarrow tick_i(X), next_i = j \quad \forall 1 \leq j \leq n_i \tag{8}$$

The value of $next_i$ must be set whenever a child returns RUNNING. In clauses 9 the value is changed. To prevent that other variables change their values, we use another auxiliary function id which ensures that variables keep their value if they are elements of the passed set.

$$\begin{aligned}
running_i(X') \leftarrow & running_{child(i,j)}(X), next'_i = j,\\
& id(X \backslash \{next_i\}) \quad \forall 1 \leq j \leq n_i
\end{aligned} \tag{9}$$

The clauses for FAILURE and SUCCESS must be adapted as well. The clauses 10 for the FAILURE cases are similar to the clauses 9 for RUNNING. They differ in the index which is assigned to $next_i$. For the clauses concerning the SUCCESS case, only the last one, clause 12, must be adapted in order to reset the variable $next_i$. The clauses 11 are identical to the ones for sequence nodes without memory, since they trigger the tick of the next child.

$$\begin{aligned} failure_i(X') \leftarrow\, & failure_{child(i,j)}(X), \\ & next_i = child(i,1), \\ & id(X \backslash \{next_i\}) \quad \forall 1 \leq j \leq n_i \end{aligned} \tag{10}$$

$$tick_{child(i,j+1)}(X) \leftarrow success_{child(i,j)}(X) \quad \forall 1 \leq j < n_i \tag{11}$$

$$\begin{aligned} success_i(X') \leftarrow\, & success_{child(i,n_i)}(X), \\ & next_i = child(i,1), id(X \backslash \{next_i\}) \end{aligned} \tag{12}$$

4.6 Selector Node

The selector node is complementary to the sequence node, as explained in Sect. 3.1. The Constrained Horn Clauses needed to encode the semantics for a selector node, with or without memory, are similar to the clauses for sequence nodes. We use the same clauses but switch the occurrences of `SUCCESS` and `FAILURE` in the Constrained Horn Clauses. The exact formalization is trivial and omitted in this paper.

4.7 Parallel Node

Verifying programs with concurrency and modelling interleaving semantics is challenging when using linear Constrained Horn Clauses. It also creates more complex and larger models which in turn impacts the time needed for verification. Often the precise modeling of concurrency is not necessary, depending on the properties which are to be verified. Therefore, we assume that it is sufficient to model the execution of a parallel node's children as atomic.

Similar to the encoding of sequence nodes with memory, we introduce new fresh variables to keep track of the execution status of the parallel node i. $cnt\_success_i$, $cnt\_running_i$ and $cnt\_failure_i$ are new integer variables which are used to store the amount of returned `SUCCESS`, `RUNNING` and `FAILURE` values. Also for every child j, we add a boolean variable $executed_j$ to memorize whether a child has been executed and in order to prevent that a child is ticked more than once.

For every parallel node i, we introduce a new predicate $intermediate_i(X)$ which represents all states before and after children of the parallel node are executed. The following formulas representing the different conditions when the parallel node stops executing and return either `SUCCESS`, `FAILURE` or `RUNNING`. The formula $continue_i$ evaluates to `true` when none of the the conditions are fulfilled.

$$\begin{aligned}
cond\_success_i &:= cnt\_success_i \geq m_i\\
cond\_failure_i &:= cnt\_failure_i \geq n_i - m_i + 1\\
cond\_running_i &:= (cnt\_failure_i +\\
&\quad cnt\_running_i \geq n_i - m_i + 1)\\
&\quad \wedge (cnt\_success_i + cnt\_running_i \geq m_i)\\
continue_i &:= \neg(cond\_success_i \wedge cond\_failure_i\\
&\quad \wedge cond\_running_i)
\end{aligned}$$

While propagating the tick to the predicate *intermediate*$_i(X)$, the newly introduced variables are initialized. The counter variables are set to 0 while the *executed* flag for the children is set to `false`. The other variables keep their values and for the sake of readability we omitted the argument for the *id* function.

$$\begin{aligned}
intermediate_i(X') \leftarrow\ & tick_i(X), cnt\_success'_i = 0,\\
& cnt\_running'_i = 0, cnt\_failure'_i = 0,\\
& \bigwedge_{j=1}^{n_i} executed'_{child(i,j)} = \texttt{false}, id(\dots)
\end{aligned}$$

From the *intermediate* predicate the tick is propagated to the children, when the *executed* flag is still `false` and none of the return conditions for the parallel node holds.

$$\begin{aligned}
tick_{child(i,j)}(X) \leftarrow\ & intermediate_i(X), executed_i = \texttt{false},\\
& continue_i \quad \forall 1 \leq j \leq n_i
\end{aligned}$$

In the following, we only present clauses when a child returns `SUCCESS`, clause 13, and when the parallel node returns `SUCCESS`, clause 14. The clauses for `RUNNING` and `FAILURE` are analogous. In case the child execution ends successfully, the counter *cnt_success* is incremented by one and the *executed* flag is set to `true` in order to prevent that from the *intermediate* predicate the *tick* of the child is again reachable.

$$\begin{aligned}
intermediate(X')_i \leftarrow\ & success_{child(i,j)}(X),\\
& executed'_{child(i,h)} = \texttt{true},\\
& cnt\_success'_i = cnt\_success_i + 1,\\
& id(...) \quad \forall 1 \leq j \leq n_i
\end{aligned} \tag{13}$$

Clause 14 encodes that once the success condition is fulfilled, the result is propagated to the parent.

$$success_i(X) \leftarrow intermediate_i(X), cond\_success \tag{14}$$

4.8 Root Node

The root node with index 0 can be any arbitrary node type, but there are some additional clauses which model the initialization, clause 15, and the repeatedly ticking, clauses 16.

The variables used in the action nodes as well as the variables we introduced for the encoding must be initialized. The initialization can be interpreted as a sequence of assignment to variables. These assignments can be encoded in a formula *init*.

$$tick_0(X) \leftarrow init \tag{15}$$

The complete Behavior Tree is usually ticked repeatedly. A Behavior Tree is only ticked again, when it is not currently executing. Therefore, clauses 16 encode a tick, after the root node finished its execution

$$\begin{aligned} tick_0(X) &\leftarrow success_0(X) \\ tick_0(X) &\leftarrow running_0(X) \\ tick_0(X) &\leftarrow failure_0(X) \end{aligned} \tag{16}$$

4.9 Safety Property

In the previous sections, we presented how behaviour trees can be encoded using linear Constrained Horn Clauses. In this section, we present how to encode the safety properties of interest. Safety properties are equivalent to the reachability problem. Here, we show how additional clauses can be used to assert a condition over the Behavior Trees' variables. Given the safety condition $safe(X)$, clause 17 encodes whether the safety property holds at every tick of the root node. This encoding of invariants is not limited to the root node's *tick* predicate. Any introduced predicate can be used, depending on where in the Behavior Tree the property should hold.

$$\texttt{true} \leftarrow tick_0(X), \neg safe(X) \tag{17}$$

A possible safety property for the example behavior tree in Fig. 1 is that the robot is executing the action node *DoWork* if no collision or obstacle is detected.

If adding clause 17 leads to the SMT solver not finding a satisfying interpretation, the Behavior Tree fulfills the safety property since there exists no variable valuation which holds at the tick of the root node and is unsafe.

4.10 Environment

In many use cases, the system described by the Behavior Tree interacts with its environment. As in Fig. 1 the environment can be modeled as an subtree and is connected via a parallel node with the Behavior Tree of the system. To model the environment adequately, it is necessary to allow nondeterminism since some events only occur randomly or do not follow specific steps. In order to accommodate this, we added the possibility of assigning a random value to a variable.

$$p_2(X') \leftarrow id(X \backslash \{y\}), p_1(X) \tag{18}$$

Clause 18 illustrates the idea of modeling nondeterminism. Predicate p_2 is reachable from predicate p_1 where all variables except variable y retain their value. In the Horn Clause, y' is unconstrained and therefore can take any nondeterministic value.

5 Experiments

We implemented the linear encoding in our verification tool ArcadeBT which is a spin-off from ArcadePLC [19], a verification tool for Programmable Logic Controller programs. ArcadeBT is written in `C++` and uses the open source SMT solver Z3 [16] in version 4.8.15. In this version Z3 uses the Spacer algorithm [17] to solve Constrained Horn Clauses.

Since, to the best of our knowledge, no publicly available verification tool, which targets safety properties of Behavior Trees, exists, we compare our implementation against the general purpose verification framework SeaHorn [15]. It analyzes `C` programs by encoding the semantics in Constrained Horn Clauses which are then solved using Z3. In this section, we show the performance improvements, gained by exploiting the structure of Behavior Trees and the direct encoding of the semantics in Constrained Horn Clauses in contrast to transforming the Behavior Tree to a `C` program which is then analyzed by a general purpose verification framework which uses a similar encoding in Constrained Horn Clauses, and the same SMT solver Z3.

Our implementation currently does not contain a counterexample generator, as Z3 does not store the necessary information during the execution to reconstruct a counterexample. In the future, it should be possible to extract a counterexample from the derivation tree [18].

5.1 Benchmark

We used 39, from us created, different Behavior Trees for our benchmark. Each verification tasks consists of one or more safety properties. In these 39 tasks, there are 26 satisfiable tasks and 13 unsatisfiable tasks. The size of the verification tasks ranges from small (less than 5 nodes) to medium size Behavior Trees with 18 nodes.

Although our experiments with Behavior Trees containing parallel nodes show similar performance, we excluded them for reasons of fairness, since the simplified semantic could not be easily represented in `C` code which is the input for the SeaHorn framework. The generated `C` code does not use arrays, external header files, pointer arithmetic or dynamic memory allocation which would create are more challenging verification task.

All benchmarks were executed on a Linux 5.10 computer with 2 GHz, 16 GB memory and a timeout of 10 s. The implementation and the tasks can be found on GitHub[1].

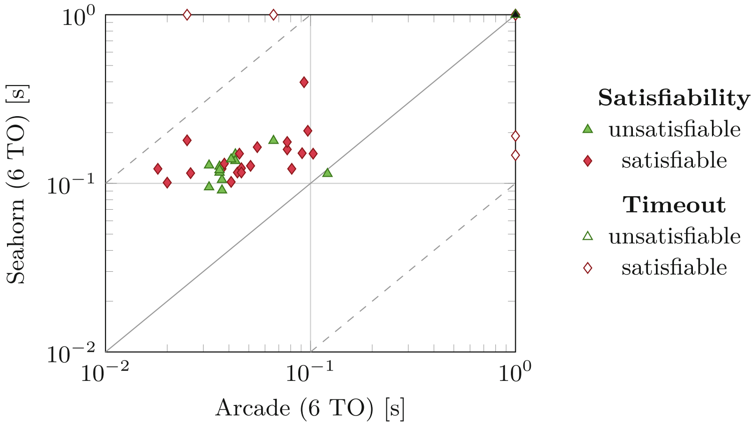

Fig. 2. Time spent by Arcade and Seahorn on verification of each task.

5.2 Discussion

Figure 2 illustrates the time needed for ArcadeBT and Seahorn to solve the 39 tasks. Each data point represents one of the verification tasks. The solid diagonal line splits the coordinate system into an area where ArcadeBT performs better (above the line) and where Seahorn performs better (below the line). Since both projects only share the SMT solver Z3 as a shared component, the time is measured for the complete execution of the respective verification tool.

In most cases ArcadeBT is 2 to 3 times faster than Seahorn and it does not matter whether the safety property is satisfiable or unsatisfiable. Both tools have 6 tasks where the time limit is reached and they do not return an answer. In four cases both tools cannot find an answer. The reason is likely a shortcoming in Z3 which in some cases has difficulties finding linear invariants for the predicates. The other two cases in which SeaHorn needs more than 10 s, are the only tasks which contains the modulo operator in at least one arithmetic expression. Since ArcadeBT can solve both tasks in reasonable time, it is very likely that this is due to a minor bug in SeaHorn. On the other hand, the two tasks where ArcadeBT needs more than 10 s, are not very different from other tasks which can be solved. A possible explanation is that these are also cases where Z3 has difficulties in finding linear invariants. Seahorn may be able to find the solution since it does not only use SMT solving, but also code optimization techniques and static analysis (e.g., value set analysis) which might simplify the Constrained Horn Clauses given to the SMT solver.

[1] https://github.com/embedded-software-laboratory/ArcadeBT.

6 Conclusion and Outlook

Behavior Trees are models which can visualize complex systems clearly. We showed that the not explicitly visible control flow can lead to overlooked bugs and that a verification approach based on linear Constrained Horn Clauses is able to find them. We also showed that an encoding utilizing the Behavior Tree structure is also faster than a general purpose verification framework.

Properties, which need a more precise modelling of concurrency than our atomic approach, are not supported yet. Also, multiple occurrences of the same action node leads to a redundant modeling since for every node new predicates are introduced. An extension of our presented encoding to handle interleaving semantics and model action nodes in a compositional way, leading to less redundancy, is left for future work.

Acknowledgements. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

1. Biggar, O., Zamani, M.: A framework for formal verification of behavior trees with linear temporal logic. IEEE Robot. Autom. Lett. **5**(2), 2341–2348 (2020). https://doi.org/10.1109/LRA.2020.2970634
2. Colvin, R., Hayes, I.: A semantics for behavior trees using CSP with specification commands. Sci. Comput. Program. **76**, 891–914 (2011). https://doi.org/10.1016/j.scico.2010.11.007
3. Klöckner, A.: Interfacing behavior trees with the world using description logic. In: AIAA Guidance, Navigation, and Control Conference (2013). https://doi.org/10.2514/6.2013-4636
4. Colledanchise, M., Murray, R.M., Ögren, P.: Synthesis of correct-by-construction behavior trees. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 6039–6046 (2017). https://doi.org/10.1109/IROS.2017.8206502
5. Klöckner, A.: Behavior Trees for UAV Mission Management (2013)
6. Ogren, P.: Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees (2012). https://doi.org/10.2514/6.2012-4458
7. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_31
8. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
9. Komuravelli, A., Bjorner, N., Gurfinkel, A., Mcmillan, K.: Compositional verification of procedural programs using horn clauses over integers and arrays, pp. 89–96 (2015). https://doi.org/10.1109/FMCAD.2015.7542257
10. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. arXiv abs/1709.00084 (2017)

11. Colledanchise, M., Parasuraman, R., Ogren, P.: Learning of behavior trees for autonomous agents. IEEE Trans. Comput. Intell. AI Games **11**, 183–189 (2018). https://doi.org/10.1109/TG.2018.2816806
12. Coronado, E., Mastrogiovanni, F., Venture, G.: Development of Intelligent Behaviors for Social Robots via User-Friendly and Modular Programming Tools, pp. 62–68 (2018). https://doi.org/10.1109/ARSO.2018.8625839
13. Colledanchise, M., Natale, L.: Improving the Parallel Execution of Behavior Trees, pp. 7103–7110 (2018). https://doi.org/10.1109/IROS.2018.8593504
14. Isla, D.: Handling complexity in the halo 2 AI. In: Game Developers Conference (2005)
15. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV (2014)
18. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reason. **5**, 363–397 (1989)
19. Biallas, S., Frey, G., Kowalewski, S., Schlich, B., Soliman, D.: Formale Verifikation von Sicherheits-Funktionsbausteinen der PLCopen auf Modell- und Code-Ebene. Tagungsband Entwicklung und Betrieb komplexer Automatisierungssysteme. EKA (2010)
20. Bohlender, D., Kowalewski, S.: Compositional verification of PLC software using horn clauses and mode abstraction. IFAC-PapersOnLine **51**, 428–433 (2018)
21. Colledanchise, M., Cicala, G., Domenichelli, D.E., Natale, L., Tacchella, A.: Formalizing the execution context of behavior trees for runtime verification of deliberative policies. In: IROS (2021)