Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics: Games Engineering

Testability of Probabilistic State Diagrams

Maximilian Vollath

Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics: Games Engineering

Testability of Probabilistic State Diagrams

Testbarkeit probabilistischer Zustandsdiagramme

Author:	Maximilian Vollath
Supervisor:	Univ.-Prof. Gudrun Klinker, Ph.D.
Advisor:	Daniel Dyrda, M.Sc.
Submission Date:	November 15th, 2020

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

November 15th, 2020 Maximilian Vollath

Acknowledgments

I owe a huge debt of gratitude to my parents because I would have not been able to finish this thesis in time without them.
Thank you, Dad, for helping me with grammar and everything LaTeX-related.
Thank you, Mom, for providing a steady supply of brain food and letting me neglect my household duties for a few weeks.
Also, thank you, Nina, for putting up with my stress, listening to me and encouraging me.
I'm also grateful that my dogs Jimmy and Bimmy didn't bark at the postman every morning in the days leading up to the deadline. Knowing them, however, they won't read this.
And to my advisor: Thank you, Daniel, for always being available for questions, even outside of your working hours.

Abstract

State diagrams are simple diagrams that are used to define the structure and behavior of discrete reactive systems in a visual and intuitive manner. In computer science, the most commonly used variants are statecharts, an extension of state diagrams using concurrent subsystems to model parallel execution.

This thesis elaborates the use of the probabilistic extension of statecharts, the P-statecharts, to support modeling of randomness in the system and in the system environment. The focus is on the development environment, able to edit, execute, simulate and especially test P-statecharts.

To facilitate productive development and use of P-statecharts, an extension using pseudo-nodes is presented. For this extension, the fundamentals of execution and simulation of P-statecharts are laid out in detail, including granular single step execution and analysis using Monte-Carlo simulations.

A development environment for P-statecharts has been developed which is also described in this thesis. It includes an editor for P-statecharts and two ways to simulate them: either step by step or in the form of Monte Carlo simulations.

In conclusion, P-statecharts become a very powerful modeling tool when supported by an appropriate development environment including visual editor, debugger and simulation.

Keywords State Diagrams, UML-statecharts, probabilistic, P-statecharts, pseudo-nodes, debugging, Monte Carlo method

Contents

1 Introduction

Statecharts State diagrams are simple diagrams that are used to define the structure and behavior of discrete reactive systems in a visual and intuitive manner. In computer science, the most commonly used variants are statecharts, an extension of state diagrams invented by David Harel. Statecharts introduce logic expressions and orthogonality[1] among other useful features [Har86]. They have become part of the Unified Modeling Language (UML) and are a broadly accepted tool for specifying software. Because they are highly expressive, analyzing their behavior is a complex and interesting task.

Probabilistic Statecharts Classical state diagrams and statecharts are not probabilistic[2] themselves even though their environment may be. That brings us to the difference between system randomness and environmental randomness, following the definition in [JHK02]. The former emerges if the system itself behaves stochastically, whereas the latter arises if its environment is of a stochastic nature.

Naturally, probabilistic extensions to introduce system randomness to statecharts have been proposed [JHK02, VCAA05, JHK03]. In addition to the ability to specify probabilistic behavior, these are useful for testing and quality-of-service (QoS) purposes in general. To facilitate analysis, environments are often modeled as being part of the system which effectively gets rid of the environmental randomness in favor of more system randomness. Very complex environments are also often considered to be random instead. This helps to abstract the elaborate processes that lead to the different outcomes [JHK03]. I use the term probabilistic statechart (or P-statechart for short, as in [JHK02]) to refer to a statechart with system randomness. However, this thesis will also cover issues on random environments which can be applied to non-probabilistic state diagrams.

Visual Scripting The visual and intuitive nature of statecharts - probabilistic or not - leads to the question of how they could be used as a visual scripting language. This would resemble how visual scripting is already employed, for example in the development of video games or functionally safe code. High-level behavior is usually specified using visual scripting whereas the underlying framework is implemented via text-based code. To create an interface between these languages, some of the framework's functions and events must be exposed to the visual scripting system. In the case of statecharts, this interface consists of events that are sent back and forth.

[1]In this thesis, the terms *parallel*, *orthogonal* and *concurrent* are used interchangeably.
[2]Furthermore, *probabilistic* and *stochastic* are used as synonyms.

Verification and Test In software development, code can have a variety of errors, causing system behavior to deviate from its specification. Model checkers to verify the algorithmic correctness of statecharts exist [KNP02, ZL10] but certain errors such as unreachable code can be found more efficiently using simpler methods. Similar to text development environments, a UI would ascertain the syntactical correctness of the diagram. It cannot, however, guarantee that they are semantically consistent.

For software engineers, the usability of a tool greatly depends on the verifiability and testability of the products built with it. This thesis explores methods to test and especially debug P-statecharts with the help of a prototype development environment that was created for this purpose.

Organization of the thesis

After a related work section, I provide the mathematical definition of the P-statecharts dialect that this thesis is based on. Afterwards, various use cases for these P-statecharts are presented. A general section on debugging and testing is followed by a specific exploration of the options of testing P-statecharts. Finally, the prototype software is presented in detail.

2 Related Work

Model Checkers for Related Problems The verifiability of statecharts is not a brand-new concept. While the model checkers that related papers employ for that task weren't specifically designed for statecharts, they do operate on systems that statecharts can be transformed into.

PAT [SL08] is a model checker designed for UML diagrams. PAT supports the use of various models, the most relevant of which is CSP#, an extension of communicating sequential processes (CSP), a formal language for specifying the interactions between concurrent subsystems. PAT offers many relevant features such as deadlock checking, reachability checking and the evaluation of assertions in linear temporal logic (LTL).

PRISM [KNP02] is also a model checker. It is used for the analysis of both discrete- and continuous-time Markov chains (DTMC and CTMC, respectively) and Markov decision processes (MDP). The languages probabilistic computation tree logic (PCTL) and continuous stochastic logic (CSL) are used to specify queries on the models.

Related Work on Statecharts A seminal and comprehensive probabilistic extension of statecharts is presented in [JHK02]. Many later works including this thesis build upon the specification of P-statecharts given in that paper. P-statecharts are mapped to strictly alternating probabilistic transition systems which are a subset of MDPs. To verify properties of P-statecharts, the properties are described using PCTL and the model checker PRISM is employed [KNP02].

In [VCAA05] statecharts are also extended with probabilities for the purpose of performance analysis of stochastic systems, especially in the long run. The probabilistic statecharts are combined with a stochastic environment that randomly fires events. This transforms the whole problem into a CTMC that can be solved with linear algebra.

The P-statechart dialect specified in [JHK03] is called StoCharts. Probabilistic delays between transitions are introduced for Quality of Service (QoS) evaluation. StoChart models are transformed into stochastic I/O-automatons, and the model checker ProVer is employed to verify characteristics specified with CSL.

[CFN10] explores modeling and simulation using statechart-based actors. Although the statecharts themselves are non-probabilistic, the environment may fire events with a random occurrence. These events do not necessarily follow an exponential distribution, and thus the resulting system cannot simply be transformed into a CTMC.

In [ZL10] the model checker PAT [SL08] is put into the context of statecharts in order to find modeling errors. Statechart models are translated into CSP#. While the statecharts used there are not probabilistic, CSP#, in principle, does support probabilistic behavior.

Although statecharts can be transformed into Petri nets [ABC14], this will not be covered in this thesis. Related work with petri nets does not consider any stochastic behavior of statecharts aside from very simple stochastic environments.

2.1 My Contributions

Since I consider P-statecharts as a visual scripting language in this thesis, I highlight new, relevant issues when they are placed in this context. I explore the testability of P-statecharts without fully focusing on one use case. Instead, I focus on broader issues and highlight the advantages and difficulties of different specifications. I introduce a definition for pseudo-nodes and a precise algorithm to work with my new concept of sub-locations. I also present a prototype development environment for P-statecharts. It has an integrated editor and it can be used to simulate the modeled system step by step while allowing the user to fully manipulate its internal state. Simulations can also be carried out in bulk, applying the Monte Carlo method to approximate probability distributions of the system's components.

3 P-Statecharts

Statecharts are a modeling language for specifying reactive systems. Nodes define the internal states of a system and edges or transitions specify the dynamic behavior. A statechart reacts to internal or external events that trigger edges which can activate or deactivate nodes and execute various actions. P-statecharts are an extension and replace edges with P-edges that exhibit stochastic behaviors. Probabilities are used to either resolve conflicts between multiple possible P-edges or to cause otherwise enabled P-edges to not be traversed in the first place. Because this makes P-edges more complex, their visual representation is facilitated by pseudo-nodes.

UML doesn't have a precise definition for even non-probabilistic statecharts. As such, different 'dialects' exist, and it is important to specify the one used in this paper. My semantics are very similar to the ones defined in [JHK02, VCAA05, JHK03] with the biggest differences being the way different P-edges are prioritized. I also specifically exclude non-determinism.

Furthermore, I introduce a variation that explicitly considers pseudo-nodes to be part of the specification. This improves the testability of statecharts for two reasons. First, the model specified by the developer is unified with the internal model used by the computer. Second, otherwise atomic steps are split up.

The only feature of statecharts that this thesis does not consider is the use of history nodes. These are used to enter a subsystem in the exact configuration it was in when it was last exited. History nodes and other temporal logic would go beyond of the scope of this thesis. They are not considered in related literature regarding probabilistic state diagrams but they could be subject of future work.

3.1 Syntax

The way this syntax is given is modeled closely after [JHK02].

Probability Space A probability space is a triple (Ω, F, P). Ω is the sample space, the set of all possible outcomes. The σ-algebra F is the event space where each event is a subset of Ω. $P : F \to [0, 1]$ is a probability measure that assigns probabilities between 0 and 1 to each event. If F is the power set of Ω, we can simply write (Ω, P). The elements of F are hereafter referred to as stochastic events in order to avoid confusion with statechart events.

3.1.1 Variant 1 - without Pseudo-Nodes

This section defines the syntax of what I call variant 1.

A single P-statechart PSC is a tuple $(N, E, \mathit{Vars}, G, A, \mathit{PE}, \mathit{PS})$ consisting of a finite number of nodes, events, variables, guards, actions, P-edges, and their priority scheme. If statechart collections (see below) are considered, the elements of statechart PCS_i are denoted with subscript i. As that isn't the case here, the subscripts will be omitted.

- Nodes N: Nodes are organized as a tree. Each statechart has exactly one root node $root$, and nodes can have any number of children.

  The functions $\mathit{parent} : N \setminus \{root\} \to N$ and $\mathit{children} : N \to \mathbb{P}(N)$ define the structure of the node hierarchy and must adhere to tree constraints.

  The function $\mathit{type} : N \to \{basic, and, or\}$ (in some literature referred to as $simple$, $composite$ and $sub - machine$ respectively [ZL10]) assigns a type to each node. A node is of type $basic$ iff it has no children, i.e. it is a leaf in the node hierarchy.

  The node types specify which nodes may be active at once. the root node is always active. If a node is active, so is its parent. If an or-node is active, exactly one of its children is. and-nodes specify orthogonal components and if an and-node is active, all of its children are.

  To simplify the specification of P-edges, or-nodes each have one default child node. They are specified by the partial function $\mathit{default} : \{n \in N : \mathit{type}(n) = or\} \to N$.

  Other literature commonly includes the restrictions that the root node must be of type or and that the direct children of and-nodes must not be and-nodes themselves. These make drawing a statechart easier, but as they do not change the semantics, they are not considered in this thesis.

- Events E: A set of events. E is identical for all statecharts within a collection.

  Internal events are events that are only used by the statechart itself. The environment does not interact with them in any way. In contrast, *external* events are only fired by the environment but the statechart may react to them.

- Variables Vars: A set of typed variables and a function $V_0 : \mathit{Vars} \to D$ to assign their initial values where D is the united domain of all variables. Typically, their domains are restricted to bounded integers.

- Guards G: Guards are boolean combinations of atomic clauses. Atoms are $\mathit{active}(x)$ for $x \in N$ (meaning the node x is currently active) and comparisons of arithmetic expressions containing the variables.

- Actions A: There are two kinds of actions: to assign an arithmetic expression (that may also contain variables) to a variable and to fire an event.

- P-edges PE: A member of PE is a P-edge, i.e. a tuple (X, e, g, p, P):

- $X \in \mathbb{P}_{\geq 1}(N)$ is a non-empty set of source nodes.
- $e \in E \ \cup \{\perp\}$ is the event that triggers the P-edge. $\perp$ denotes that no event is required.
- $g \in G$ is a guard.
- $p \in [0, 1]$ is the probability that the P-edge is actually traversed when it is triggered, control reaches it, and g evaluates to true.
- The function $P : A^* \times \mathbb{P}_{\geq 1}(N) \to [0, 1]$ is the probability measure of the probability space $(A^* \times \mathbb{P}_{\geq 1}(N), P)$, defining the probabilities of all possible *targets*.

  A target $(C, Y) \in A^* \times \mathbb{P}_{\geq 1}(N)$ consists of a (possibly empty) list C of actions to be executed in the given order and the non-empty set Y of destination nodes. Y must not contain any two nodes that are descendants (i.e. the transitive children) from the same or-node as they could not be active at the same time.

A P-edge is *simple* if P has exactly one target (C, Y) with probability 1, and will be denoted as (X, e, g, p, C, Y).

- Priority scheme PS: Multiple P-edges can be triggered by the same event at once, either because their source nodes overlap or because their respective source nodes are orthogonal and may be active at the same time. To ensure deterministic behavior of statecharts, a priority measure on P-edges is needed. PS is a partial order on PE where all P-edges that may come into conflict are comparable. To express that the P-edge a has a higher priority than b, one can write $a \leq b$. The simplest approach is to give all P-edges a numerical priority value. The lower this value, the higher the priority.

  A collection of statecharts also has a priority scheme that assigns priorities to its elements. These priorities define in which order the statecharts react to an event.

3.1.2 Variant 2 - with Pseudo-Nodes

Pseudo-nodes are widely used to aid in the visual specification of statecharts, either by breaking up complex P-edges into smaller, more manageable parts or by grouping multiple edges or P-edges together. However, a statechart's execution cannot halt on a pseudo-node. Other publications thus generally do not consider them to be part of the underlying data structure. Since the ability to temporarily halt on a pseudo-node is very useful for a debugger, they are explicitly considered to be part of the syntax in this variant.

P-Edges and PE are defined in a slightly simpler way and a P-statechart has the additional element PN, its set of pseudo-nodes. The other components of the statechart are defined identically to section 3.1.1. To avoid confusion, nodes that aren't pseudo-nodes are referred to as 'real nodes' in some sections.

P-Edges P-Edges are simple edges, but can point to pseudo-nodes. A P-Edge is a tuple (X, e, g, p, C, Y), $X \in \mathbb{P}_{\geq 1}(N) \cup PN$ is now either a non-empty set of nodes OR a single pseudo-node. The next three elements remain the same. However, instead of P, there is only one target whose components C and Y are directly part of the P-edge. $C \in A^*$ is still the list of actions. The destination $Y \in \mathbb{P}_{\geq 1}(N) \cup PN$ is now either a non-empty set of nodes OR a single pseudo-node.

A statechart's set PE now only contains the outgoing P-edges of real nodes. The P-edges of a pseudo-node a are considered to be part of the pseudo-node and are contained in the set $a.PE$.

Pseudo-Nodes In contrast to real nodes, pseudo-nodes are not arranged as a tree. A pseudo-node's outgoing P-edges do not require an event to be triggered, so $e = \perp$ for all of them. The P-edge with the lowest priority is a sort of 'default' edge for which $p = 1$ and $g = true$. This is to ensure that the pseudo-node can always be exited.

The most commonly used types of pseudo-nodes have one purpose each. They can be expressed with this syntax.

- Conditional pseudo-nodes (or *cond-nodes*) have multiple outgoing P-edges that are non-probabilistic ($p = 1$) but have different guards. They only have a singular destination node or pseudo-node. Their outcome is non-probabilistic and only depends on the statechart's internal state at the moment they are traversed.

- Purely probabilistic pseudo-nodes (or *prob-nodes*) are used to model random outcomes where the probabilities do not depend on the statechart's location. Their edges can have probabilities but no guards ($g = true$). They also only have one destination node or pseudo-node.

  The exact meaning of the edge's probabilities p is a bit unintuitive because they are cascading probabilities. Consider a prob-node with three outgoing edges a, b and c. Their respective probabilities are 0.2, 0.5 and 1, and $a \leq b \leq c$. As explained in section 3.3.1, during execution, these edges are 'tried' in order of their priorities. First, a is 'tried' and traversed with probability 0.2. If it isn't traversed, which occurs at a probability of 0.8, b is tried and traversed with probability 0.5. The *absolute* probability that b is traversed is thus $0.8 \times 0.5 = 0.4$. Following this, the absolute probability of c is $0.8 \times 0.5 \times 1 = 0.4$.

  To simplify the concept of these probabilities, an editor could allow the user to specify the absolute probabilities directly. This makes priorities unnecessary. However, it must be ensured that the probabilities add up to 1. Alternatively, the they can be specified as relative weights. I then call the pseudo-node a *weighted-node*. This is the way they are implemented in my prototype.

- Fork-nodes have exactly one P-edge and thus, $p = 1$ and $g = true$. Y consists of multiple 'real' nodes. Fork nodes model the simultaneous activation of multiple orthogonal nodes.

Equivalence Although variant 2 can be used to model everything that variant 1 can, this is not true the other way around. In variant 2, it is possible to create loops of pseudo-nodes. Such a loop could be constructed in a way that an event is fired a number of times according to a variable's value. Testing and verifying such loops is an entirely new problem. It makes sense to explicitly prohibit the construction of loops in an editor. My prototype, however, allows them.

3.1.3 Transformations Between the Variants

Transforming variant 1 to variant 2 is simple. For every P-edge $pe \in PE$

$$pe = (X, e, g, p, P)$$

a prob-node pn is created. The original P-edge pe is replaced with a simple edge pe' that points directly to a new prob-node pn:

$$pe' = (X, e, g, p, \emptyset, \{pn\})$$

Now, for each target (C_k, Y_k) in the original measure P, a P-edge pe_k containing the corresponding probability and actions is created.

$$pe_k = (\{pn\}, \bot, \mathit{true}, P(C_k, Y_k), C_k, Y_k')$$

If $|Y_k| = 1$, this edge points to the only node in Y_k directly:

$$Y_k' = Y_k$$

If $|Y_k| > 1$, the edge points to a new fork-node fn_k first, from which a further edge pef_k points to all nodes in Y_k.

$$Y_k' = \{fn_k\}$$

$$pef_k = (\{fn_k\}, \bot, \mathit{true}, 1, \emptyset, Y_k)$$

The other direction is trivial if P-edges only point to specific types of pseudo-nodes according to the result of the above transformation. Edges that come straight from real nodes can only point to prob-nodes, fork-nodes or real nodes. Edges from prob-nodes to fork-nodes or real nodes.

If these restrictions aren't satisfied or there are even loops of pseudo-nodes, the transformation is difficult or even undecidable.

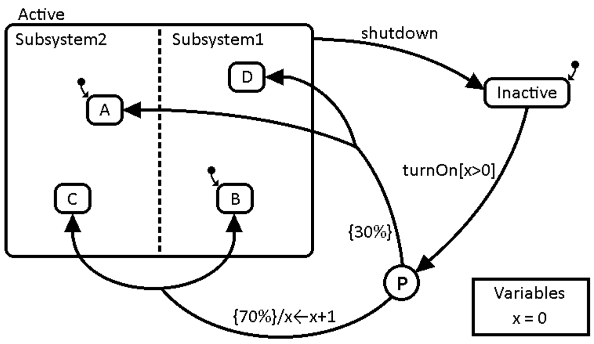
3.1.4 Collection of Statecharts

A collection of statecharts is a finite set $\{PCS_1, PSC_2, \ldots, PSC_n\}$ of communicating statecharts. This is a modular way to specify a system. A collection of statecharts can be transformed into a single statechart and will thus, besides said transformation, not be covered in the rest of this thesis.

This transformation is done by constructing a new node of type and that serves as the new $root$. Its children are the roots of the statecharts in the collection. The new priority scheme is constructed by combining the priorities of the statecharts and the priorities of their edges.

3.2 Visual Representation of a P-Statechart

As P-statecharts are an extension of statecharts, they are drawn in similar ways. I adopt the method used in [Har86] and summarize it. I extend this method to model probabilistic events with pseudo-nodes in a similar fashion to [JHK02, VCAA05, JHK03]. This means that a statechart in the form of variant 1 needs to first be transformed into variant 2, as described in section 3.1.3. I also make modifications that facilitate working on them in an editor.



An example P-statechart.

Nodes Nodes are drawn as squares with rounded corners. Children of or-nodes are depicted inside of their parents. The children of and-nodes are instead drawn as different parts of the parent that are separated by dotted lines.

Inside of every or-node is small arrow that points to the default child.

Pseudo-Nodes Pseudo-nodes are round, and their location is up to the statechart's creator. Pseudo-nodes are usually unnamed and display a single letter in their middle. This letter signifies their type: P stands for probabilistic and C for conditional. Fork-nodes usually display a small forking arrow or aren't drawn at all. Pseudo-nodes are typically unnamed, but if they are named, their behavior is apparent from the outgoing P-edges.

P-Edges A P-edge is drawn as an arrow that points from its source nodes to its destination node(s). The segments from the source nodes join a single segment with a *label* displayed above. This segment then points to the destination node or splits up to point to all destination nodes. Note that some of the destination nodes can be omitted. For example, a P-edge that points to a certain node n does not have to explicitly point to its parent, too. When n is activated, its parent automatically will be, too. A P-Edge's label is a string containing its triggering event e, its guard g, its transition probability p and its actions C. It is written in the format $e[g]p/C$.[1] C is written as a sequence of actions separated by commas.

A P-edge's priority is displayed as a superscript to the right of its label. If a P-edge cannot come into conflict with any others, its priority is omitted.

Variables are displayed in a list at the edge of the diagram, alongside their initial values.

3.3 Semantics

This section defines the semantics of P-statecharts, i.e. the behavior of the systems that they model.

Priorities The main distinction between different statechart semantics are the order in which events are reacted to, whether only one or all subsystems may react to the same event and how multiple triggered P-edges are prioritized. Prioritization is important even for multiple orthogonal systems because atomic steps that operate on the same objects cannot occur at the same time. In some cases, the order in which P-edges are traversed leads to different outcomes. Therefore, to ensure determinism, priorities are used to synchronize the subsystems. Prioritization was already addressed in the syntax.

I introduce another distinction: the *granularity* of steps, or what exactly constitutes an atomic time step. This is the key semantic difference between the two P-statechart variants. In principle, more granular steps lead to higher flexibility during debugging.

Locations At runtime, a statechart's current internal state is called a *location*. Locations carry information about which nodes are active, the valuations of the variables, and which events are yet to be processed. The transition from one location to the next is the process of completely reacting to a single event. To divide this process into smaller, analyzable steps, I use more precise sub-locations that describe the state of the system at certain points while it is processing an event. They carry all the information of the current location in addition to information required to resume and complete the transition to the next location.

A location, also referred to as a state in some literature like [JHK02], is a tuple $(Conf, Q, V)$ consisting of the following elements:

- $Conf$: The configuration $Conf \subseteq N$ is a set of currently active nodes with the following invariants:
  - $root \in Conf$

[1]Writing p in curly brackets is adopted from [JHK03].

  - $\forall n \in Conf : type(n) = or \Rightarrow |children(n) \cap Conf| = 1$

    (If an or-node is active, exactly one of its children is.)
  - $\forall n \in Conf : type(n) = and \Rightarrow children(n) \subseteq Conf$

    (If an and-node is active, all of its children are.)
  - $\forall n \in N : n \notin Conf \Leftrightarrow children(n) \cap Conf = \emptyset$

    (Iff a node is inactive, all its children are inactive. If a node is active, its parent also is.)

- Event queue Q: A list of events that are yet to be reacted to. This queue is identical for every statechart in a collection. For many statecharts, this queue will contain at most one event.
- Valuation of variables V: The function $V : Vars \to D$ assigns the current value to each variable.

The validity of a guard g depends only on the statechart's location L. The term 'g evaluates to $true$' is written as $L \models g$.

The structure of a sub-location will be explained after the step construction.

Initial Location A P-statechart's initial location includes the initial valuations of its variables and an empty event queue. The initial configuration $Conf_0$ is constructed by adding the following rule to the above mentioned invariants:
$\forall n \in Conf_0 : type(n) = or \Rightarrow default(n) \in Conf_0$.

3.3.1 Step Construction

When a statechart SC in a location L processes a new event e, at first, the queue $T \subseteq PE$ of enabled P-edges is computed. A P-edge t is enabled (by e) if $t.X \subseteq L.Conf$, $t.e = e$ and $L \models t.g$. The P-edges in T are then executed in the order according to their priorities.

P-Edges To execute a single P-edge t, with a probability of $1 - p$, no action is performed at all. Otherwise, it is traversed, the exact manner of which depends on the P-statechart variant.

For variant 1, a target is chosen at random from P. This target consists of a list of actions that are executed in the given order and of the destination nodes that are activated.

For variant 2, the actions in C are executed in the given order, too. Then, if the destination Y is a set of 'real' nodes, they are activated. If the destination is a pseudo-node, however, it is traversed before any other P-edges from T are executed.

Pseudo-Nodes If the target's destination is a pseudo-node, it is immediately traversed. From its outgoing P-edges, the one with the highest-priority so that $L \models g$ is selected. It is traversed with the probability p. Otherwise, this is repeated for the remaining edge with the highest priority. Since each pseudo-node has a 'default' P-edge, this process always leads to a P-edge being traversed.

If the next destination is another pseudo-node, this is repeated until the obtained destination is finally a set of real nodes. They are then activated.

Activating a Set of Nodes The process $Activate\_Nodes$(configuration $Conf$, node set Y) activates the nodes in Y by adding them to the configuration $Conf$. To keep the configuration consistent with its invariants, additional nodes may be activated or deactivated. The new configuration retains all possible nodes from the original configuration.

For statechart variations where nodes can have sets of events that are executed upon being entered or exited, the nodes that were activated and the ones that were deactivated during have to be kept track of.

Iterations At this point, any other P-edges that were disabled by this process are removed from T. Then, the next P-edge in T is traversed. When T is empty, this entire process is repeated for all P-edges that require no event, i.e. where $t.e = \perp$. It should be noted that it can also make sense to instead react to event-less P-edges first. It is only when there are no more enabled P-edges left that the next event is reacted to. A statechart with an empty event queue cannot execute any more steps and is considered to be dormant. It wakes up and resumes processing once an event is fired by an external source.

If sub-locations are not used, the following algorithm describes how a step is constructed. Here, a single step encompasses the statechart's entire reaction to the next event in the queue. In the following pseudo code functions, green colored lines are unique to variant 1 and purple colored lines are unique to variant 2.

```
React_to_Event(P-statechart PSC, location L)
1.    if (|L.Q| = 0)
2.        end //There is nothing to process.
3.    e $\leftarrow$ pop event from L.Q
4.    T $\leftarrow$ queue of all P-edges enabled by e, ordered by their priorities
5.    Execute_Edge_Queue(PSC, L, T)
6.    T $\leftarrow$ queue of all P-edges enabled by $\perp$, ordered by their priorities
7.    Execute_Edge_Queue(PSC, L, T)
8.    end

Execute_Edge_Queue(P-statechart PSC, location L, P-edge-queue T)
1.    while (T is not empty)
2.        t $\leftarrow$ pop edge from T
3.        do (randomly at probability t.p)
4.            tr $\leftarrow$ choose target randomly from t.P
5.            tr $\leftarrow$ (t.C, t.Y)
6.            foreach (action a in tr.C) in order of appearance in list
7.                execute a
8.            Y $\leftarrow$ tr.Y
9.            while (Y is pseudonode)
10.               pn $\leftarrow$ Y
11.               foreach (P-edge x in pn.PE) in order of their priorities
12.                   if (L $\models$ x.g $\wedge$ randomly at x.p)
13.                       t $\leftarrow$ x
14.                       break
15.               foreach (action a in t.C) in order of appearance in list
16.                   execute a
17.               Y $\leftarrow$ t.Y
18.           Activate_Nodes(L.Conf, Y)
19.           remove all now disabled P-edges from T
20.   end
```

At the latest, the condition at line 12 is true when the pseudo-node's last P-edge is reached. This is because it is the 'default' edge and $p = 1$ and $g = true$. Pausing execution on a pseudo-node or before the next P-Edge in T is executed requires an instruction such as the yield return of C#. This would be inserted after the lines 17 and 19 respectively. The yield return instruction makes it possible to pause a method by turning it into an iterator. Alternatively, sub-locations can be used that are much easier to analyze and manipulate than a yielded function.

Sub-Location A sub-location (L, e, pn, T) contains the following elements:

- The location L
- The event $e \in E \cup \{\perp, null\}$ that is currently being processed. $e = \perp$ means event-less edges are currently being processed. $e = null$ means that no event is currently being processed. In this case, the next step is to react to the next event in Q.
- The currently active pseudo-node pn, or $null$ if no pseudo-node is active. This entry does not exist for variant 1.
- The queue T of enabled P-Edges

Invariant: If no event is currently being processed, i.e. $e = null$, the queue T must be empty and pn must be $null$.

Since sub-locations also keep track of the current pseudo-node in variant 2, activating a destination is slightly modified. For variant 1, Activate_Destination and Activate_Nodes are identical.

```
Activate_Destination(sub-location SL, destination Y)
1.    if(Y is pseudo-node)
2.        SL.pn ← Y
3.    else
4.        SL.pn ← null
5.        Activate_Nodes(SL.Conf, Y)
```

The following algorithm carries out the step to the next sub-location. In one step, either a single P-edge is executed or the next event from the queue is popped and activated. Setting the active event to $\perp$ also counts as one step.

```
Step(P-statechart PSC, sub-location sL)
1.  if (sL.pn $\neq$ null)
2.      foreach (P-edge x in pn.PE) in order of their priorities
3.          if (L $\models$ x $\wedge$ randomly at x.p)
4.              t $\leftarrow$ x
5.              break
6.      foreach (action a in t.C) in order of appearance in list
7.          execute a
8.      Activate_Destination(SL, Y)
9.      if (Y is not pseudo-node)
10.         remove all now disabled P-edges from T
11. else if (|T| > 0)
12.     t $\leftarrow$ pop edge from sL.L.T
13.     do (randomly at probability t.p)
14.         tr $\leftarrow$ choose target randomly from t.P
15.         tr $\leftarrow$ (t.C, t.Y)
16.         foreach (action a in tr.A) in order of appearance in list
17.             execute a
18.         ActivateDestination(SL, tr.Y)
19.         if (tr.Y is not pseudo-node)
20.             remove all now disabled P-edges from T
21. else if (SL.e $\in$ {$\perp$, null})
22.     if (|SL.L.Q| > 0)
23.         SL.e $\leftarrow$ pop event from SL.L.Q
24.         SL.T $\leftarrow$ queue of all P-edges enabled by SL.e, ordered by their priorities
25.     else SL.e = null //The P-statechart goes dormant.
26. else
27.     SL.e $\leftarrow$ $\perp$
28.     SL.T $\leftarrow$ queue of all P-edges enabled by $\perp$, ordered by their priorities
```

As in React_to_Event, the condition at line 3 will always be true when the pseudo-node's last P-edge is reached. It is possible to further granulate this process, for example by considering a single executed action to be one step.

4 Applications of P-Statecharts

Statecharts are typically used to model the software of reactive systems. P-statecharts extend this with the ability to include system randomness.

Environmental Randomness Statecharts can also be used to model possibly non-probabilistic systems that are subject to environmental randomness. When analyzing software, we assess how the system under inspection acts in its environment. If said environment is modeled, the focus should thus be on the parts that interact with the system. Complex processes in the environment that lead to a certain outcome may be abstracted and simplified by modeling them in a probabilistic way, instead.

For instance, consider another system where a certain interaction with the environment may fail due to entirely external reasons. The specifics of how and when this happens don't matter for our system under inspection and it is enough to just model the probability at which the failure occurs.

Quality of Service and Performance An important application for these models is their analysis for the purposes of QoS testing and performance modeling. Examples include the model of production machines in [VCAA05] and the model of an ATM machine and its interactions with a bank and human clients in [JHK03]. Extensions to statecharts that include time, such as the delay action or the after operator, can be employed to analyze a system's time performance.

Visual Scripting The premise of this thesis is the use of P-statecharts directly as a visual scripting language. This way, statecharts don't have to be transformed first into code to execute them; they are the code.

Visual scripting is a powerful tool that is, for instance, increasingly used in video game development. The Unreal Engine [unr] uses a language called blueprints for game object behavior, shaders and animations. In Unity [uni19], there's the shader graph, the animator and experimental visual scripting for DOTS [ans17]. CryEngine [cry] also has a system called Schematyc.

Visual scripting languages typically model sequential behavior, just like conventional code. The notable exception are animators that make use of a state machine model where each state corresponds to an animation that is currently being played. Conditional edges then describe how and when these animations transition into each other.

Like existing visual scripting languages, P-statecharts can be a great tool to specify the behavior of certain components of a framework in an intuitive and designer-friendly way while still retaining the ability to interact with the rest of the system.

Gaming Examples This example refers to a computer controlled opponent in a fighting game. The enemy behaves randomly so that the player cannot fully predict its actions. The internal state might be specified by the orthogonal components FightingPlan and Aggravation, and the variable LastMove. FightingPlan specifies the enemy's immediate behavior and makes use of states such as Blocking, Walking Backwards, and Lockout (used for while mid-air or when stunned by the player). The enemy inputs their moves in the game via events such as LightPunch, HeavyPunch and Parry.

Aggravation influences the probabilities used inside FightingPlan. If in state Berserk the enemy will not block or dodge and will instead keep trying to hit the player with a combo.

Components P-statecharts can also be used to specify very small systems like loot tables or other in-game events like acquiring an achievement or failing a level. Minecraft [min], for instance, uses a quite sophisticated system to specify loot tables, and achievements (called 'advancements') and their rewards with JSONs. These can be modified to execute a wide array of commands while considering many in-game criteria [Wik20], e.g. whether or not the player is wearing a piece of equipment with a specific custom name. For games with complex reward systems, probabilistic statecharts can be a great way to specify them.

5 Testing and Debugging

Testing According to the Oxford dictionary, a test is a procedure intended to establish the quality, performance, or reliability of something, especially before it is taken into widespread use [OUP20b]. A central task in testing is validation, which can be defined as the activity of checking for deviations between the observed behavior of a system and its specification [Bru18]. The causes of incorrect behavior are found via the process of debugging, defined as the process of identifying and removing errors from computer hardware or software [OUP20a].

Active and Passive Testing We can further distinguish between active and passive testing: Passive testing is a software testing technique that observes the system without interaction. On the other hand, active testing involves interaction with the system [Dic20].

The simplest form of active testing is to just execute the program to see if it executes the way it should, either by inspecting its output or by looking under the hood, at its internal state. Being able to set breakpoints and pause the program's execution to thoroughly evaluate its internal state is indispensable.

Goals To evaluate system behavior, one must assess whether it meets defined goals. Objective goals are specific ways in which the system is to react when given a defined set of circumstances.

In a fighting game, for example, such a goal may be ″if the player presses the down key and is currently on the ground, the character starts crouching.″ In stochastic settings, probabilities are also the subject of goals, as in ″the character *Demon Guy*'s fireballs deal double damage 30% of the time.″ Subjective goals represent less tangible results. A designer may specify that ″the character *Lightning Guy* walks fast″ but the exact speed that 'feels' the most correct to the designer is found by testing at runtime.

Debugging and Development As stated in the definition, once the error was identified, the next step is to rectify it by modifying the software. Debugging and further development of the software go hand in hand, and so, ideally, the testing environment and the development environment are the same thing. In addition to finding errors, the resulting environment supports the developer in software's design to begin with.

The development environment is also used for most of the passive testing. Syntax checking is one of the most important features here. Less trivial features include the identification of patterns that are likely mistakes such as the specification of unreachable code, or unreachable nodes in this context.

5.1 Testing and Debugging Applied to Statecharts

This section discusses the options and the goals of testing and debugging of both P-statecharts and their non-probabilistic counterparts. Subsequently, section 5.2 goes in-depth on the testing of probabilistic systems.

The central concept to statechart-based systems is that they react to events. In order to fully analyze their behavior, assumptions about their environments have to be made. Standalone systems with purely internal events that do not require an environment to function can be analyzed more easily. Fortunately, there are many ways to combine a statechart with a model of its environment to create a new, closed system.

Closed Systems Standalone statechart systems, like other systems that do not require any external input, can simply be executed. Existing debuggers for text-based code offer a variety of features to facilitate the analysis of a program at runtime. Breakpoints can be set at certain lines so that when control reaches them, execution is paused. In this paused state, a developer can view and alter the current values of variables and go line-by-line to resume execution one step at a time. Very much in the same fashion, P-statecharts can be debugged. Breakpoints can be set at nodes, transitions, other specified conditions or at the very beginning of the execution. Then, the developer can execute one step at a time or skip forward until another specified condition is met.

The internal structure, defined by the location or sub-location, of P-statecharts is by no means complex because the only part with no fixed maximum size is the event queue. This allows for full manipulation at runtime.

Modifying which nodes are part of the current configuration is the first feature that comes to mind. Due to the simplicity of statecharts it is furthermore possible to manipulate their entire structure. New variables can be added, conditions changed and even the structure of the node hierarchy can be altered by deleting nodes, adding new ones, or changing a node's type.

Sub-locations introduce even more information that can be modified if an event is currently being processed, such as which transitions have yet to react to it. Changing which pseudo-node is active is also trivial.

Testing of Probabilistic Systems Section 5.2 goes in-depth on the methods mentioned in the following.

Debugging the probabilistic behavior of P-statecharts presents new challenges. The simplest approach is to apply the Monte Carlo method and execute the program repeatedly to create samples. These samples are then used to make inferences on the system, such as the probability distribution of nodes after a specified sequence of events has been reacted to.

P-statecharts can furthermore be transformed into matrices. Their behavior is then described by a Markov Chain to provide the probability distribution of the internal state at any point in time during the execution. This works best when there are few orthogonal components and variables, as otherwise the number of possible internal states blows up.

Environment Modeling Testing a statechart in an external environment presents many challenges, especially if the environment isn't accessible for an arbitrary number of test runs because of cost, time, or other resources required. An example of this would be if the system controls a firework show. If the way the environment interacts with the statechart's events is known to a sufficient degree, the environment can be simulated by replacing it or parts of it with mock-objects.

One simple way of doing this is for the tester to simulate the environment themselves by manually firing events. This is essentially equivalent to adding events to the event queue at runtime.

Oftentimes, the environment itself can be modeled as a P-statechart. Randomness may be used to abstract more complicated processes in addition to any stochastic behavior it may already exhibit. The simulated environment is then combined with the statechart(s) under inspection to create a new statechart that doesn't depend on any external environment anymore. Naturally, this simulated environment needs to be adequately tested as well.

This methodology can also be employed when testing a specific subsystem. Other parts of the system may be replaced by less complex mock-objects to facilitate the simulations and calculations. In principal, the replacement of any systems, whether they be part of the environment or not, sacrifices accuracy in favor of testing efficiency.

Human environments are tough to model and oftentimes, there is no way around testing the system with human testers.

In situations where the statechart does not influence the environment (or at least the part of it that fires events), it makes sense to just model the events fired by the environment. The assumption that events fire randomly according to an exponential distribution leads to a CTMC. With this, the probability distribution of a statechart's internal state at any specified point in time can be calculated with ease. In some circumstances, it makes sense to record sequences of events that the environment fires. These sequences can then be used in testing.

Specifying Input An important issue in testing is how exactly the developer specifies what is tested and how that is carried out. Obviously, the statechart itself is part of the input which is why it's beneficial for the debugger to be integrated into the editor. This way, the developer does not have to manually copy the entire statechart into the testing software. The statechart's transformation into whichever models are used by the software internally can occur automatically. These models can be entirely concealed from the developer unless an external tool like is to be used. In that case, there can be an option to export the statechart as the required input model.

Test statements are often assertions with various levels of complexity. Expressive languages like CSL are widely used for their specification. Example assertions include that a specified configuration is reachable from another and that the probability of a specified event is within a certain range. My prototype uses an ad hoc format where simple clauses are combined with temporal notation to form queries on probabilities.

Passive Testing The editor itself should offer functionality to passively test attributes of the system. Detecting syntax errors is relatively simple and absolutely necessary. A statechart with an incorrect syntax can either not be executed at all, or worse, lead to corrupt invariants and behave in an undefined way which makes testing unnecessarily complicated and confusing.

The typical goal of passive testing is the identification of patterns that likely indicate mistakes. A common example is unused or unreachable code. In the context of statecharts, the counterparts to this are isolated groups of nodes with no edges to connect them to the starting configuration. They can never be activated and their specification, if intentional, would be obsolete.

The possibility of the system to enter a deadlock [ZL10] also in all likeliness points to an error and the developer should be notified about it.

Absorbing states are states that, once entered, can never be exited again. They don't necessarily point to an error, but they are relatively cheap to detect and visualize.

Another useful tool is the detection of ambiguities. Two edges may come into conflict, meaning there are situations where both of them are enabled and it has to be decided which one is prioritized. These ambiguities are solved by the priority measure. Its specification may be easier for the developer if they are able to see which groups of edges could potentially come into conflict. This could be done liberally, as having to specify one priority too many is annoying but specifying one too few violates the syntax. It's possible that the editor cannot detect whether a set of edges may come into conflict, but the developer is sure that they may not. In that case, there could be an option to allow the developer to explicitly confirm that there is, in fact, no ambiguity. If developer was wrong, this should then lead to an exception at runtime.

5.2 Test Statement Checking on P-Statecharts

In this section, I apply two classes of methods for test statement checking to P-statecharts. These can be used to evaluate the probability of a single stochastic event or entire probability distributions.

5.2.1 Monte Carlo Simulation

The Monte Carlo method is a simple yet effective approach that is very valuable when dealing with large systems where exact inference is too expensive. The core principle of this method is to carry out a great number of simulations and then to observe the outcomes. To increase efficiency, the samples can be run simultaneously on multiple threads or machines. Their observations are then used to approximate certain probability distributions.

Depending on the context, approximations are enough. In video game development, for example, it is often not required to fully prove that the exact values of these probabilities are equal to the specified ones. Even if a small deviation can, in rare cases, force the restart of a sub-routine, the player likely won't notice [Dyr20]. Of course, the more samples are created, the more accurately they approximate the system's actual probability distributions.

It is important that a system can be run or simulated quickly and at low cost. If the costs to run a system in its environment are too high, the environment can be modeled and/or simplified to facilitate this method. An example of this would be if a P-statechart is used to program a machine at an assembly line.

The Monte Carlo method requires some error tolerance because there can be no guarantee that the observed probability distributions accurately reflect reality. The best way to improve its accuracy is to increase the sample size.

I explore two ways to apply the Monte Carlo method to statecharts. In method A, samples are created from the initial location to obtain the probability distributions of selected measures. In method B, the probability of a specified stochastic event is evaluated with the use of more targeted samples.

Parameters For both alternatives, an initial location or sub-location is specified. This is the simulations' starting point. Optionally, a sequence of external events can be given to model the environment. This is not required for statecharts with purely internal events as they can simply be executed. These events are not immediately added to the queue at once; one is added each time the statechart finishes processing and becomes dormant. The sample size n, i.e. the total number of simulations, is also specified.

In order make statements about the probability distribution at specific points in time, these *moments* need to be well-defined so their data can be matched across the different simulations.

Moments A *moment* is a point in time during execution when information about the system is recorded. For instance, if an external event sequence is specified, it is useful for take a snapshot of the system at every moment when an event from the sequence is fully processed. If the last moment is reached, simulation terminates. In the following, $T = 0$ denotes the starting point of the simulation and $T = t$ denotes the t^{th} moment.

Method A All n simulations are carried out and at each moment, the internal state and other data (such as how many times each event has been fired) is recorded. This information is then used to calculate a variety of statistics.

The probability distribution of a numerical variable's value can be summarized with key measures such its mean, range and measures for its dispersion. When more detailed information is requested, the entire distribution can be displayed as a histogram. The change of this distribution through time could be modeled graphically as well, either with boxplots for each moment or with stacked graphs displaying the development of different percentiles.

The probability distribution of the nodes, i.e. the probability that each given node is active at the considered moment, can simply be displayed below the node. In addition, the nodes can be colored in the statechart diagram to provide a clearer overview of the resulting heatmap. The frequency with which internal events are fired and P-edges are traversed could similarly be represented with a color scale.

Method B Samples can also be used to directly evaluate the probability of certain occurrences, such as a group of nodes being active at a specified moment.

At first, the exact statement that shall be evaluated is specified as a probabilistic expression. This can include conditional probabilities and combinations of clauses as long as the entire statement can be evaluated conclusively for each sample.

For instance, $\Pr((active(N) \text{ at } T = 5) \wedge (active(N)) \text{ at } T = 6)$ is a valid statement that describes the probability of node N being active at the moments 5 and 6.

It is worth pointing out that if a full location is inside the statement, it can instead be set as the starting location for the simulation. Ex: $\Pr(something\, at\, T = 30 \mid exact\, L\, at\, T = 15)$.

The samples are then generated by simulating the system n times. Every time a sample's simulation reaches a *moment*, the parts of the clause that refer to it can be evaluated. If at any point the entire statement can be evaluated, the simulation of the sample terminates. Consider the expression $\Pr(active(N)\ at\ T = 10 \vee active(M)\ at\ T = 15)$. If $T = 10$ is reached and N is active, the entire statement evaluates to true and the sample is finished. The same principle applies to conditional probabilities. If the clause is $\Pr(var1 = 0\ at\ T = 2 \mid var2 = 0\ at\ T = 1)$ and $var2 \neq 0$ at $T = 1$, the sample can be discarded because its outcome does not affect the result.

The resulting probability is then $\frac{t}{a}$. t is the number of samples for which the clause evaluates to true and a is the total number of accepted samples. a includes all samples except for the ones that are discarded when to the condition of a conditional probability evaluates to false.

Due to the nature of Monte Carlo simulations, this inference is an approximation of the real distribution. To make a statement on the accuracy of the observed probability, confidence intervals can be computed.

5.2.2 Statecharts as Markov Chains

This section describes how probabilistic state diagrams can be transformed into Markov Chains. However, the description is superficial and incomplete. It should serve as a general idea and does not go into detail on how this would be implemented.

Markov chains are stochastic models that describe how a discrete, finite, and probabilistic system changes through time. These are expressed with transition matrices where each entry corresponds to the transition probability from one state to another. Markov chains are memoryless. That means that these probabilities only depend on the system's current state, not its history.

In contrast to the Monte Carlo method, Markov chains can provide exact results rather than approximations.

Limitations The transformation to a Markov chain works best if there are few distinct internal states that a system can be in. This is not generally the case for statecharts because the use of orthogonal components can lead to the state-blowup phenomenon [Dru94]. Furthermore, the maximum length of the event queue and the domains of variables are generally not restricted in length. It may not be feasible or even possible to transform a given statechart into a Markov chain. Sub-locations are not considered here as they drastically increase the number of states.

If the event queue is restricted in length, a P-statechart can be transformed into a discrete-time Markov chain (dtMC).

If a P-statechart's entire reaction to external events is considered as a single step, it can be transformed into a Markov decision process (MDP) or a continuous-time Markov chain (ctMC).

Transformation into a Discrete-Time Markov Chain In the model of a dtMC, a P-statechart is expressed as a single square matrix of transition probabilities. I will only consider P-statecharts whose event queue is restricted to one element at most. This transformation is still possible if it's restricted to a higher number but the number of matrix dimensions drastically increases.

Matrix Dimension To construct a transition matrix, at first, the total number of distinct locations is computed. This will be the dimension of the transition matrix. This number is equal to:

$|Conf_{root}| \times \prod_{v \in Vars} dom(v) \times |E \cup \emptyset|$

$|Conf_{root}|$ is the total number of possible configurations. Not every subset of N is a legal configuration, and the number depends on the specific hierarchy that N is organized in. This number can be constructed recursively: For a node $n \in N$, let $|Conf_n|$ be the number of possible *sub-configurations* that begin with n. That is, it is the number of possible configurations of a statechart with n as its $root$ node. This number depends on its type and its descendants.

For a $basic$-node, $|Conf_n| = 1$.
For an or-node, $|Conf_n| = \sum_{c \in children(n)} |Conf_c|$.
For an and-node, $|Conf_n| = \prod_{c \in children(n)} |Conf_c|$.

The second factor is the total number of possible variable assignments. It is the product of the domains of each variable.

The third factor is the number of possible arrangements of the event queue. As I limit this queue to a maximum length of 1 here, it is simply the number of events plus 1.

Entries In this matrix, each row and each column corresponds to one location. For each location, all of the possible outcomes of the next step are computed alongside the probability of their occurrence. The longer the chains of pseudo-nodes are, the more complex this is and the more outcomes are possible. The corresponding row is then filled with the computed probabilities. Each probability is entered in the column of the corresponding outcome.

Transformation into a Markov Decision Process In the model of an MDP, the P-statechart is expressed as a set of transition matrix where each matrix corresponds to an external event. This is useful when analyzing a system that always completely reacts to an external event and goes dormant before the environment fires the next one. For this reason, only locations where the statechart is dormant are considered, i.e. locations with an empty event queue.

The dimensions of the matrices are then given by:
$|Conf_{root}| \times \prod_{v \in Vars} dom(v)$

Similar to above, the entries are constructed by simulating all possible steps that the statechart can take. This begins with the reaction to the event that the matrix corresponds to. Every possible outcome is simulated until a dormant state is reached. Processes that fire many internal events, especially those that can fire indefinitely many, may not be able to be solved at all. In that case, transforming the statechart into an MDP is not a viable option.

This set of matrices can used to simulate a long sequence of external events. Multiplying these matrices together results in a resulting matrix. Each row is then filled with the probability distribution of the P-statechart's location after reacting to the entire sequence, if it is initially at the row's corresponding location.

Transformation into a continuous-time Markov Chain In the model of a ctMC, the P-statechart is expressed as a matrix containing the rates of transition from each location to the next. A ctMC is useful when analyzing a system that, like above, fully reacts to external events before the next one is fired. In addition, this model assumes that the external events are each fired at random intervals according to an exponential distribution, as in

[VCAA05]. At any point in time, the time that passes until the event v is fired again is given with the probability density function $\lambda_v e^{-\lambda_v t}$.

The P-statechart is then modeled as a single matrix that describes how the location probability distribution changes through time. Unlike in a dtMC and an MDP, the matrix does not contain transition probabilities for a discrete time step. Instead, each entry is the expected rate of change from one location to another. This value corresponds to the parameter λ in the above equation.

To transform a P-statechart into a ctMC, it is transformed into an MDP. The resulting matrix is then the sum of each matrix of the MDP, multiplied by the rate λ_v of the corresponding event v.

The steady-state of a ctMC models the distribution of the P-statechart's locations in the long run [VCAA05]. This is useful for systems that are in continuous use, like a factory.

6 Prototype Editor and Debugger for P-Statecharts

A prototype was created with Unity [uni19][1] to explore the testability of P-statecharts. It encompasses an editor and supports various features for testing and debugging. Although the specification of an editor is not the primary topic of this thesis, the editor was created because debuggers and editors go hand in hand.

6.1 Overview

Using the editor, the user specifies the P-statechart visually, in accordance with variant 2 introduced in section 3.1.2. The main difference to the defined syntax is that the source node set X of a P-edge can only contain one single source node. Apart from requiring additional clauses in guards, this is semantically equivalent to the original P-edge.

The visual representation of the designed statechart follows the method of section 3.2. For technical reasons, the types of nodes are represented with colors instead of separating the children of and-nodes with dotted lines.

The internal model used is equivalent to the displayed graphic model and sub-locations are used at run-time. This has the benefit that everything is executed in the exact same way it is specified, including pseudo-nodes.

The user can simulate the P-statechart step by step with full control over its internal state. The Monte Carlo method is used to calculate the probability distributions of various properties and the probabilities of specified expressions. The former are displayed visually.

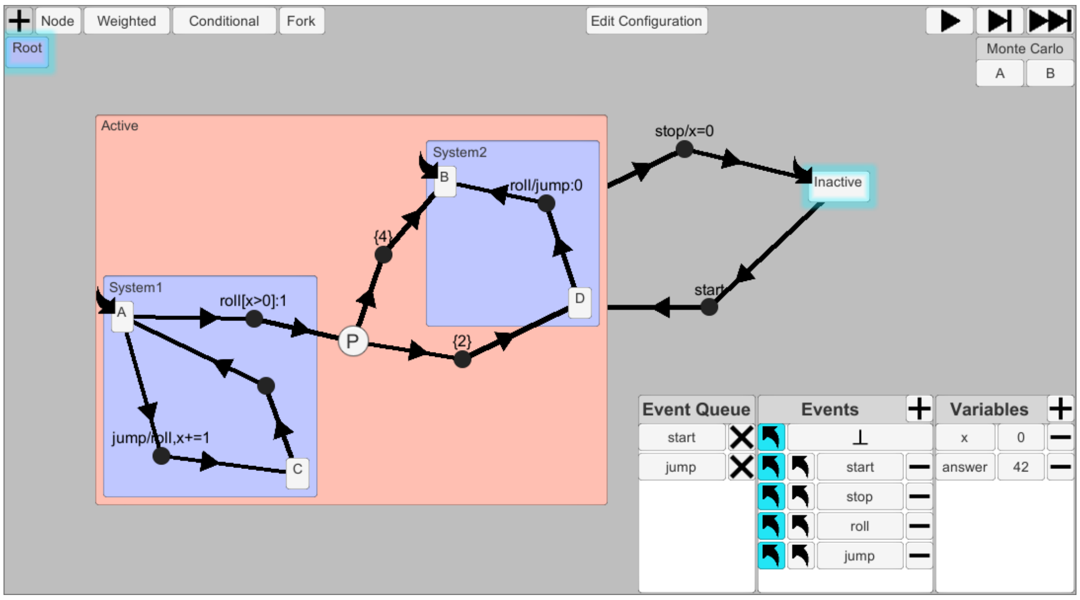
6.2 Features

6.2.1 Editor

This editor is the main feature that differentiates this program from model checkers. The user employs it to specify a P-statechart visually and they can work on it directly, without having to translate it into, or from, another model.

[1]Most of the graphic assets used are default assets from Unity.

All of the editor's features can also be used at runtime. Technically, the statechart is always at runtime because this software does not differentiate between the starting location and the current location.



Event Queue
start
jump

Events
⊥
start
stop
roll
jump

Variables	
x	0
answer	42

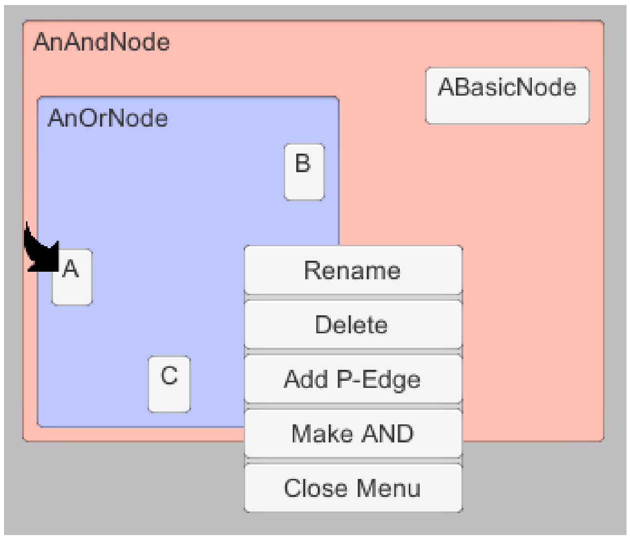
The editor's GUI.

6.2.2 Structure Manipulation

Nodes Nodes - the building blocks of statecharts - are typically the first elements that are specified. The *root* node always exists, and it is drawn in the top left corner. To reduce visual clutter, its children are not drawn inside of it. More nodes can be created by clicking on the button "Node" in the top left corner, then clicking on the parent node. Clicking in empty space sets the *root* node to be the parent. Nodes can be rearranged by dragging them. This also moves their children whose relative positions stay constant.

Right-clicking on a node opens a small context menu. It has buttons to rename and delete the node, and to swap the node's type between *and* and *or*. Nodes without children don't have this option as they are always *basic*-nodes. If its parent is an *or*-node, the node can be set as the default child. The menu is also used to create P-edges.

To improve visual clarity, nodes are colored. *or*-nodes are blue, *and*-nodes red and *basic*-nodes gray. Every second hierarchical level of nodes with the same type has a slightly darker tone. This makes nodes that have the same type as their parents more visible.

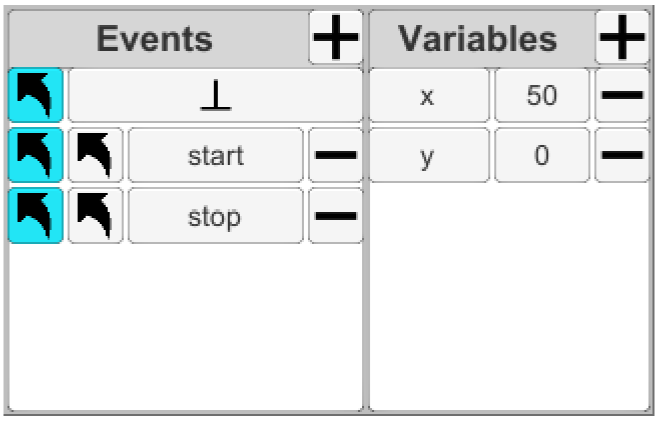


A group of nodes created with the editor. The *and*-node *AnAndNode* is a direct child of the *root*. Its own children are the *basic*-node *ABasicNode* and the *or*-node *AnOrNode*. The latter has three *basic* child nodes of which *A* is the default child. *AnOrNode* was right-clicked and its context menu is displayed. This menu does not contain the option "Set Default", because *AnOrNode* is not itself the child of an *or*-node.

Variables and Events Variables and events are displayed in scrollable lists. They can be created by clicking on the respective $+$ buttons and deleted by clicking on $-$. They can be renamed by clicking on their names. Variable types are restricted to signed 32 bit integers.

The names of nodes, events and variables must be unique. In addition, only alphanumeric names that start with a letter are accepted.

The event $\perp$ is always displayed in the list. It can neither be renamed nor removed.



In the above example, there are two events *start* and *stop*, and two variables x and y. Their values are 50 and 0, respectively.

P-Edges P-edges are created by right-clicking on the source node or pseudo-node and clicking "Add P-edge" and then clicking on the target node or pseudo-node. Finally, a text field appears and the label, i.e. the behavior, is specified in a single string. This string uses the following format: $e[g]\{p\}/A : P$

- e is the name of the triggering event. Instead of $\perp$, omitting it specifies that there is none.

- g is the guard, written in brackets. Like in the syntax specification, it consists of boolean combinations of two types of atomic clauses. It is optional; omitting it implies that $g = true$.
  In my prototype, g is defined as an expression $expr$
  $expr := -(expr) \mid (expr)\text{v}(expr) \mid (expr)\hat{}(expr) \mid \text{in}(n) \mid var \oplus z$

  $\text{in}(n)$ is short for $active(n)$ and signifies that a node n is active. $var{\oplus}z$ is a comparison $\oplus \in \{=, <, >, \leq, \geq\}$ between a variable var and the integer $z \in \mathbb{Z}$. To facilitate work with the use of a standard keyboard, - (minus), ^ (caret), v (the letter v), $<=$ and $>=$ are used in place of the unicode characters $\neg$, $\wedge$, $\vee$, $\leq$ and $\geq$.

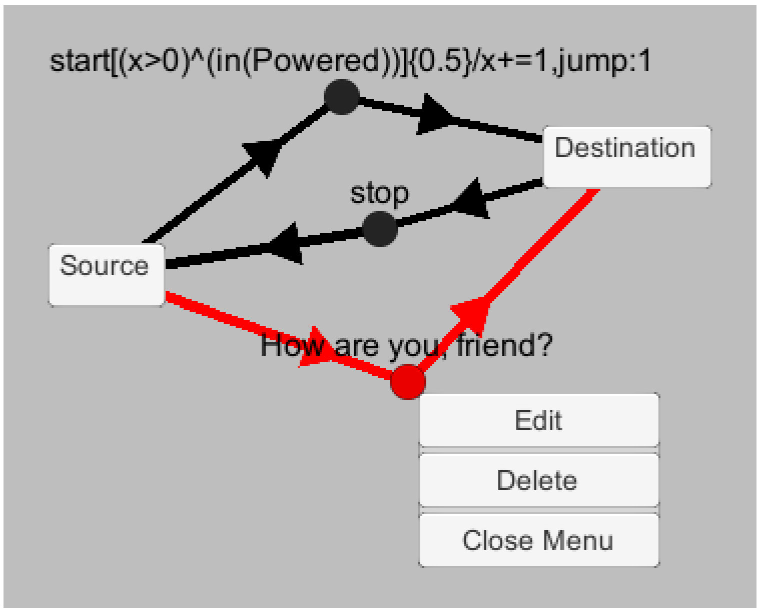
- $p \in [0, 1]$, written in curly brackets, is the probability that the edge is traversed if it is enabled and control reaches it. If omitted, $p = 1$.

- A is a list of actions separated by commas. It is preceded by a slash '/'. The following types of actions are supported:
  - e is written in place of $send(e)$ and adds the event e to the queue.
  - $var = z$ assigns the value $z \in \mathbb{Z}$ to the variable var.
  - $var+ = z$ and $var- = z$ increments/decrements the variable var by $z \in \mathbb{Z}$.

- $P \in \mathbb{N}_0$ is the priority, written after a colon ':'. A smaller value corresponds to a higher priority. Assigning a value to P that is already in use will automatically add 1 to the conflicting P-edge's priority. This is repeated until all priorities are unique. This priority can be omitted but if a conflict arises during simulation, it stops and an error message is shown.

If this format cannot be parsed due to an error, the text turns red.

Each P-edge is displayed as two arrows pointing from source to destination that are connected via a small circular button in the middle. The label is displayed above this button. This button can be dragged to deform the P-edge or right-clicked to open a small context menu. With this menu, the P-edge can be deleted or its label can be edited.



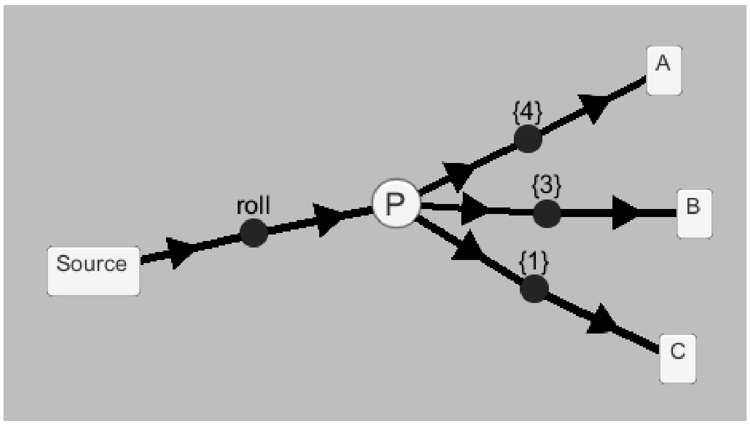
In this example, there are three P-Edges. The one that points from the node *Destination* to the node *Source* is triggered by the event *stop*. The top one that points from *Source* to *Destination* is rather complex. It is triggered by the event *start*, but only if the variable x is greater than 0 and the node *powered* is active. Its transition probability is 0.5. When traversed, it increments the variable x by 1 and fires the event *jump*. Its priority is 1. The bottom edge is red because its label couldn't be parsed. It has just been right-clicked and its context menu is shown.

Pseudo-Nodes Three types of pseudo-nodes can be created, and the respective buttons are located next to "+ Node". Pseudo-nodes are displayed as circular button with a letter on it, depending on the type. Their outgoing P-edges follow syntax rules that are specific to the pseudo-node's class, as defined in section 3.1.2.

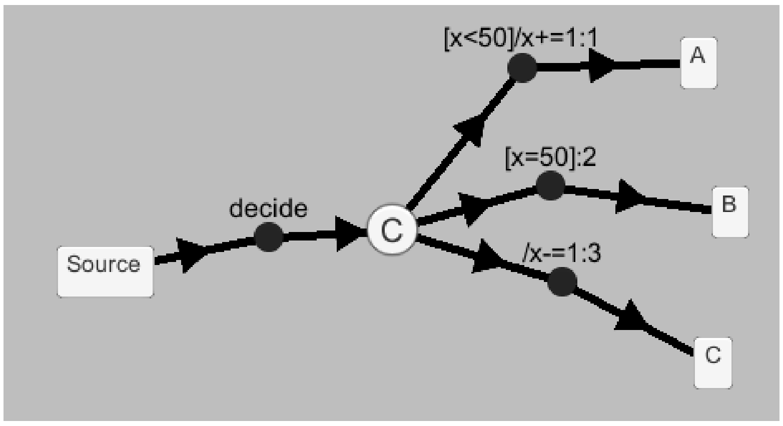
- *Weight-Nodes* (P for probabilistic) are pseudo-nodes that have a number of transitions whose probabilities are determined by their relative weights. Each P-edge has the label w/A where $w \in \mathbb{N}_0$ denotes its weight.
- *Cond-Nodes* (C) are pseudo-nodes that function similarly to a series of if-else-statements. They have one default transition that only has actions. The other transitions must have priorities and, optionally, guards. These priorities are unique per cond-node and specify the order in which they are evaluated.
- *Fork-Nodes* (F) are pseudo-nodes that model the simultaneous activation of multiple real nodes. They can have multiple edges that do not have any labels. These edges must point to real nodes that can be active concurrently. The edges are automatically simplified and made consistent.

> Consider, for example, the nodes A, B and C. A is an or-node and B and C are its children. If a fork-node already has an outgoing edge pointing to A and the user adds another edge to B, the edge to A is removed. This is because A is already implicitly a target when B is. If the user then adds another edge to C, the one to B is removed. This is because B and C cannot both be in the configuration at the same time. This logic extends to children.

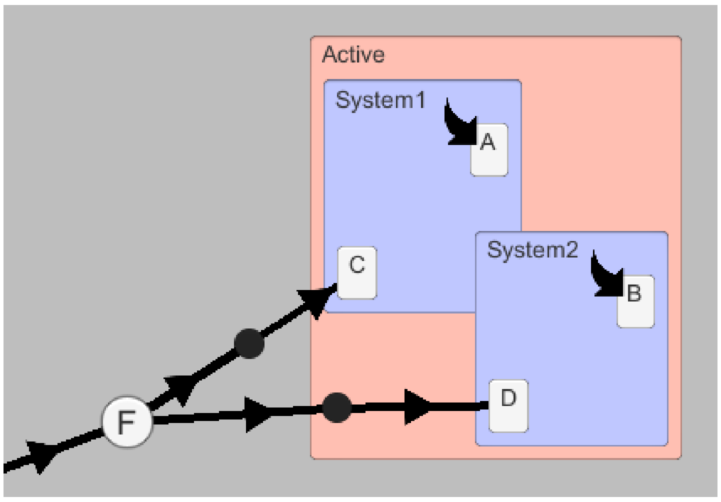
Pseudo-nodes are required to have at least one outgoing P-edge, otherwise the execution cannot start, and they turn bright red. Cond-Nodes must have a default transition. Pseudo-nodes also have a context menu with which they can be deleted or P-edges can be added to them.



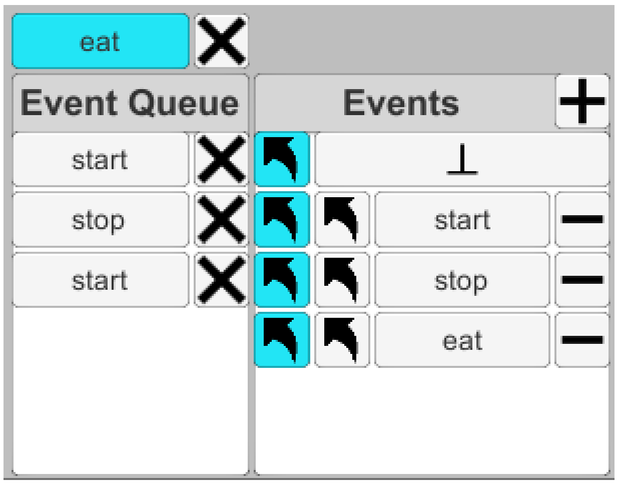
A weighted-node with three outgoing P-edges. Their respective weights are 4, 3 and 1, for a total of 8. This means their absolute probabilities are 0.5, 0.375 and 0.125, respectively.



A cond-node with three outgoing P-edges. The first has the highest priority. It is traversed if x<50. If that's the case, it increments x by 1. The second one is traversed if x=50. The third one, having the lowest priority, is the default edge and it mustn't have a guard. It is traversed if the above P-edges aren't, which is the case if x>50. If that happens, x is decremented by 1.



A fork-node that points to two nodes *C* and *D* from different subsystems. If traversed, both nodes are activated (along with their ancestors *Active*, *System1* and *System2*, to keep the configuration consistent). If it pointed to the node *Active* directly, the default children *A* and *B* of its subsystems would be activated instead (again: including their ancestors).



The queue contains the events *start*, *stop* and *start again*.

6.2.3 Location and Sub-Location Modification

Event Queue The event queue is displayed next to the events. Clicking on the black arrow button next to an event from the list adds it to the end of the queue. Clicking on the X button next to an event in the queue removes it. An event is also removed from the queue if it is deleted.

If an event is currently being processed, it is displayed right above the queue. It is set by clicking on the blue arrow button next to an event in the list. It can be removed from this position by clicking X. This will also deactivate the active pseudo-node and clear the set T of P-edges.

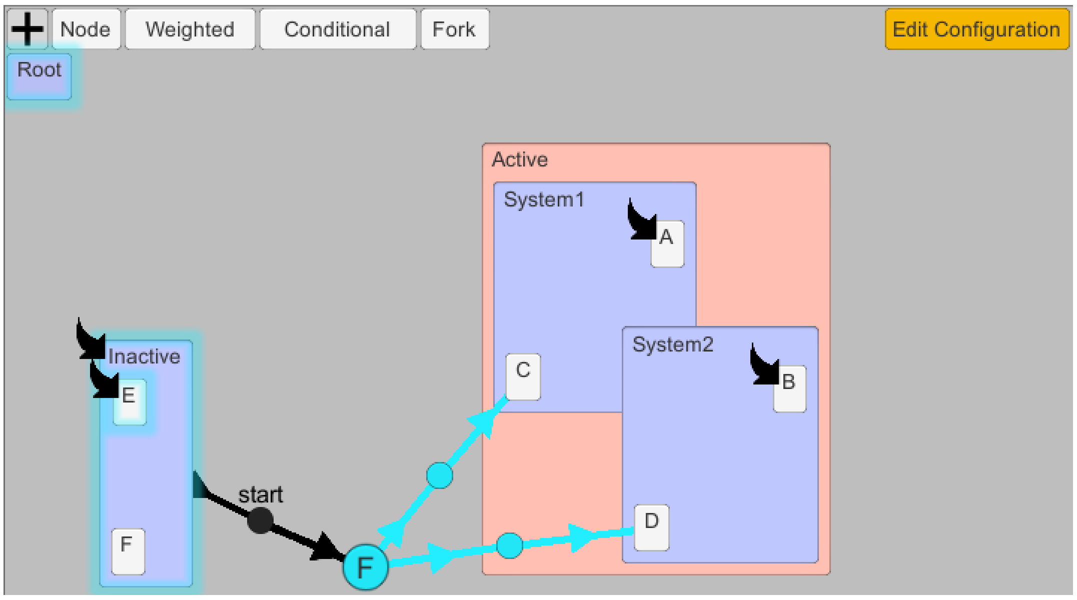
The event $\perp$ cannot be added to the queue but it can be set as the active event.

Variables Variable values are changed by clicking on the field right next to their names.

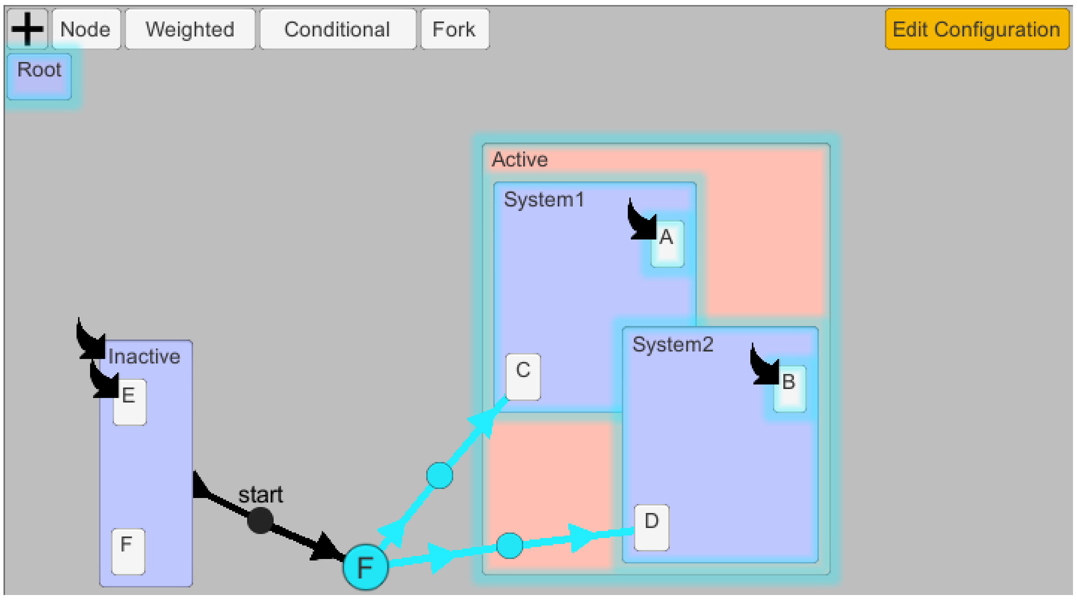
Edit Configuration Clicking on the button Edit Configuration toggles *Config Mode*. If active, clicking on a node or pseudo-node toggles whether it is active or inactive. The configuration is automatically made consistent with its invariants, according to the algorithm Activate_Nodes in section 3.3.1. Active nodes are highlighted with a cyan border.

Pseudo-Nodes In *Config Mode*, clicking on a pseudo-node sets it as the sub-location's active pseudo-node, and colors it cyan. If another pseudo-node is active at the moment, it is deactivated. Clicking on an already active pseudo-node will deactivate it. A pseudo-node can only be activated if an event is currently active.

Enabled P-Edges *Config Mode* is also used to modify the set T of enabled P-edges. If active, clicking on a P-edge toggles its status. Members of the set are cyan. P-edges that aren't triggered by the current event or ones who aren't enabled can be added to T nonetheless. They just won't be executed when control reaches them. T can only be modified if an event is currently active.



"Config Mode" is active and the button is highlighted. Currently, the nodes *Root*, *Inactive* and *E* are active and highlighted. The fork-node in the middle is also active and thus blue. Its outgoing edges turned blue automatically when it was activated. If the user clicks on the node *Active* to add it to the configuration, the outcome is the next graphic.



The node *Active* was activated. To keep the configuration consistent, its children and their default children were automatically activated as well. This means that *Inactive* and *E* had to be deactivated. As *Root* is an *or*-node, *Inactive* and *Active* cannot both be part of the configuration at once.

6.2.4 Deletion

The deletion of an object requires a number of other modifications to keep the statechart syntactically correct. These are carried out automatically.

Variable or Event Deleting a variable or an event invalidates the labels of all P-edges that refer to them. The user must then manually correct the syntax errors.

The deleted event is also removed from the event queue.

Pseudo-Node If a pseudo-node is deleted, all of its outgoing edges are deleted as well. If it was active, it will be deactivated.

Node Deleting a node is more complicated.

- If the node was its parent's default-child, one of its siblings becomes the new default-child.
- If the node was part of the configuration, it and its children are removed from it and the configuration is modified to stay consistent.
- If the deleted node's parent is an or-node, the parent's default-child takes its place in the configuration.
- If a deleted node was its parent's only child, the parent's type is set to $basic$.

Each one of the node's incoming and outgoing P-edges is deleted as well. Any pseudo-nodes that the edge points to remain unchanged, however.

Then, the node's children are also deleted and the process repeats recursively. However, other iterations do not change the configuration or default-node entries as they are already consistent at that point.

Event Deleting an event removes it from the event queue. If it is currently active, it loses that position as well.

6.3 Advancing the Simulation

Once all syntax errors are eliminated, the statechart can be simulated. There are three buttons that advance the simulation with different step sizes. The diagram is then updated to reflect the new sub-state.

The button ▶ simulates a single step according to the algorithm specified in STEP. If the step lands on a pseudo-node, the outgoing P-edge (or P-edges, if on a fork-node) that will be taken next is highlighted. This is done even if the step lands on a prob-node. To make that possible, its probability measure is immediately evaluated.

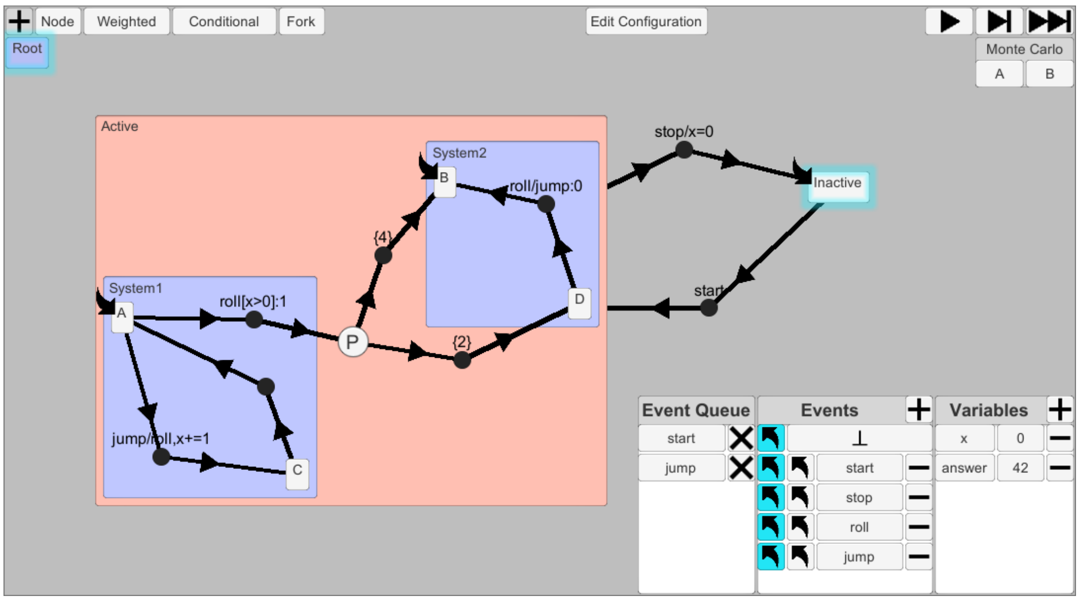
Clicking ▶| simulates steps in a loop until the statechart finishes processing the current event. If no event is currently active, the next one is first popped from the queue.

▶▶| runs the simulation until the entire event queue is processed.

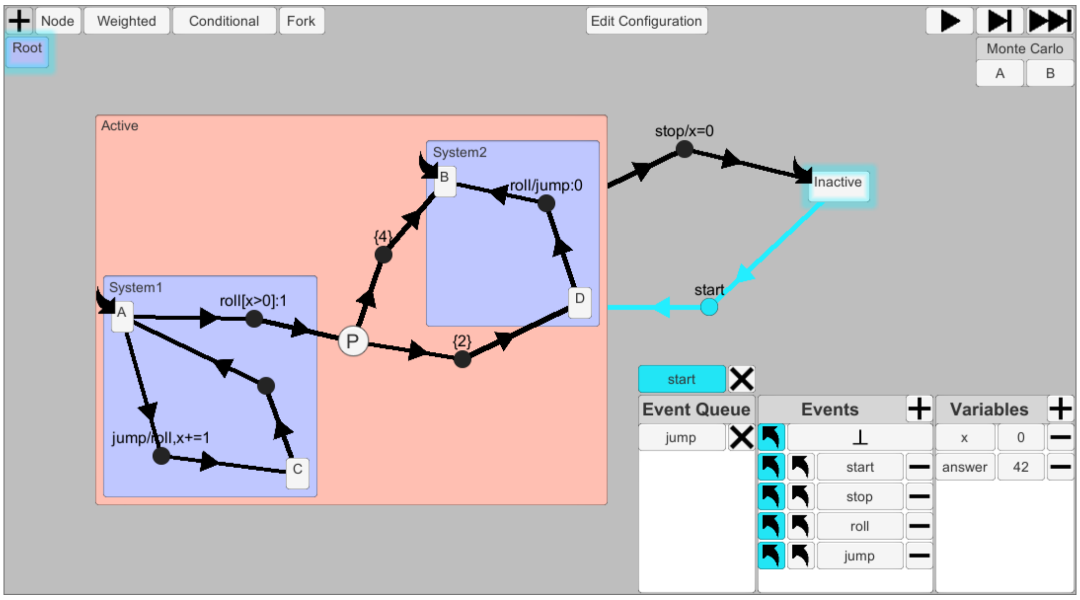
Each button simulates a (hardcoded) maximum number of 1000 steps to ensure that the execution doesn't get caught up in a never-ending loop. If the statechart is already dormant, neither of those buttons will do anything.

The following screenshots serve as an example of what stepping through a P-statechart using ▶ looks like.

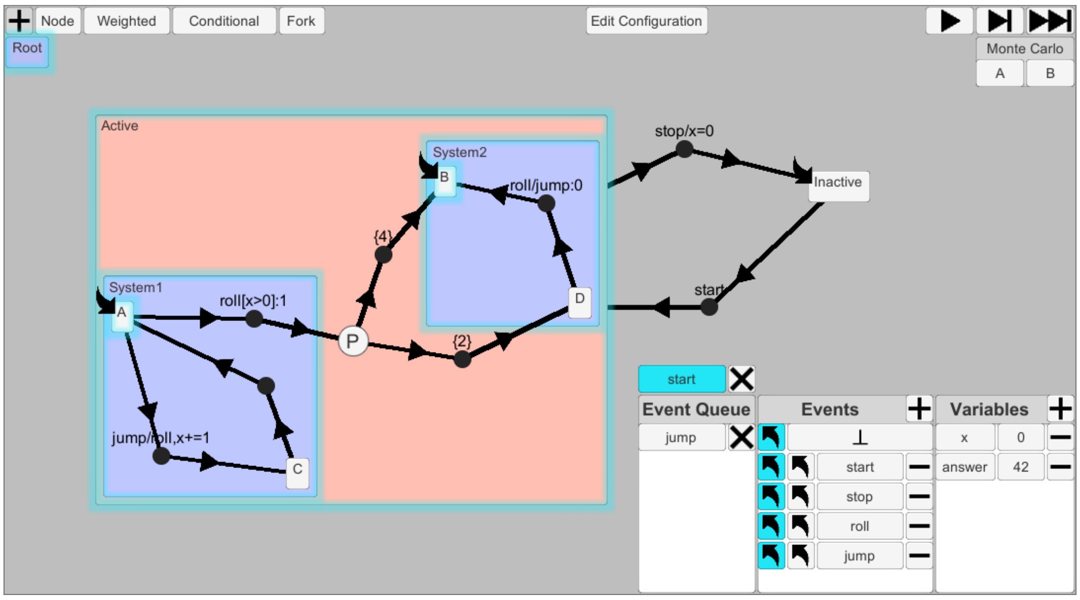
Clicking on ▶| would skip some of the intermediate steps. The resulting sequence of sub-locations would be given by the images 1, 5, 9 and 13. Clicking on ▶▶| would skip to the end (image 13) immediately.



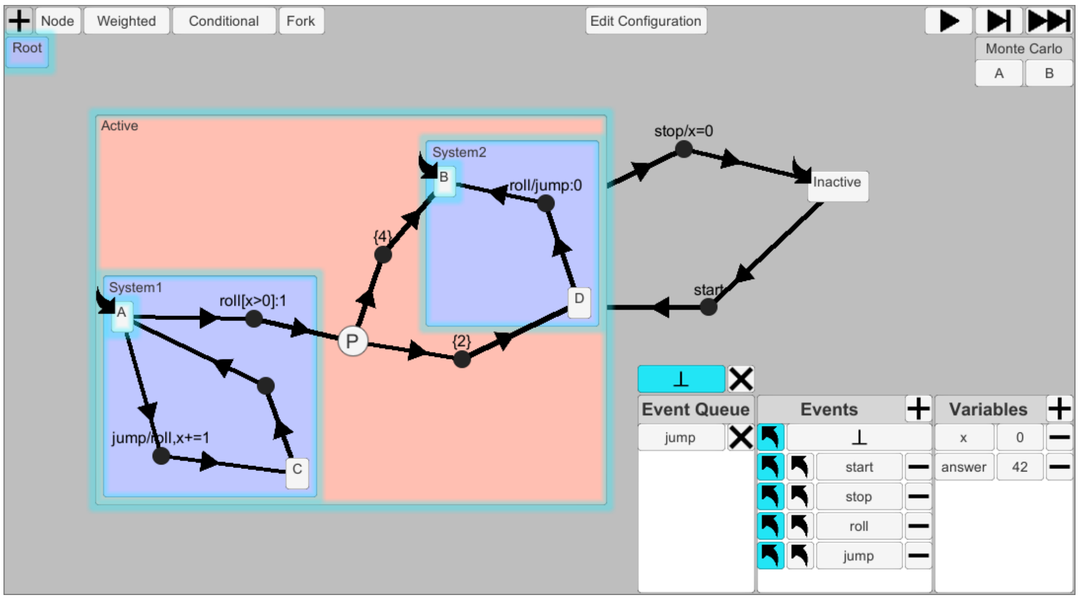
1. The initial sub-location of the P-statechart. The node *Inactive* is in the configuration (alongside *Root*, of course) and the events *start* and *jump* are in the queue. No event is currently active. This sub-location could arise if the P-statechart is initially dormant and the events *start* and *jump* are fired.



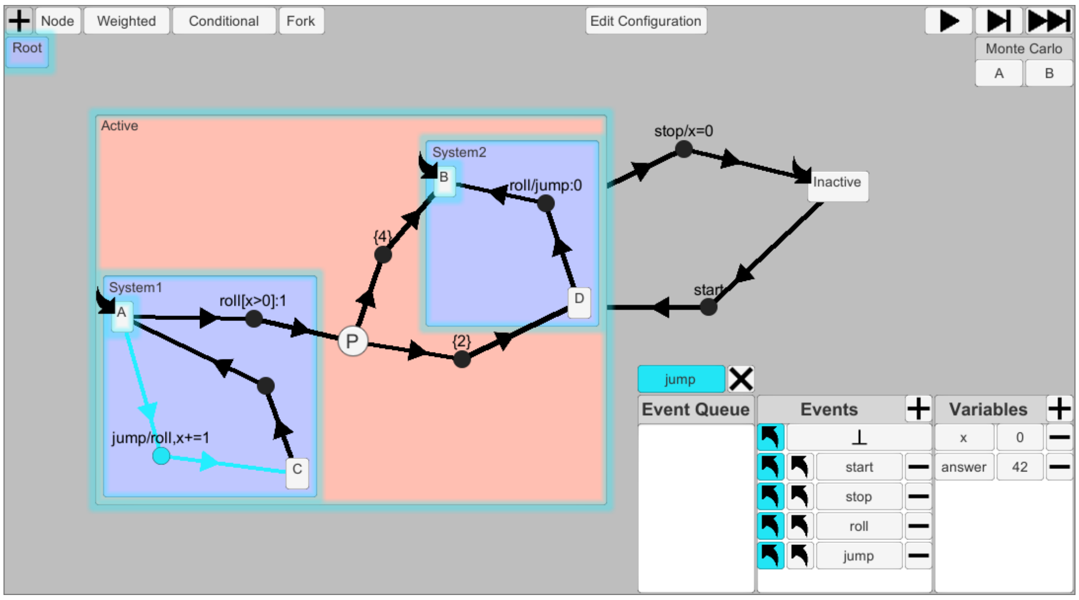
2. In the first step, the event *start* is popped from the queue and activated. One P-edge becomes enabled and is colored blue.



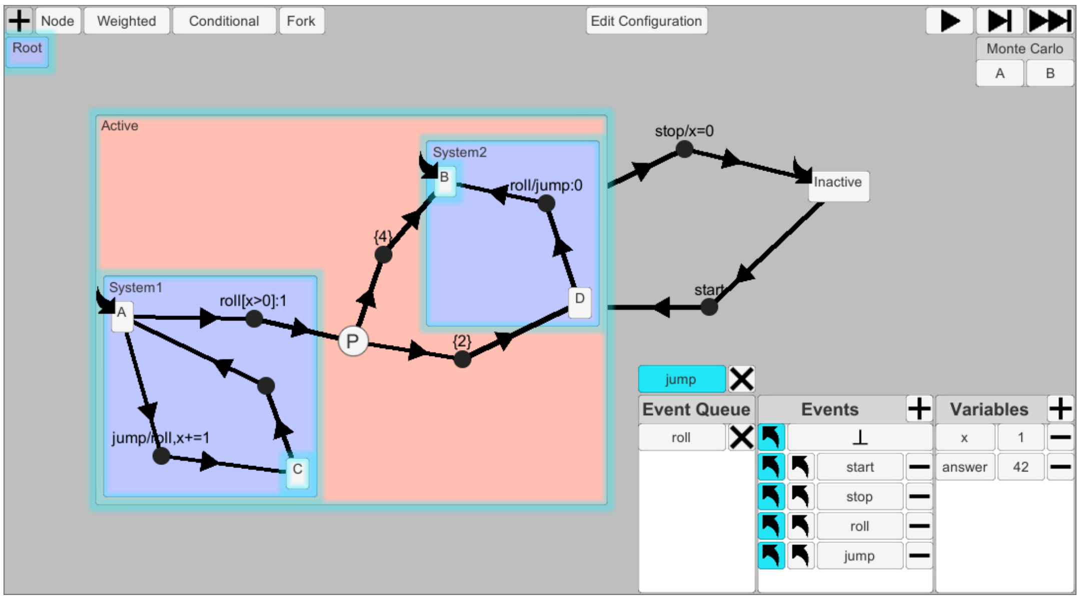
3. The highlighted P-edge is traversed and the node *Active* is activated. To keep the configuration consistent, its children and their default children are also activated. *Inactive* is removed from the configuration. No other P-edges or pseudo-nodes are active so the reaction to the event *start* is halfway done; the next step is to execute any potential edges that do not require an event.



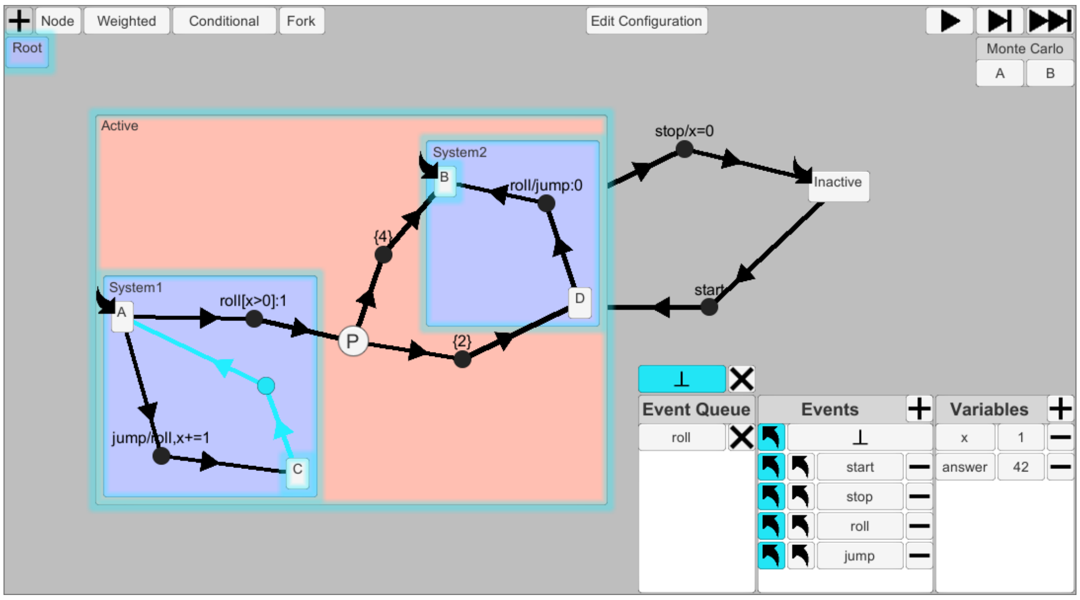
4. ⊥ is set as the active event and replaces *start*. No P-edges can be enabled at this time so nothing else happens. The reaction to the first event, *start*, is thus finished.



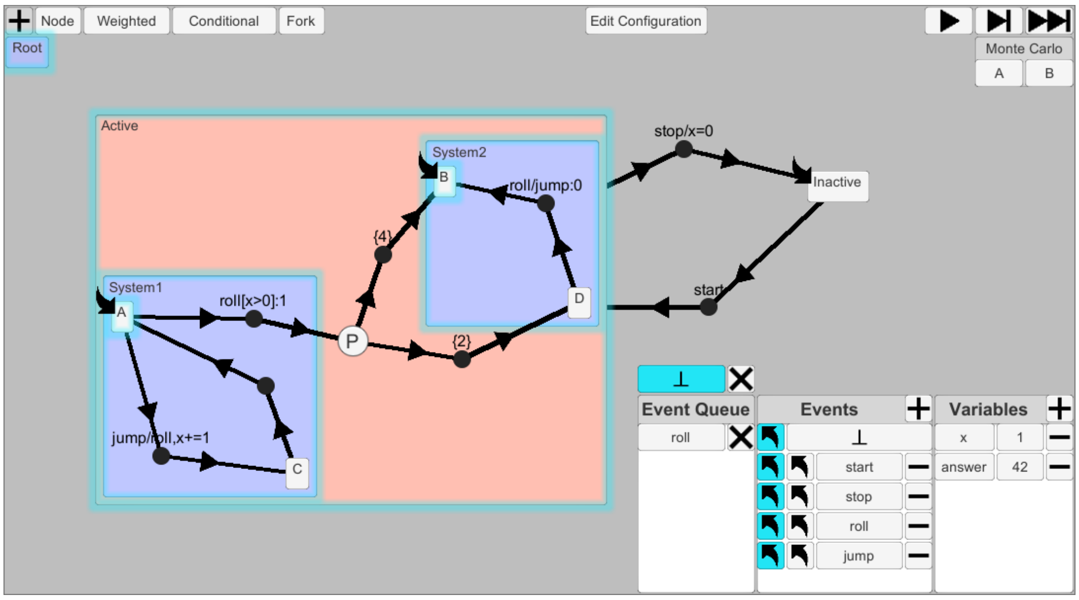
5. The event *jump* is popped from the queue and activated. In turn, one P-edge is now enabled. It has two actions: to fire the event *roll* and to increment the variable *x* by 1.



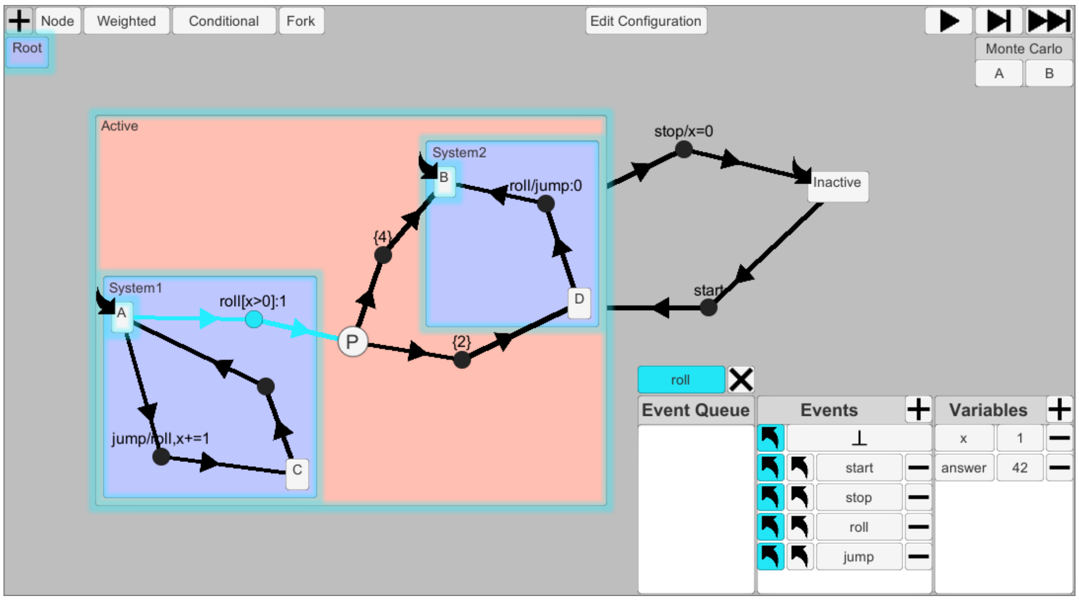
6. The edge is traversed and the node *C* replaces *A* in the configuration. Due to the P-edge's actions, the event *roll* is added to the queue and the value of *x* is now 1.



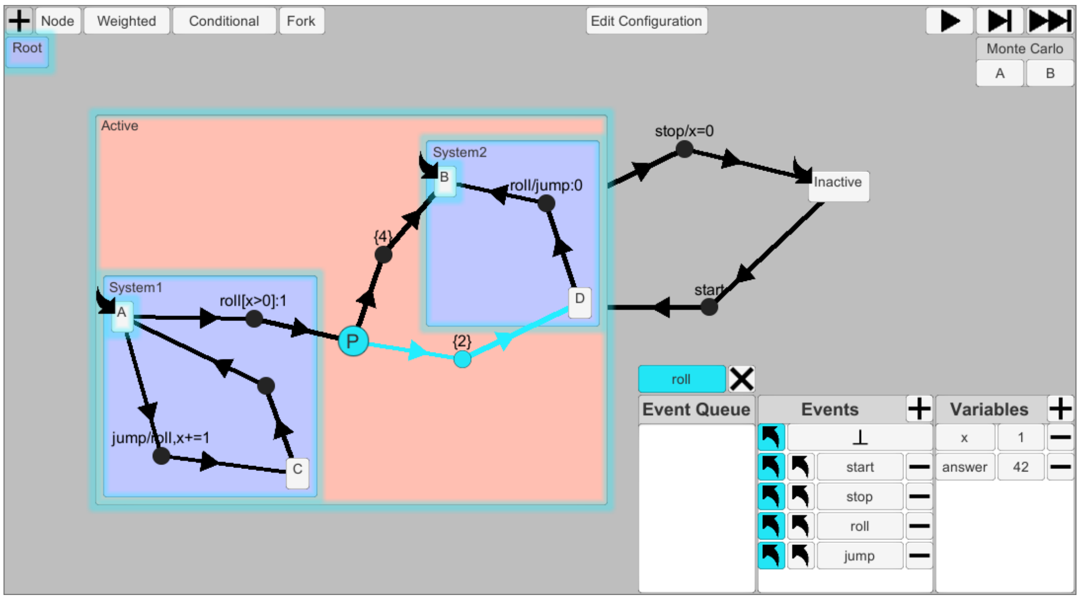
7. ⊥ takes the place of *jump* as the active event. The returning P-edge to *A* is now enabled.



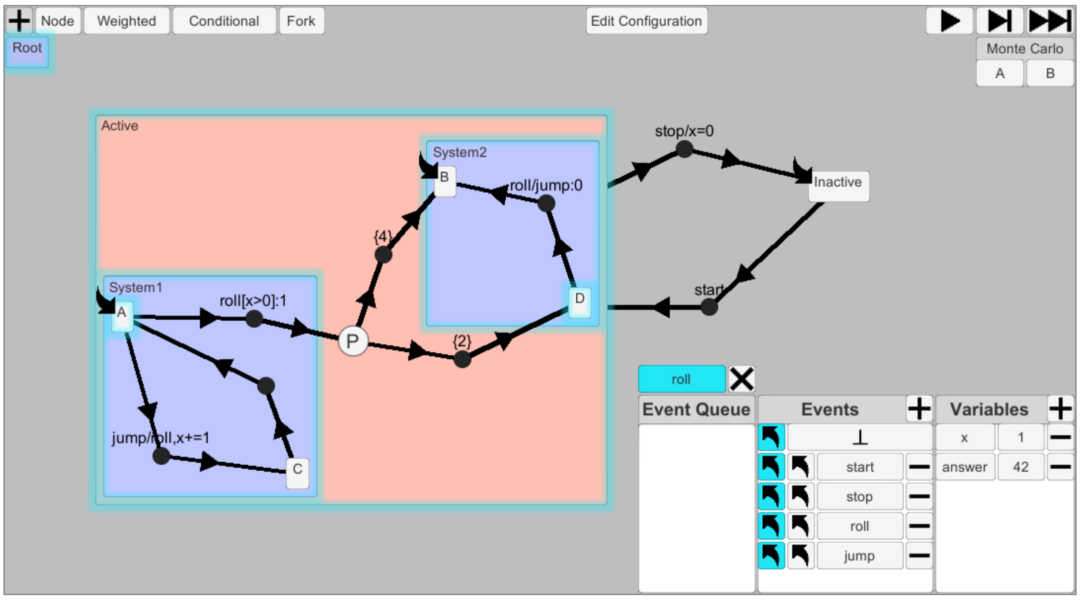
8. The P-edge back to A is traversed.



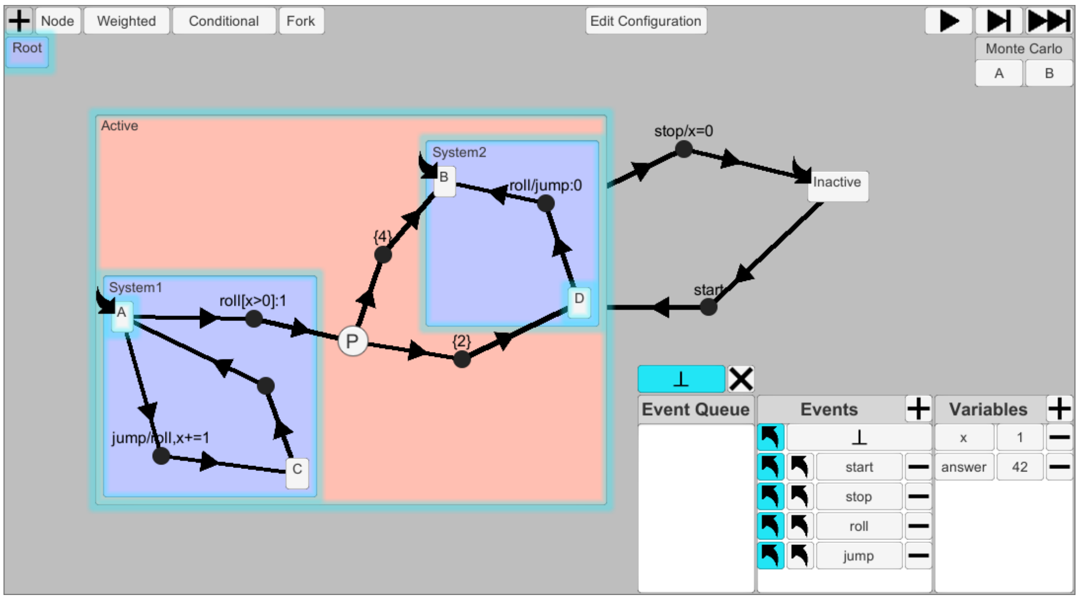
9. The event *roll* is popped from the queue and activated. The edge from A to the prob-node is enabled because its guard, $x > 0$ evaluates to $true$ here. The edge from D to B is not enabled because D isn't active.



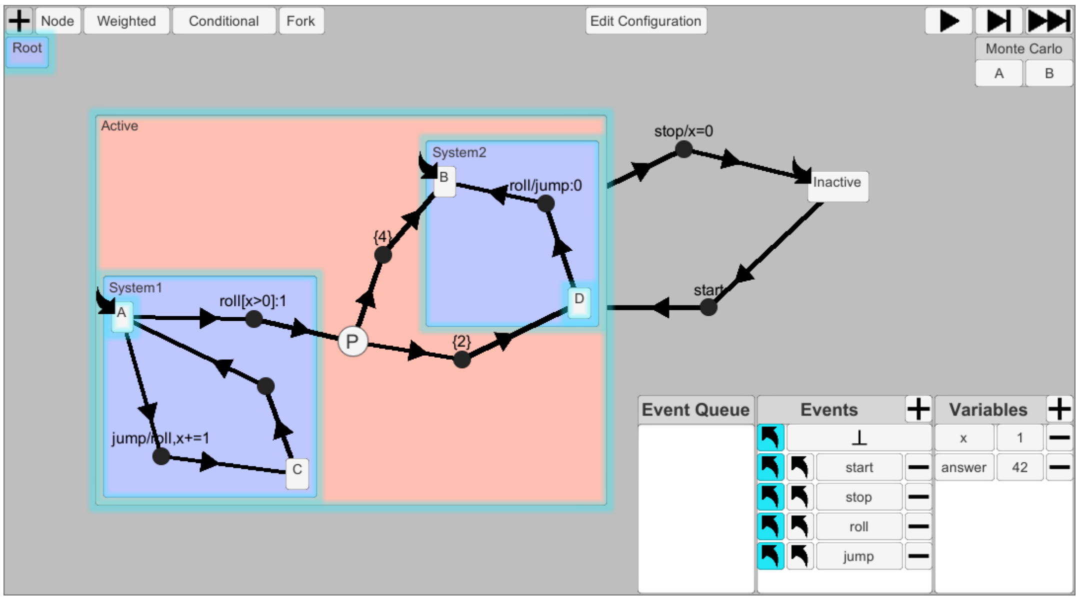
10. The edge is traversed and the prob-node is now active. It has two outgoing edges with the weights 4 and 2. The system immediately evaluates the probabilistic event and highlights the edge that will be traversed next. In this case, the edge to D is chosen. This had a $\frac{2}{4+2} \approx 33.3\%$ chance of occurring.



11. The edge to D is traversed and D replaced B in the configuration. As no more P-edges or pseudo-nodes are active...



12. ... ⊥ is now the active event. No P-edges could be enabled, however.

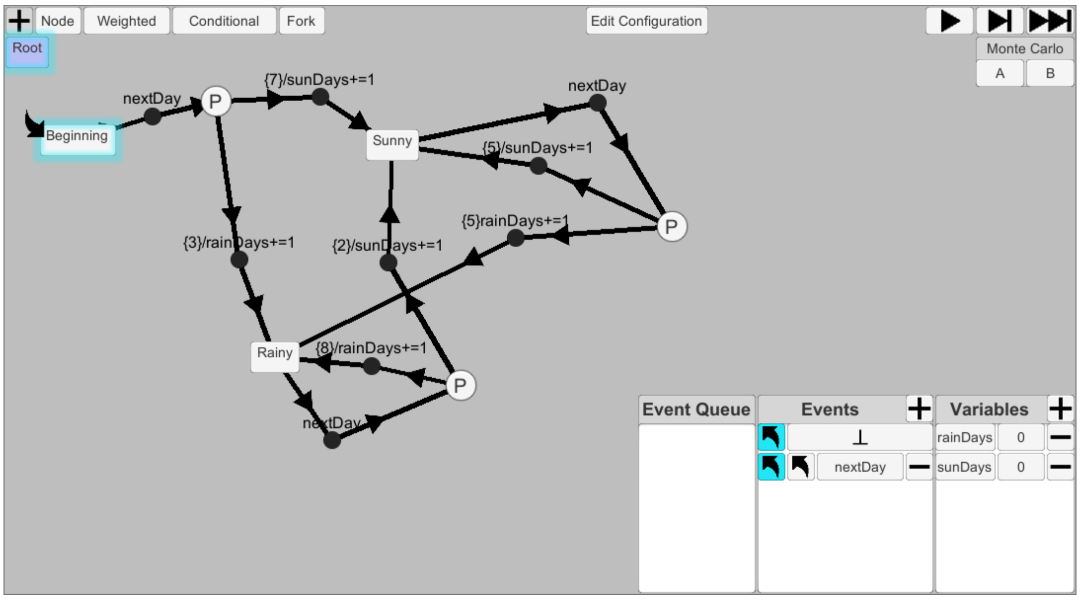


13. The current event is now *null* and the P-statechart becomes dormant.

6.4 Use of Monte Carlo Simulations

The prototype supports both methods described in section 5.2.1. There are two buttons "A" and "B" in the top right corner with which each method can be accessed. Both the sample size and the external event sequence (as a list of event names separated by commas) are specified by the user.

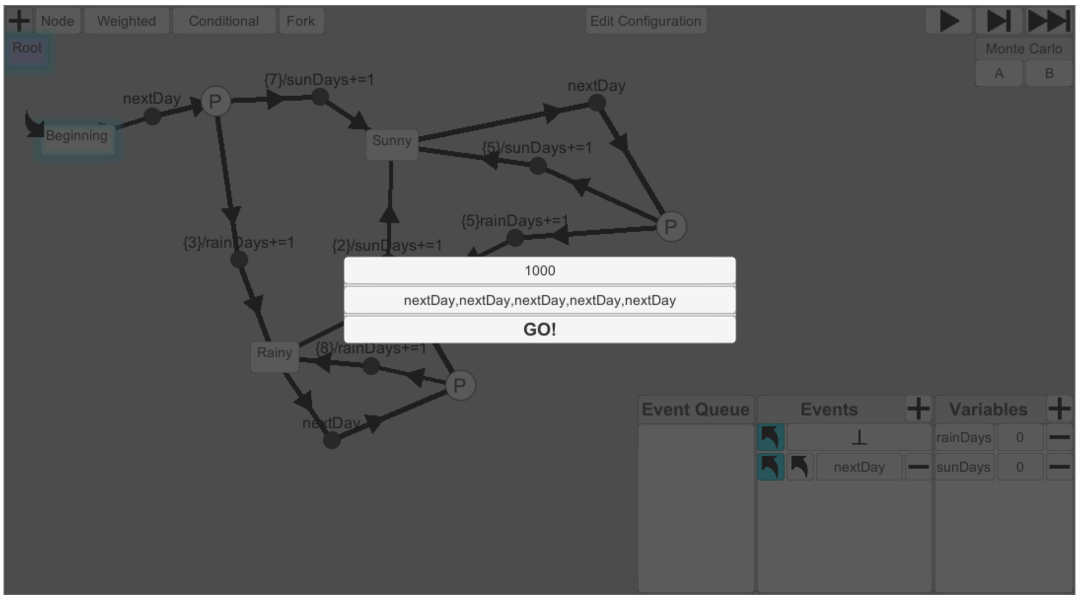
Here, the moments are defined as follows: The initial moment, moment 0, is the initial location of the P-statechart. Moment 1 is the point in time at which the initial location has been fully reacted to. This is the point in time right before the statechart starts processing the first event in the sequence. Moment t is the point in time right after the $(t-1)^{\text{th}}$ event is fully reacted to. For an event sequence consisting of n events, there are $n+2$ moments.



To illustrate this prototype's Monte Carlo features, this P-statechart is analyzed. It models a rainy working week. The probability that it rains on Monday (day 1) is 30%. If it rains on one day, the probability of it raining on the next is 80%. If it's sunny on one day, the probability of it being sunny again is 50%. The event *nextDay* is used to initiate a new day. The variables rainDays and sunDays keep track of the total number of rainy and sunny days. They are incremented every time their respective weather event occurs.

6.4.1 Method A

After clicking "A", the user is prompted to specify the sample size and the event sequence. For each sample, the statechart's reaction to the event sequence is then simulated. At every moment, data about the current location and the number of times each event was internally fired by the statechart are recorded. After the simulations, the gathered data is processed and visualized.

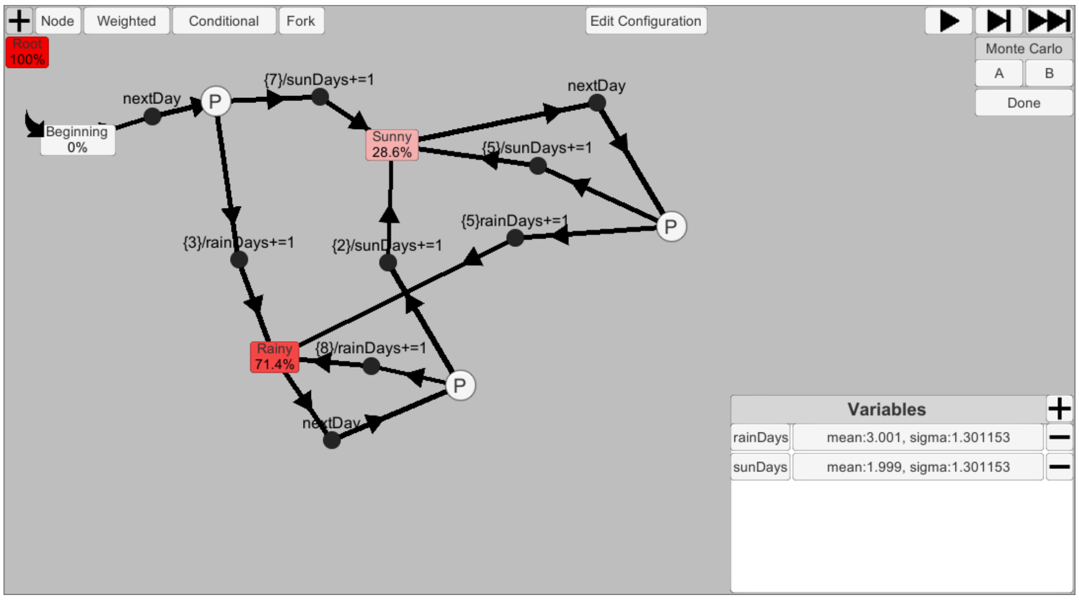


1000

nextDay,nextDay,nextDay,nextDay,nextDay

rainDays	0
sunDays	0

After clicking on the button "A", I was prompted to enter a sample size and an event sequence. I specified a sample size of 1000 and a sequence consisting of five times the event *nextDay*. Pressing "GO!" will start the simulations.



rainDays	mean:3.001, sigma:1.301153
sunDays	mean:1.999, sigma:1.301153

The result of the simulations. Of course, the node *Beginning* was never active on the last day and the node *Root* always was. Friday (day 5) was sunny 28.6% of the time and rainy 71.4% of the time. Each week had an average of 3.001 rainy and 1.999 sunny days. Of course, they add up to 5. The standard deviation of the number of rainy and sunny days is ≈1.301. It makes sense that both standard deviations are the same because data sets are flipped.

Visualization Each variable displays the mean and standard deviation of its values during the last moment. Each node displays a percentage below it that signifies the probability that it was active during the last moment. Additionally, the nodes are colored. This color depends on its probability. White corresponds to 0 and red to 1.

6.4.2 Method B

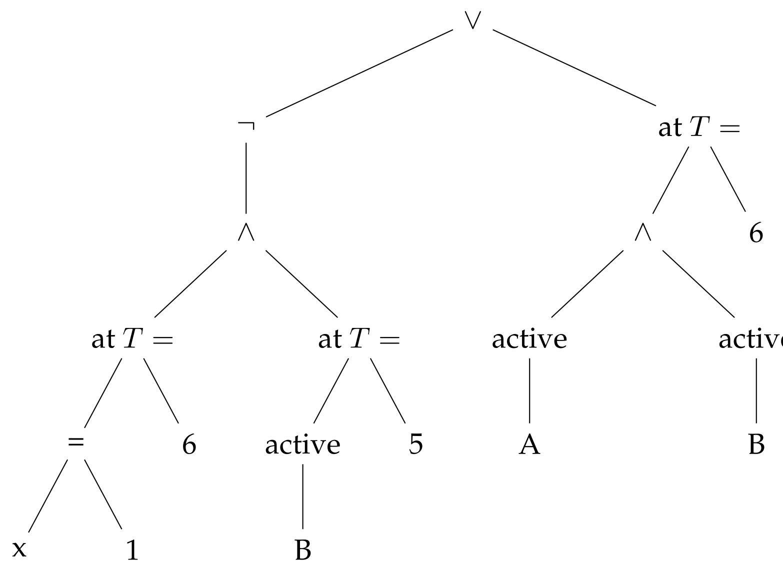
Upon clicking the button ”B”, the user is prompted with three input fields. As in the previous method, the user specifies the sample size and the event sequence. Additionally, they have to specify a *query* for the probabilistic event whose probability is requested. After parsing the inputs, the software then evaluates the query’s result and, upon completion, informs the user with a popup.

Query Specification A *query* is an expression describing the (conditional) probability of a stochastic event $sevent$. This event is either a boolean combination of sub-events or a *guard* that is bound to a moment t. A *guard* is a logic expression on the P-statechart’s location that follows the definition of guards given in section 6.2.2.

The prototype supports queries with the following syntax:
$query := \Pr(sevent) \mid \Pr(sevent \mid sevent)$
$sevent := (guard)@t \mid -(sevent) \mid (sevent)\text{v}(sevent) \mid (sevent)\hat{}(sevent)$
Like guards, the boolean operation symbols are replaced with ASCII characters and $in(n)$ is input in place of $active(n)$. Furthermore, $guard$ at $T = t$ is written as $(guard)@t$.

Implementation A query consists of the two specified stochastic events: the main event and the condition. If no conditional probability was queried, the condition is set as $true$. This is implemented by transforming a non-conditional query in the form $\Pr(A)$ into an equivalent conditional query in the form $\Pr(A \mid true)$. A stochastic event is internally modeled as an abstract syntax tree that has the same structure as the parsed input. As an example, the following query would result into the subsequent diagram for the main event:

Pr((-(((x=1)@6)^(in(B)@5)))v(((in(A))^(in(B)))@6))



Nodes that correspond to $(guard)$ at $T = t$ are the interfaces between guards and sub-events and are especially important. For each moment, there is a list of references to its corresponding nodes.

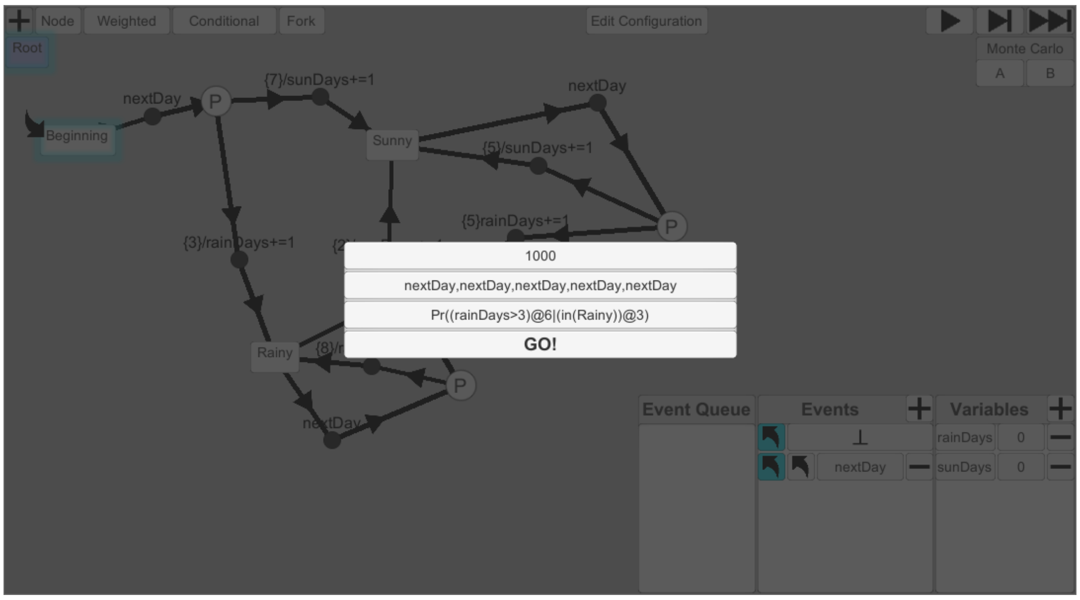
Two integer variables keep track of the simulation results. $accepted\_samples$ counts the number of samples for which the condition evaluates to true. $accepted\_samples$ counts the number of accepted samples for which the main event also occurs.

At the beginning of each sample's simulation, these trees are cloned. Every time a moment is reached, the corresponding moment-bound guards are evaluated and their nodes replaced with the respective primitives $true$ or $false$. Any stochastic events and sub-events that are now assignable are also replaced with primitives.

If, at any point, the entire query can be assigned, the simulation is stopped and the sample's result is recorded. If the condition is evaluated to $false$, the sample is immediately discarded. If the condition evaluates to $true$ and the main event is assigned, the sample is accepted and $(accepted\_samples)$ is incremented. If the main event evaluates to $true$, $true\_samples$ is incremented as well.

For example, consider the query specified above. During one sample's simulation, the point $T = 5$ is reached and B is not active. The entire tree immediately evaluates to $true$. As the condition is already implicitly $true$ (because none was specified), there is no reason to simulate the sample further. Both $(accepted\_samples)$ and $(true\_samples)$ are incremented.

Displaying the Result After all samples have been simulated, a popup informs the user of the result. It displays the evaluated probability both as a percentage and as a fraction of true samples to accepted samples. These sample counts can be used to gauge the expressiveness of the result.



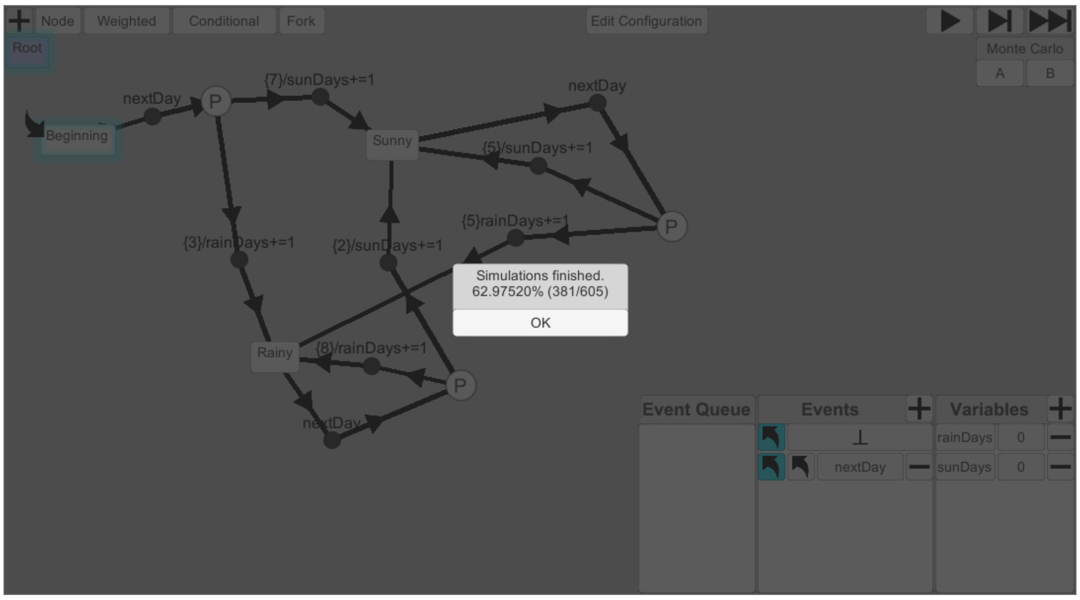
1000

nextDay,nextDay,nextDay,nextDay,nextDay

Pr((rainDays>3)@6|(in(Rainy))@3)

GO!

After clicking on the button ”B”, I was prompted to enter a sample size, an event sequence and a query. The first two inputs were the same as in method A. For the query, I specified, in words ”What is the probability that a working week has more than three rainy days if it rains on Tuesday?” Note that T=3 is the moment immediately after the second event in the sequence is processed. Pressing ”GO!” will start the simulations.



Simulations finished.
62.97520% (381/605)

I am prompted with the results. In 605 simulations, it rained on Tuesday. In 381 of them, there were more than three rainy days in the week. The probability was thus evaluated to ≈ 62.975%.

7 Discussion and Outlook

This thesis explored ways to test and verify systems that are modeled with probabilistic state diagrams. The focus was on probabilistic extensions of David Harel's statecharts. This is because I value their inclusion of orthogonality and variables highly, especially in the context of video game development. In related literature, probabilistic statecharts (P-statecharts) are not defined uniformly. Furthermore, their underlying models have to make use of pseudo-nodes in order to be visualized. In addition, to resolve conflicts between multiple possible edges/transitions, they either allow non-determinism or they make use of unintuitive priority measures.

The dialect of P-statecharts used in this thesis was designed as a bridge between the way a developer would think about specifying a P-statechart and the internal model used by a machine that executes it. I define two variants, one of which is closer to the definitions used in related works. In the other one, pseudo-nodes are explicitly part of it and the priorities of conflicting edges are stated directly.

A prototype development and testing environment for P-statecharts was created. With it, P-statecharts can be created and simulated. The simulations can be carried out step-by-step, allowing for full control over the system's internal structure. Alternatively, multiple simulations can be carried out in bulk. This allows for the evaluation of specified probabilities using the Monte Carlo method.

Testability Probabilistic state diagrams can be effectively tested in various ways. Simulating them one step at a time is very intuitive and makes full use of their visual and intuitive nature. The fact that the system's entire internal state can be depicted in one diagram makes it trivial to see exactly what changes from one moment to the next.

The Monte Carlo method proved to be a quick and effective way to test certain properties that the system should have.

Next Steps During development I found myself implementing the different pseudo-node types as explicitly different data types. This stands in contrast to the way I defined pseudo-nodes in this thesis. The different types of pseudo-nodes arise based on which properties of a pseudo-node are expressed. This means that my definition had to include all of the functionality that any type of pseudo-node could use. Using all of its functionality simultaneously would lead to a pseudo-node that isn't any more intuitive than the P-statechart syntax used in other literature. Instead of this, it could be valuable to define pseudo-nodes in a more abstract way that allows the types to be different implementations of it.

The most important feature the prototype is missing is a function to import or export the constructed P-statecharts. It is obviously crucial to be able to save one's progress. In addition, it should be possible to export the P-statecharts in a model that can serve as the input of other model checkers. This could include an automatic conversion from P-statecharts to Markov Decision processes.

Another great feature would be ability to import formats that are already in use, such as the XML format mentioned in [CFN10]. This would open up the prototype to various types of state diagrams that were created with other tools.

Finally, it would also be valuable to fully integrate the prototype into the development of a software, especially a small video game. This would give a better intuition on what the workflow with probabilistic state diagrams is like.

Bibliography

[ABC14] Etienne Andre, Mohamed Mahdi Benmoussa, and Christine Choppy. Translating uml state machines to coloured petri nets using acceleo: A report. https://arxiv.org/pdf/1405.1112.pdf, 2014.

[ans17] ans_unity. Dots visual scripting first experimental drop. https://forum.unity.com/threads/dots-visual-scripting-first-experimental-drop.677476/, October 12, 2017.

[Bru18] Bernd Bruegge. Testing. Technical University of Munich, Lecture on Introduction to Software Engineering, July 5, 2018.

[CFN10] Franco Cicirelli, Angelo Furfaro, and Libero Nigro. Modelling and simulation of complex manufacturing systems using statechart-based actors. https://www.sciencedirect.com/science/article/pii/S1569190X10002224, 2010.

[cry] Cryengine. https://www.cryengine.com/. Version 5.6.7 (Computer software).

[Dic20] Tutorialspoint Software Testing Dictionary. Passive testing. https://www.tutorialspoint.com/software_testing_dictionary/passive_testing.htm, 2020.

[Dru94] Doron Drusinsky. Extended state diagrams and reactive systems. https://calhoun.nps.edu/bitstream/handle/10945/60583/Drusinsky_Extended_State-Times_Mag.PDF?sequence=1, 1994.

[Dyr20] Daniel Dyrda, November 6, 2020. (personal communication).

[Har86] David Harel. Statecharts: A visual formalism for complex systems. https://www.inf.ed.ac.uk/teaching/courses/seoc/2005_2006/resources/statecharts.pdf, 1986.

[JHK02] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A probabilistic extension of uml statecharts. https://link.springer.com/chapter/10.1007/3-540-45739-9_21, 2002.

[JHK03] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A qos-oriented extension of uml statecharts. https://link.springer.com/chapter/10.1007/978-3-540-45221-8_7, 2003.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: A hybrid approach. https://link.springer.com/chapter/10.1007/3-540-46002-0_5, 2002.

[min] Minecraft java edition. https://www.minecraft.net/en-us. Version 1.16.4 (Computer game).

[OUP20a] Lexico.com Oxford University Press. Debugging. https://www.lexico.com/definition/debugging, 2020.

[OUP20b] Lexico.com Oxford University Press. Test. https://www.lexico.com/definition/test, 2020.

[SL08] Jun Sun and Yang Liu. Model checking csp revisited: Introducing a process analysis toolkit. https://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=6053&context=sis_research, 2008.

[uni19] Unity. https://unity.com/, 2019. Version 2019.4.1f1 (Computer Software).

[unr] Unreal engine. https://www.unrealengine.com/en-US/. Version 4.25 (Computer Software).

[VCAA05] N.L. Vijaykumara, S.V. Carvalhoa, V.M.B. Andradeb, and V. Abdurahiman. Introducing probabilities in statecharts to specify reactive systems for performance analysis. http://www.lac.inpe.br/~vijay/download/Papers/COR_2006.pdf, 2005.

[Wik20] Official Minecraft Wiki. Advancement/json format. https://minecraft.gamepedia.com/Advancement/JSON_format, 2020.

[ZL10] Shao Jie Zhang and Yang Liu. Probabilistic symbolic model checking with prism: A hybrid approach. https://ieeexplore.ieee.org/abstract/document/5521548, 2010.