Simulation of Deterministic and Parallel Execution of Extended Symbolic Transition Systems

Ing. Martin Decker, BSc

martin.decker@v2c2.com

Institute for Software Technology, Graz University of Technology

VIRTUAL VEHICLE Research Center

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Institute for Software Technology
Graz University of Technology
Assistant Supervisor: Dipl.-Ing. Dr.techn. Christian Schwarzl
VIRTUAL VEHICLE Research Center

October 10, 2013

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________________ ______________________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ______________________________ ______________________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Cyber-Physical Systems (CPSs) where the interacting subsystems are connected by a network have, beside the communication with the environment, also communication within the system. Therefore, the different communications have to be defined and tested carefully. It is a convenient way to create models for such systems before the implementation using Unified Modeling Language (UML) State Machines. These models can be transformed to Extended Symbolic Transition Systems (ESTSs), which can be simulated without generating program code. The big advantages of simulation in comparison to executing program code is the time handling and handling of non-determinism. The simulation calculates the time and consider all possible executions and therefore there are more possibilities to detect undesired behaviors like races or deadlocks. In contrast, the execution of code selects always one out of all possibilities.

We consider three different communication modes for the communication between the subsystems namely Deterministic, Full Interleaving and Full Parallel. Deterministic and Full Interleaving executes transitions sequentially. In contrast to Deterministic, where only one subsystem at a time is executed, Full Interleaving executes the subsystems simultaneously which can lead to non-determinism also for deterministic models. The Full Parallel approach executes the subsystems simultaneously where transitions can be executed at the same time in parallel. Furthermore, time is considered in all simulations, where every transition has an execution time. This is the time that is elapsed by a state change triggered by the transition. By this means a system can be simulated using different communication modes and execution times, which allows to get the information about the consequences of different simulation setups to the output of the system.

The impact of different executions due to the selection of varying inputs and execution times leads to possible different execution traces. These traces have an impact on for example co-simulations, where different model types are simulated cooperatively. An interface to the co-simulation framework Independent Co-Simulation (ICOS) was created, which offers a very uncomplicated way to use ESTSs in ICOS.

Contents

List of Figures

List of Algorithms

1. Introduction

1.1. Motivation

Cyber-Physical System (CPS) consist of embedded computers which are connected via networks to physical components to control physical processes [19]. The fields of application increases very fast for CPSs in general and especially in the automotive industry. There are many such systems in use like Anti-lock Braking System (ABS) or air bags. Also, a lot of researching is done in this area, namely Car-to-X, where cars communicate with other cars, traffic lights and others. An additional application is drive by wire, where mechanical connections are replaced by mechatronic connections. It is required that such systems work correctly because a bug can cause great harm.

It is a challenge to add for example a new embedded control unit to an existing system of physical components. The control unit can be for example a controller of the airflow for the engine of a car. In such a system a number of subsystems are executed concurrently connected via a network. Depending on the addressed problem set different types of communication between the CPS components have to be considered.

There exist tools like Rhapsody®[1], Enterprise Architect™[2] or Visual Paradigm®[3] where UML models can be created and for which code can be generated. This code is executable and is used for simulation and for a graphical animation of the models. Due to the different communication modes and time behavior options the simulation of a model has advantage in comparison to execute generated code from a model.

A co-simulation framework like ICOS[4] couples different simulation tools. Therefore, a system which consists of subsystems of different domains can be co-

[1] http://www.ibm.com/software/awdtools/rhapsody/

[2] http://www.sparxsystems.com/ea

[3] http://www.visual-paradigm.com/

[4] http://www.v2c2.at/icos

simulated. For example, ICOS can co-simulate a system of a virtual vehicle including different parts of a vehicle, the driver and the environment [26].

1.2. Contribution

The aim of this work is the simulation of concurrent models with different communication modes by considering execution time. The simulated models are ESTSs [23, 22] which can be transformed from UML State Machines or can be created manually. Simulation means the execution of enabled outgoing transitions of states where the execution leads to an execution tree. In addition, an interface to ICOS is described that is able to co-simulate different simulators including Hardware in the Loop (HiL) simulators.

The simulation provides three communication modes to be able to simulate different compositions of subsystems in a system namely Deterministic, Full Interleaving and Full Parallel. Deterministic and Full Interleaving are for sequential execution of ESTSs, where the deterministic mode simulates a nonpreemptive multitasking processor and Full Interleaving calculates all possible execution traces for a preemtive multitasking processor. Full Parallel executes the ESTSs simultaneously in parallel considering times. Therefore, multiple transition executions at the same time are possible, like in multi-core processors.

Every transition of an ESTS consists of an execution time and because of the possibility of defining lower and upper bounds a transition consists also of an execution time range. The simulation provides options to use different time behavior alternatives. This means the lowest, the highest or a random execution time within the execution time range can be used.

The execution of the ESTS simulators are done by three simulation executors namely Random Walk, Linear Walk and Manual Walk. The Random Walk is an automated run which generates inputs for the simulated ESTSs. The impact of different simulation modes can be discovered with the Linear Walk. With this simulation executor precalculated inputs are used for simulating and the generated outputs can be compared with the different runs. The Manual Walk gets inputs from an external program and sends it to the simulator. The generated outputs are sent back to the external program.

The ICOS interface was implemented in this work to couple ESTS simulators. With respect to CPS an ICOS co-simulation can for example look as follows. The software components are simulated with ESTS simulators and the physical components are simulated by domain specific simulator tools or they are coupled via HiL simulators.

1.3. Outline

The remainder of this paper is organized as follows: Chapter 2 defines the semantics of an ESTS and Chapter 3 describes the simulation of ESTSs. The possibility to be able to nest such simulations is explained in Chapter 4. The simulation executors are presented in Chapter 5 and Chapter 6 is about the implemented interface for ICOS. The experimental results of this approach are shown in Chapter 7. This thesis is closed with Chapter 8 which includes the related and future work as well as the conclusion.

2. Extended Symbolic Transition System

An ESTS describes a system behavior similarly to an UML State Machine. In general, an ESTS is an extended symbolic version of a Timed Labeled Transition System (TLTS) [18]. It relies on timing groups and transition execution times to describe progress in time and to handle clocks.

In this section we give an overview of the essential definitions of [23, 22] which we use in following sections of this work.

2.1. Structure

An ESTS contains similar components in comparison to an UML State Machine. This allows an easy model transformation of the UML State Machines into ESTSs. As the name already suggest is ESTS an extension of Symbolic Transition System (STS) [11, 12], where STS was extended by timing group, delay and completion transition, transition priority and transition execution duration.

Definition 2.1. (Extended Symbolic Transition System)
An Extended Symbolic Transition System consists of a set of states, a set of labels, a set of attributes, a set of signal parameters, a set of transitions, a set of timing groups and the initial configuration. □

The set of labels contains the labels for input and output transitions used for communication. Labels which are representing unobservable and completion transitions and the delay (natural numbers without 0) which labels delay transitions.

Attributes and signal parameters are variable sets and are used for the symbolic treatment of data. Attributes are properties of an ESTS and parameters are part of signals.

Definition 2.2. (Transition)
A transition consists of source and target state, a label, a guard, an attribute update function, a priority and an execution duration with lower and upper bound. □

The priority defines an execution order in the case a state has more than one outgoing transitions. The execution duration defines the elapse of time of the configuration change defined by the transitions. The lower and upper bound can be used to define a scope around the execution duration. The time lapse of the transition has to be in this scope.

Definition 2.3. (Variable Valuation)
A variable valuation is an ordered pair of a variable and a value. □

Corresponding to Definition 2.3 we use attribute valuation and parameter valuation to refer to variable valuations of attributes and signal parameter.

Definition 2.4. (Parameterized Input and Output)
A parameterized input consists of an input label and its possibly empty parameter valuation. A parameterized output is defined in the same way, where an output label is used instead of the input label. □

Definition 2.5. (Timing Group)
A timing group consists of a clock, a set of states, a set of delay transitions and a set of clock reset transitions. □

A timing group defines a set of states sharing the same clock that is used to keep track of the elapsed time within the timing group. There exists a clock valuation containing a variable valuation for each timing group clock and for the simulation clock. The simulation clock keeps track of the elapsed time of all ESTSs.

A clock reset transition is able to reset the clock of the timing group, where a reset means the clock valuation will be set to zero.

Definition 2.6. (Configuration)
A configuration consists of a state, an attribute valuation and a clock valuation.
□

The initial configuration of an ESTS holds the initial state, the initial attribute valuation and the initial clock valuation.

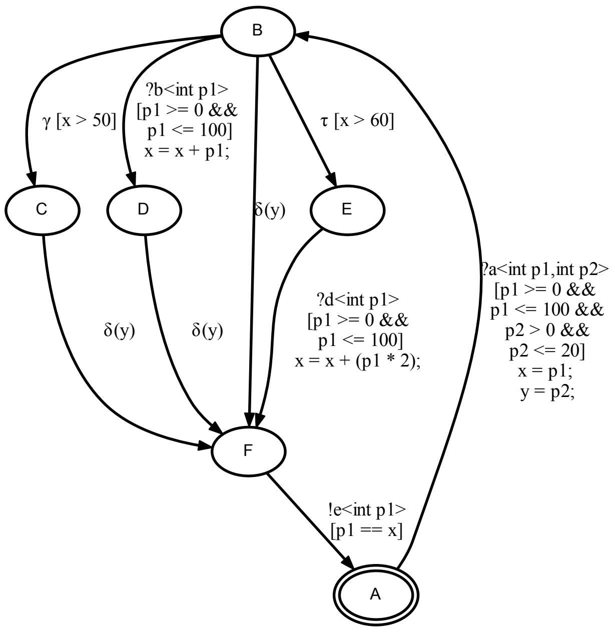


Figure 2.1.: Example of an ESTS.

Example 2.1. (Extended Symbolic Transition System)
ESTSs can be displayed as graphs, where nodes represent a state and edges are transitions. The double framed node is the start node. The transitions are indicated with symbols, where **!** stands for an output transition, **?** for input transition, γ for completion transition, τ unobservable transition and δ for delay transition. The guards of the transitions are displayed in square brackets and the signal parameter in angle brackets.

In Figure 2.1 an ESTS is shown which consists of states `A`, `B`, `C`, `D`, `E` and `F`, attributes `x` and `y` and nine transitions. There are three input transitions which are labeled with `a`, `b` and `d`, an output transition `e` and they all have signal parameters. The attributes `x` and `y` will be set in the update functions of the input transitions. The states `B`, `C` and `D` are part of a timing group and have delay transitions to state `F`. The delay transitions have a variable delay that is based on the attribute `y`.

This model is non-deterministic because the guards of the completion and the unobservable transition are evaluating to true for the case the attribute `x` is greater than 60. □

3. Simulation

The simulation of one or more ESTSs is defined in this chapter which calculates as a result an execution tree. In the case the simulated ESTS is deterministic, the execution tree consists of one execution path. Otherwise, for a non-deterministic ESTS the execution tree can hold a number of different execution paths.

The simulated ESTSs are able to communicate with each other, where input and output transitions are used for communication. An output transition of an ESTS sends a message to another ESTS which can receive it in the case a corresponding input transition exists.

Messages also can be sent to ESTSs from the environment. That means the simulation is controlled from an external source which can be for example a user who sends messages a random message generator or predefined sequences of messages.

Some of the definitions in this chapter are redefined from [8].

Definition 3.1. (Transition Execution)
An execution of a transition is a state change with an update of the attributes corresponding to the attribute update functions of the transition. Moreover, the simulation clock and the corresponding timing group clocks are updated by the time the transition need for execution. □

Definition 3.2. (Blocked State)
A state is blocked if at least one input or delay transition exists where the state is their source state. □

Definition 3.3. (Stopped State)
A state is stopped if no transitions, where the state is source state, can be executed and the state is not blocked. □

Definition 3.4. (Message)
A message extends the parametrized input and output (Definition 2.4) by a send time. This is the time the message was sent and the time it can be received from an ESTS. □

A message is produced by executing an output transition, where the send time gets the value of the simulation clock after executing the transition. In addition, a message can be generated from the environment. An input transition can be only executed if a corresponding message was sent and the simulation time is advanced enough to be greater or equal to the send time of the message.

3.1. Execution Tree

The result of the simulation is an execution tree which consists of one or more execution tree nodes. An execution tree includes the calculated execution paths of the simulated ESTSs. This structure was also defined in [8], for the sake of completeness we redefine it in this section.

Definition 3.5. (Execution Tree Node)
An execution tree node consists of a set of states, an executed transition or an empty transition, a possible empty list of messages, an attribute valuation, a clock valuation and a possible empty set of child execution tree nodes. □

An execution tree node represents a snapshot of the simulation of ESTSs at a given point in time. The set of states contains at most one state for each simulated ESTS. Non-deterministic executions result in multiple execution tree nodes. These states and their transitions are the basis for further executions. In the case the execution tree node is the root execution tree node or the simulator did only elapse time the executed transition is empty. Otherwise, a transition was executed, the transition is held in the execution tree node. The executed transition is needed to know the reason of the state change. The messages are ordered in the list like a queue because of that they are processed in the order they were added. In the attribute valuation the values of all attributes of all simulated ESTSs are held. Also, the clock valuation holds the values of all timing group clocks and the simulation clock. An execution tree node is able to hold child execution tree nodes and we call it leaf execution tree node in case the set of child execution tree nodes is empty. For simplicity we use in the remainder of this document *node* for execution tree node.

Definition 3.6. (Execution Path)
An execution path is an ordered list of nodes which represents a path through the execution tree without a branch. The beginning is always the root node of the execution tree and the end is a leaf node. □

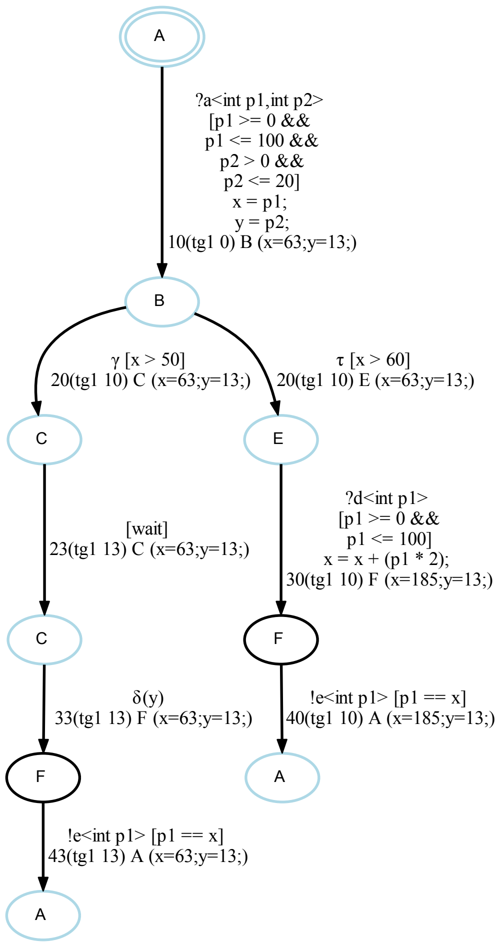


Figure 3.1.: Example of an execution tree.

Definition 3.7. (Blocked Execution Tree Node)
An execution tree node is blocked if at least one state of the set is blocked. □

Definition 3.8. (Stopped Execution Tree Node)
An execution tree node is stopped if all states of the set are stopped. □

Example 3.1. (Execution Tree)
The execution trees are displayed as graphs. The double framed node is the start node. Furthermore, the nodes are colored respective to their properties, where blocked nodes are blue, stopped nodes are red and otherwise the nodes are black. The execution tree graphs include additionally the clock valuation and attribute valuation information for every state change. This information is displayed in the following way:

- Transition symbol
- Message name or delay (optional)
- Simulation clock valuation
- Timing group names plus associated clock valuation (optional)
- States plus the attribute valuation

In Figure 3.1 an execution tree is shown that is the result of the simulation of the ESTS in Figure 2.1. Due to the attribute `x` gets the value `63` from the input transition `a`, the execution tree has two execution paths. These are `A, B, C, F, A` and `A, B, E, F, A`. The `[wait]` indicates an execution tree node which does only elapse time without executing a transition. The difference of the timing group clock value and the value of the attribute `y` has to elapse. In this case, 3 time units were elapsed. □

3.2. Single ESTS Simulation

The simulation of a single ESTS is the basis to simulate a set of communicating ESTSs. This simulation calculates an execution tree where the nodes have always one state. This is because only on state of an ESTS can be the current state in a node.

There are two different types of single ESTS simulators, the single step and the continuous simulator. The single step simulator executes all enabled transitions of a given node and terminates. The continuous simulator executes all enabled transitions of the given nodes and the generated leaf nodes until no further enabled transitions exist.

Definition 3.9. (Processable Message)
A message is processable if the send time is less or equal to the simulation clock valuation. In addition, at least one input transition has to be labeled with the message's label. □

Definition 3.10. (Enabled Transition)
A transition is enabled if its guard evaluates to true. A delay transition additionally has to fulfill that the timing group clock valuation is greater or equal to the delay the delay transition is labeled with. In case of an input transition a processable message has to be first in the message list. □

Definition 3.11. (Completion Step)
A completion step is part of an execution of an ESTS, where enabled completion transitions are executed before enabled input and delay transitions. □

That means in the case a completion transition is enabled, enabled input and delay transition won't be executed.

Definition 3.12. (Calculated Execution)
The calculated execution duration is the calculated time lapse of the transition, with respect to the execution duration of the transition and its lower and upper bound. □

Definition 3.13. (Instantiated Transition)
An instantiated transition consists of a transition, a possible empty message and a calculated execution duration. □

3.2.1. Single Step Simulator

This simulator makes one simulation step and stops afterward. A step is the execution of all enabled transitions of a state of a leaf node. In the case enabled transitions exist the node gets child nodes. These child nodes can be the inputs for further executions of this simulator. Due to the simulation stops after every node messages can be added to the list of messages before the simulation continues.

Algorithm 3.1 defines such a simulation step. In the case the node has more than one states only the state which belongs to the simulating ESTS is considered. The algorithm expects as parameter a node which has to be a leaf and a maximum simulation time. The enabled transitions of the given node will be executed, where the maximum simulation time must not be exceeded. In the case transitions are only enabled after a certain time, the simulation elapses time. The algorithm has three different return values. First, `EXECUTED` if transitions were executed. Second, `INSUFFICIENT_TIME` if the simulation clock value is greater than the maximum simulation time, after executing a transition. Third, `NO_ENABLED_TRANSITIONS` if no transition is enabled.

Determine Enabledness

The function `get_enabled_transitions` returns a set of instantiated transitions or an empty set in the case no transitions are enabled for given state. The enabledness of the transitions are determined in this function. For output, completion and unobservable transitions only the guard has to evaluate to true, to be enabled. In the case more than one transition of a subset of the set of labels are enabled, only the once with the highest priority will be returned. That means the priority is for ordering the transitions in the subset of their labels and not for the whole set of transitions. After evaluating the guard of an output

Algorithm 3.1 Single Simulation Step

```
Require: $node$ is a leaf node.
 1: function SINGLE_SIMULATION_STEP($node, max\_time$)
 2:     $state \leftarrow$ GET_RELATED_STATE($node.states$)
 3:     $enabled\_transitions \leftarrow$ GET_ENABLED_TRANSITIONS(
            $state, node.attribute\_valuation$)
 4:     if $enabled\_transitions = \emptyset$ then
 5:         $time\_elapsed \leftarrow$ TRY_ELAPSE_TIME($node, max\_time$)
 6:         if $time\_elapsed = TIME\_ELAPSED$ then
 7:             $enabled\_transitions \leftarrow$ GET_ENABLED_TRANSITIONS(
                    $state, node.attribute\_valuation$)
 8:         else if $time\_elapsed = INSUFFICIENT\_TIME$ then
 9:             return $INSUFFICIENT\_TIME$
10:         else
11:             return $NO\_ENABLED\_TRANSITIONS$
12:         end if
13:     end if
14:     $max\_finalization\_time \leftarrow$ GET_MAX_EXECUTION_DURATION(
            $enabled\_transitions) + node.simulation\_clock$
15:     if $max\_finalization\_time <= max\_time$ then
16:         for all $t \in enabled\_transitions$ do
17:             EXECUTE_TRANSITION($node, t$)
18:         end for
19:         return $EXECUTED$
20:     else
21:         return $INSUFFICIENT\_TIME$
22:     end if
23: end function
```

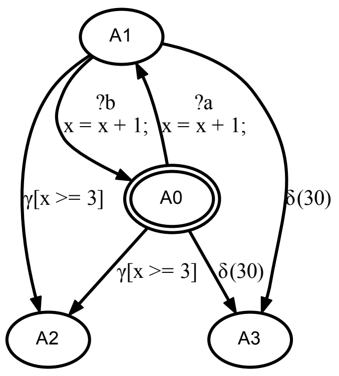
transition, a message with the parameter valuation will be created and added to the corresponding instantiated transition.

Input and delay transitions are not considered in the case a completion transitions is enabled. This is because of the Definition 3.11, where completion transitions have to execute before input and delay transitions. If no completion transition is enabled the enabledness of input and delay transitions also is determined. Input and delay transitions have to in addition to a guard evaluating to true the following conditions to fulfill: An input transition needs a processable message (Definition 3.9), where always the first message which has the corresponding ESTS as target in the list of messages will be checked. The send time has to be less or equal to the simulation clock valuation and the label has to be the same as the transition label. If the message fulfills the conditions it will be added to the instantiated transition. In the case of a delay transition, the clock valuation of the corresponding timing group has to be greater or equal to the delay. Also, for input and delay transitions only the one with the highest priority will be returned.

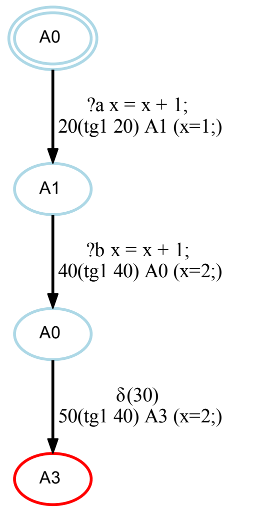
For an enabled transition an instantiated transition is created, where the execution duration will be calculated. There are different possibilities how to calculate the execution duration.

- The execution duration is used.
- The lowest execution duration is used.
- The highest execution duration is used.
- The average execution duration is used.
- A random execution duration will be calculated.
- A constant execution duration can be used for all transitions. This can be also zero to simulate without considering the time behavior.

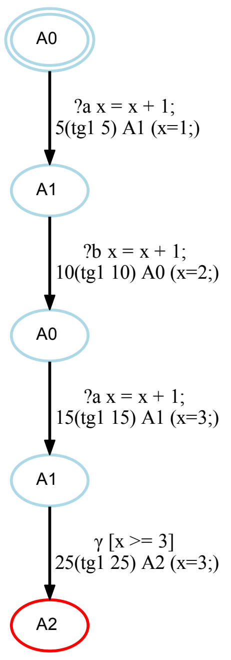
Example 3.2. (Different execution duration calculations)
On the basis of the ESTS, in Figure 3.2a, the consequences of the use of different execution duration calculations is shown. The ESTS receives messages a and b in a loop. The loop ends after the attribute x is greater or equal to 3. Another exit criteria are the delay transitions which ends the loop after 30 time units. However, if the input and the delay transition are enabled at the same time the execution tree will be non-deterministic. In Figure 3.2d a message was processable when the delay expired and due to the arising non-determinism two execution paths exist. The three execution trees in Figure 3.2 are the results of three different simulations with different execution duration calculations. The results are different because of the execution duration and the defined bounds of the input transitions a and b. They have a lower and an upper bound for the execution duration and so they can take from 5 to 20 time units for execution. In Figure 3.2b the simulation took always maximum execution duration and so the delay transition was executed after processing two messages. The minimum execution duration was used to generate the execution tree in Figure 3.2c. Due to the fast execution of only 5 time units three messages can be processed and the completion transition ends the loop. The last execution duration calculation in this example is a random one which takes a random execution duration when a transition has an execution duration scope. The result of the use of this calculation is shown in 3.2d, where the completion and the delay transitions are executed as shown by the execution tree. □



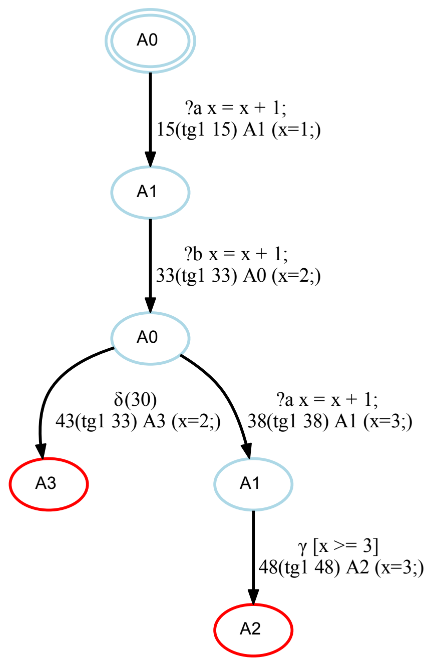
(a)ESTS



(b)Maximum execution time



(c)Minimum execution time



(d)Random execution time

Figure 3.2.: ESTS and execution trees with different execution duration calculations.

Elapse Time

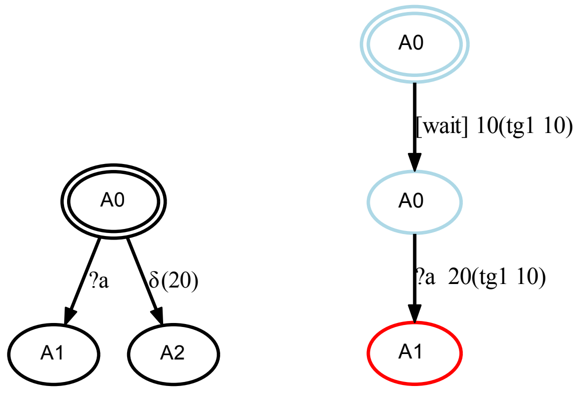
If no transition of the state is enabled yet, the input and delay transitions will be checked for enabledness in the future with the function `try_elapse_time`. Input transitions can be only enabled in the future when the send time of the first message which has the corresponding ESTS as target in the list of messages is greater than the simulation clock valuation. Nevertheless the guard of the transition has to evaluate to true. The difference of the clock valuation and the simulation clock valuation can be elapsed, to ensure an enabled input transition. Also, for delay transitions the difference of the timing group clock valuation and the delay can be elapsed.

In the case more than one transition is enabled in the future the shortest time that is needed to enable a transition will be elapsed. Time elapsing is done by increasing the simulation clock valuation. In addition, the clock valuation of the timing groups which the state belongs to have to be increased by the same amount. After increasing, the simulation clock valuation has to be less or equal to the maximum simulation time. Is this not the case the algorithm returns `insufficient_time` and no clock valuation will be changed. The other return values can be `time_elapsed` and `no_enabled_transitions` if no transition will be enabled in the future. In the case clock valuations were changed the function `get_enabled_transitions` will be applied a second time, to get the newly enabled transitions.

Example 3.3. (Elapse Time)
This example demonstrates the behavior of the simulator when two transitions are enabled in the future. In Figure 3.3a an ESTS with one input and one delay transition with the same source state is shown. At the start of the simulation a message for the input transition is in the list of messages where the send time has the value 10. In this case, the input and the delay transition will be enabled in the future. The input transition after 10 time units and the delay transition after 20 time units, which is the value of the delay of the transition. The shortest time to get an enabled transition will be elapsed and therefore 10 time units are elapsed and the input transition can be executed. The execution tree is displayed in Figure 3.3b. □

Execute Transitions

Before executing the enabled transitions, the finalization times have to be verified. The maximum finalization time of the enabled transitions has to be less or equal to the maximum simulation time. The finalization time is the sum of the calculated execution duration and the simulation clock valuation. Only in



(a)ESTS (b)Execution tree

Figure 3.3.: Elapse time example.

the case that the execution of all enabled transitions will be finished within the maximum simulation time they will be executed, otherwise `insufficient_time` is returned.

Algorithm 3.2 Execute a transition

```
 1: procedure EXECUTE_TRANSITION($node, instantiated\_transition$)
 2:     $transition \leftarrow instantiated\_transition.transition$
 3:     $message \leftarrow instantiated\_transition.message$
 4:     $child\_node \leftarrow node$.CREATE_CHILD($transition$)
 5:     $child\_node$.ELAPSE_TIME($instantiated\_transition.execution\_duration$)
 6:     if $message = \emptyset$ then
 7:         $child\_node$.EXECUTE_UPDATE_FUNCTIONS()
 8:     else
 9:         $child\_node$.EXECUTE_UPDATE_FUNCTIONS(
                $message.parameter\_valuation$)
10:     end if
11:     if $transition.label \in$ {output transitions} then
12:         $message$.SET_SEND_TIME($child\_node.simulation\_clock$)
13:         $child\_node.message\_list$.ADD($message$)
14:     else if $transition.label \in$ {input transitions} then
15:         $child\_node.message\_list$.REMOVE($message$)
16:     end if
17:     $node$.ADD_CHILD($child\_node$)
18: end procedure
```

The execution is handled in the function `execute_transition` which is defined in Algorithm 3.2. This algorithm is based on a similar algorithm in [8]. An execution of a transition leads to a new node in the execution tree. Therefore, a new node is created that is based on the given node. The message list, the attribute valuation and the clock valuation where copied to the new node. The target state of the transition is used as state of the node and the transition is also held as executed transition in the node. The list of child nodes is empty because the new node is a leaf node.

The time the transition need to execute has to be elapsed. Therefore, the simulation clock valuation and the clock valuations of the timing groups, to which the new node belongs, were increased by the calculated execution duration. In the case the transition is a clock reset transition the clock valuations of the corresponding timing groups are set to zero. Afterward, the update functions of the executed transition are executed. If a message exists, the parameter valuation of the message can be used to execute the update functions.

In the case the executed transition is an output transition the message is added to the list of messages. Before adding to list, the send time of the message is updated to the simulation clock valuation. The message can be received from an input transition at the moment the output transition has been executed. Is the executed transition an input transition than the message is removed from the list of messages. The message was received from the ESTS which leads to input transition execution and therefore the message is removed from list.

The execution of the transition ends by adding the new node to the list of child nodes of the given node. Is the child node the first one in list than the execution path grows, otherwise a new execution path is created.

3.2.2. Continuous Simulator

In contrast to, the single step simulator this simulator does not interrupt after every execution of a node. The simulation runs as long enabled transitions exist. If no enabled transitions in the leaf nodes exist the simulation interrupts and messages can be added to the list of messages. While simulating no messages can be added to the list of messages from the environment or from other ESTSs. Only messages from output transitions which are executed are added to the list of messages.

The Algorithm 3.3 defines the continuous simulation. As parameter the algorithm expects a leaf node and a maximum simulation time. Due to the single

Algorithm 3.3 Continuous Simulation

```
Require: $node$ is a leaf node.
 1: function SIMULATE_CONTINUOUSLY($node, max\_time$)
 2:     $sim\_state \leftarrow$ SINGLE_SIMULATION_STEP($node, max\_time$)
 3:     if $sim\_state = EXECUTED$ then
 4:         for all $leaf \in node$.GET_CHILD_NODES( )do
 5:             $child\_sim\_state \leftarrow$ SIMULATE_CONTINUOUSLY($leaf, max\_time$)
 6:             if $child\_sim\_state = INSUFFICIENT\_TIME$ then
 7:                 $sim\_state \leftarrow child\_sim\_state$
 8:             else if $sim\_state \neq INSUFFICIENT\_TIME$ then
 9:                 $sim\_state \leftarrow child\_sim\_state$
10:             end if
11:         end for
12:     end if
13:     return $sim\_state$
14: end function
```

ESTS simulator only the state which belongs to the simulating ESTS is considered of the given node. The algorithm calls itself recursively as long transitions can be executed. Therefore, the algorithm only has two possible return values `INSUFFICIENT_TIME` and `NO_ENABLED_TRANSITIONS`.

First of all the function `single_simulation_step` of the single step simulator is applied. This function executes all enabled transitions of the given node, if there are someone. In the case the function has executed transitions the algorithm continues. Otherwise, the return value of the function which provides information why no transitions was executed will be returned.

The execution of the enabled transitions leads to new nodes in the list of child nodes of the given node. These child nodes are the new parameter for the recursive execution of this algorithm. The return values of the recursive executions are weighted, where `INSUFFICIENT_TIME` has a greater weight than `NO_ENABLED_TRANSITIONS`. Therefore `INSUFFICIENT_TIME` is used if at least one execution returns this value.

3.3. Multiple ESTS Simulation

The simulation of two or more ESTSs at the same time is described in this section. The single step simulators are used to simulate each ESTS and the resulting execution trees will be merged together.

The ESTSs are able to communicate with each other while simulation and therefore there are three different ways how to do this. First, the communication is deterministic, where the ESTSs will be simulated in a specific order and so only the current simulated ESTS is able to send messages. The messages will be received at the turn of the target ESTSs. Second, full interleaving communication, where all ESTSs are simulated simultaneously and so they are able to send and receive at any time. Third the ESTSs are simulated full parallel, where all ESTSs are again simulated simultaneously, but in this case the transitions can be executed parallel. Therefore, all ESTSs can send messages, but receiving is only possible if no transition is executing at the moment.

Definition 3.14. (External Input Transition)
An input transition which has no corresponding output transition in the simulating ESTSs is named external input transition. Corresponding in this context means no output transition exists, which can generate a message for the input transition. Therefore, such a message has to come from the environment to enable the transition. □

3.3.1. Deterministic Simulator

The deterministic simulator simulates the ESTSs consecutively in a predefined order. Therefore, the simulation leads to a non-deterministic execution tree only in the case an ESTS is non-deterministic. Otherwise, the execution tree is deterministic and holds exact one execution path.

The Algorithm 3.4 defines the deterministic simulation. There are three parameters, where `sim_list` is a list of single ESTS simulators, `execution_tree` is the execution tree which will be extended, and `max_time` is the maximum time the simulation time is allowed to reach. Every simulator will be executed once, where this algorithm is executed in rotation while `EXECUTED` is returned. At the first run the execution tree has to consists of a root node only which holds all start states of the ESTSs of the simulators. In further runs the resulting execution tree is used as parameter and therefore it will grow as long as transitions are executed. In the case no transitions can be executed `NO_ENABLED_TRANSITIONS` is returned and if the simulation time will exceed the `max_time` `INSUFFICIENT_TIME` is returned. If one of these two return values are returned the given execution tree will not be changed.

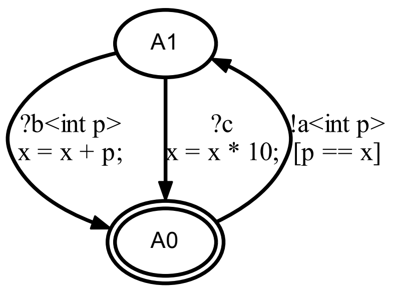
There are two nested loops in the algorithm, one for the simulators of the list of simulators and one for the leaf nodes of the execution tree. The simulators are executed for every leaf node of the execution tree in the order they were added to the list. In the case transitions are executed child nodes are added to the leaf node. This extends the execution tree and therefore it has one or more new leaf nodes. Due to this fact the simulator that is next in the list can have different leaf nodes than his predecessor. That is why the order of the simulators in the list is important because different orders can lead to different execution paths.

In the case a simulator returns `INSUFFICIENT_TIME`, by reaching the given maximum simulation time, the simulation stops and resets the execution tree. The reset is done by removing the child nodes of the nodes which were the leaves at begin of the first loop. This is necessary because the execution order of the simulators can be disordered on further executions of the algorithm with a higher maximum time.

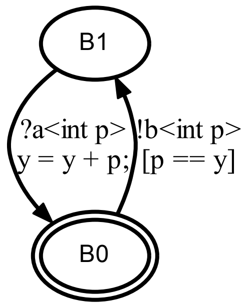
Algorithm 3.4 Deterministic Simulation

```
1:  function DETERMINISTIC_SIMULATION($sim\_list, execution\_tree, max\_time$)
2:      $executed \leftarrow false$
3:      $origin\_leaf\_nodes \leftarrow execution\_tree$.GET_LEAF_NODES( )
4:      for all $sim \in sim\_list$ do
5:          $leaf\_nodes \leftarrow execution\_tree$.GET_LEAF_NODES( )
6:          for all $leaf \in leaf\_nodes$ do
7:              $sim\_state \leftarrow sim$.SIMULATE($leaf, max\_time$)
8:              if $sim\_state = INSUFFICIENT\_TIME$ then
9:                  for all $origin\_leaf \in origin\_leaf\_nodes$ do
10:                     $origin\_leaf$.REMOVE_CHILD_NODES( )
11:                 end for
12:                 return $INSUFFICIENT\_TIME$
13:             end if
14:             if $sim\_state = EXECUTED$ then
15:                 $executed \leftarrow true$
16:             end if
17:         end for
18:     end for
19:     if $executed$ then
20:         return $EXECUTED$
21:     else
22:         return $NO\_ENABLED\_TRANSITIONS$
23:     end if
24: end function
```

Example 3.4. (Deterministic simulation)



(a)ESTS 1



(b)ESTS 2

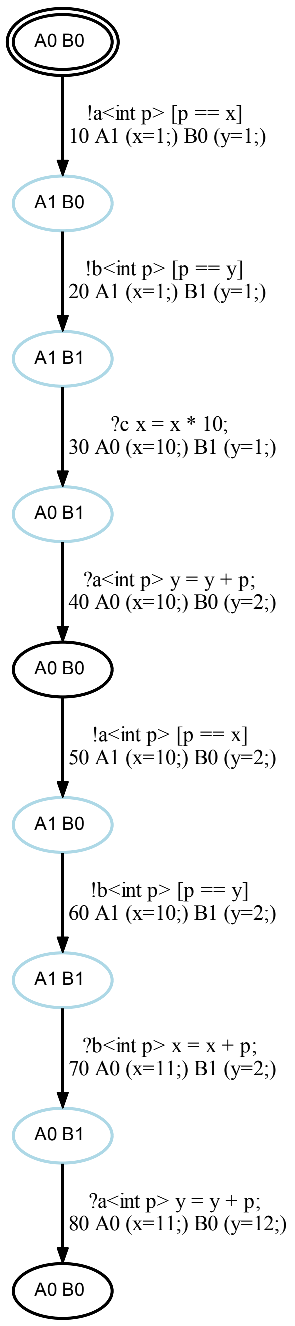
Figure 3.4.: ESTSs for deterministic simulation.

The functionality of the deterministic simulator is shown in this example, where two ESTSs are simulated with different single ESTS simulators. The first time the single ESTSs are simulate with a single step simulator and the second time they are simulated with a continuous simulator.

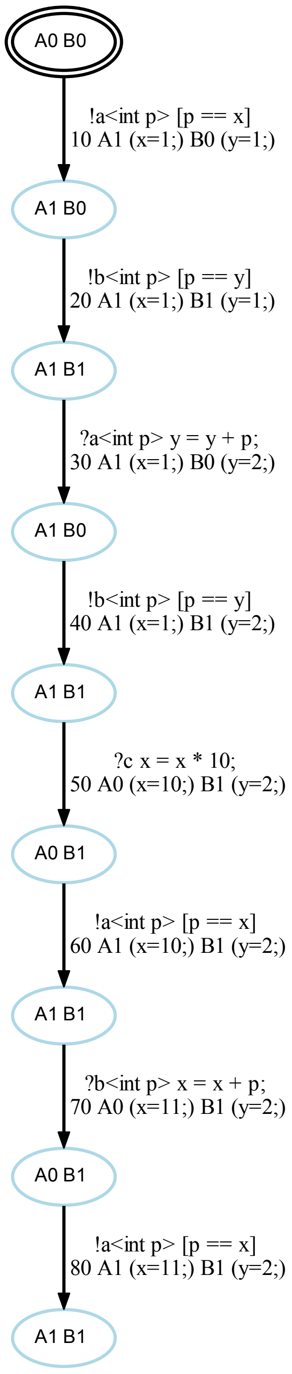
The ESTSs which were simulated are shown in Figure 3.4. ESTS 1 has two states `A0` and `A1`, one output transition `a` and two input transitions `b` and `c`. ESTS 2 also has two states `B0` and `B1`, one output transition `b` and one input transitions `a`. For the output transitions `a` and `b` there are corresponding input transitions which can receive the messages. The input transitions are always part of the other model than the output transitions belong to. Therefore, the models are able to communicate with each other. In addition, the input transition `c` is external which means the message has to come from the environment.

In Figure 3.5 two execution trees of two different deterministic simulation runs are shown which are cut off at simulation time 80. The single ESTS simulators are, in both simulations, added in the same order, first ESTS 1 and then ESTS 2. In addition, in both runs the environment sends a message for the external input transition `c` with send time 10.

The result of a deterministic simulation with two single step simulators is shown in Figure 3.5a. The start states of the two ESTSs are `A0` and `B0` which are displayed in the root node of the execution tree. The first simulator in the list was executed first. Therefore, the enabled output transition of state `A0` was executed. At this time, the environment sent a message for input transition `c`. The second simulator was executed next, where the new generated node is the base of simulation. Afterward, the algorithm terminated and returned `EXECUTED`. The current states at this moment were `A1` and `B1`. The algorithm was repeated and so the input transition `c` was executed. This is done because this message was first in the list. Then, the input transition `a` was executed. The



(a)Two single step simulators.



(b)Two continuous simulators.

Figure 3.5.: Deterministic simulation.

algorithm was repeated a few times and the simulators of ESTS 1 and ESTS 2 were executed alternately and executed in every turn one transition.

The second execution tree in Figure 3.5b was created by a deterministic simulation with two continuous simulators. The simulation starts like the first simulation, the output transitions `a` and `b` were executed. After the output transition `b` the continuous simulator of ESTS 2 was not terminated because further enabled transitions were available. Therefore, the input transition `a` and after this again the output transition `b` were executed. Afterward, the second simulator terminated and therefore the algorithm also terminated and returned `EXECUTED`. The algorithm was executed again and the simulator of ESTS 1 was also executed first. The messages which were sent from the environment and from the simulator of ESTS 2 enabled the input transitions at state `A1`. As already mentioned the execution tree was cut off before the second run of the algorithm terminates.

The usage of different single ESTS simulators leads to different execution trees which can be seen at the execution trees in Figure 3.5. The order of the execution of the transitions are different. This leads to nodes which have different states and different attribute valuations at the same simulation time. □

3.3.2. Full Interleaving Simulator

This simulator is also a multiple ESTS simulator, where all enabled transitions of all states of a given node are executed at the same time. In the case more than one transitions are enabled a non-deterministic execution tree will be the result of simulation. The order of the added simulators has no importance because all simulators are executed for a node and therefore the order has no impact.

In Algorithm 3.5 the full interleaving simulation is defined. This algorithm is in many respects similar to Algorithm 3.4, but the differences are essential. The first loop iterates through the leaf nodes of the given execution tree. For all leaf nodes in the second loop all simulators in its list are executed. Due to this fact all enabled transitions are executed and the newly created nodes are added to the current leaf node.

The algorithm returns `EXECUTED` if at least one simulator has executed a transition. In the case no transitions are enabled, `NO_ENABLED_TRANSITIONS` is returned and the execution tree will not be changed. If one of the simulators exceeded the maximum simulation time `INSUFFICIENT_TIME` is returned. The execution tree will be reset because not all simulators have executed the enabled transitions.

Algorithm 3.5 Full Interleaving Simulation

```
function FULL_INTERLEAVING_SIMULATION($sim\_list, execution\_tree, max\_time$)
    $executed \leftarrow false$
    $leaf\_nodes \leftarrow execution\_tree$.GET_LEAF_NODES( )
    for all $leaf \in leaf\_nodes$ do
        for all $sim \in sim\_list$ do
            $sim\_state \leftarrow sim$.SIMULATE($leaf, max\_time$)
            if $sim\_state = INSUFFICIENT\_TIME$ then
                for all $origin\_leaf \in leaf\_nodes$ do
                    $origin\_leaf$.REMOVE_CHILD_NODES( )
                end for
                return $INSUFFICIENT\_TIME$
            end if
            if $sim\_state = EXECUTED$ then
                $executed \leftarrow true$
            end if
        end for
    end for
    if $executed$ then
        return $EXECUTED$
    else
        return $NO\_ENABLED\_TRANSITIONS$
    end if
end function
```

Example 3.5. (Full interleaving simulation)
The functionality of the described algorithm will be shown in this example. Like in Example 3.4 the ESTSs of Figure 3.4 will be used for simulating. The full interleaving simulator will be applied twice with different single ESTS simulators.

The basic conditions are similar to Example 3.4. The order of the single ESTS simulator are in both runs the same and at simulation time 10 a message for the external input transition `c` will be sent.

In Figure 3.6 the resulting execution tree of a full interleaving simulation with two single step simulators is shown. The execution tree is non-deterministic and holds 6 execution paths. At the begin both states `A0` and `B0` have an enabled output transition which were executed. After executing output transition `a` the following two transitions become enabled. The input transition `c`, because of the

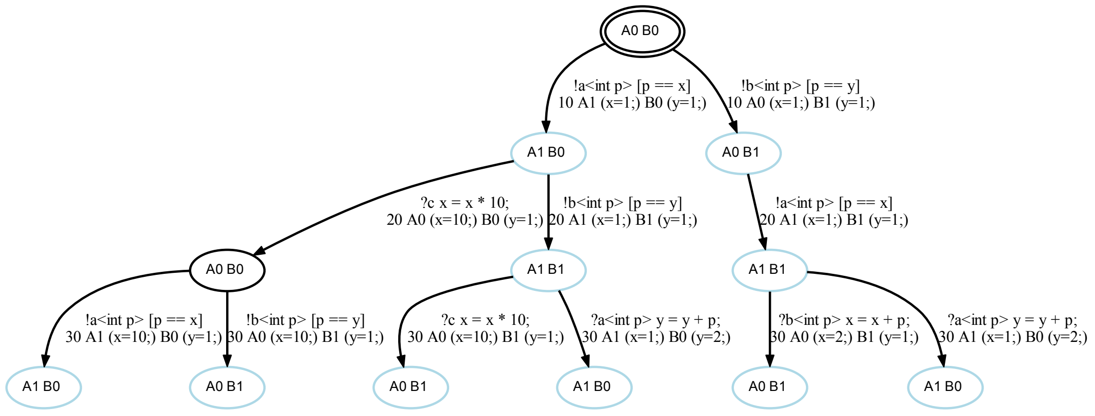


Figure 3.6.: Full interleaving simulation of two single step simulators.

message from the environment, and output transition b. All enabled transitions of all states of all leaf nodes will be executed as long the simulation will be repeated. Therefore, all possible execution paths of the two ESTSs with the one message for input transition c are the result of the simulation.

The second full interleaving simulation was performed with two continuous simulators. The created execution tree is shown in Figure 3.7 which is cut off at simulation time 80. At the beginning the two output transitions a and b are enabled and so they are executed. In both cases the continuous simulator terminated because there were no messages to receive. During the second run of the algorithm the leaf nodes process messages contained in the list of messages. These messages were sent from the environment and from the executed output transitions. Therefore, the continuous simulations have to execute a number of transitions before the termination.

The execution path in the middle of the resulting execution tree is the same as in Figure 3.5b. Further the execution path on the right side will be the result of the deterministic simulation with a reverse order of the single ESTS simulators. The left one will never be an execution path of the deterministic simulation because the single ESTS simulator of ESTS 1 was executed twice consecutively.

As mentioned before the execution tree in figure 3.6 consists of all possible execution paths. Therefore, all execution paths of the Figures 3.5a, 3.5b and 3.7 are part of the execution tree. In addition, there are three execution paths more which only can be created with a full interleaving simulator with single step simulators for every simulating ESTS. □

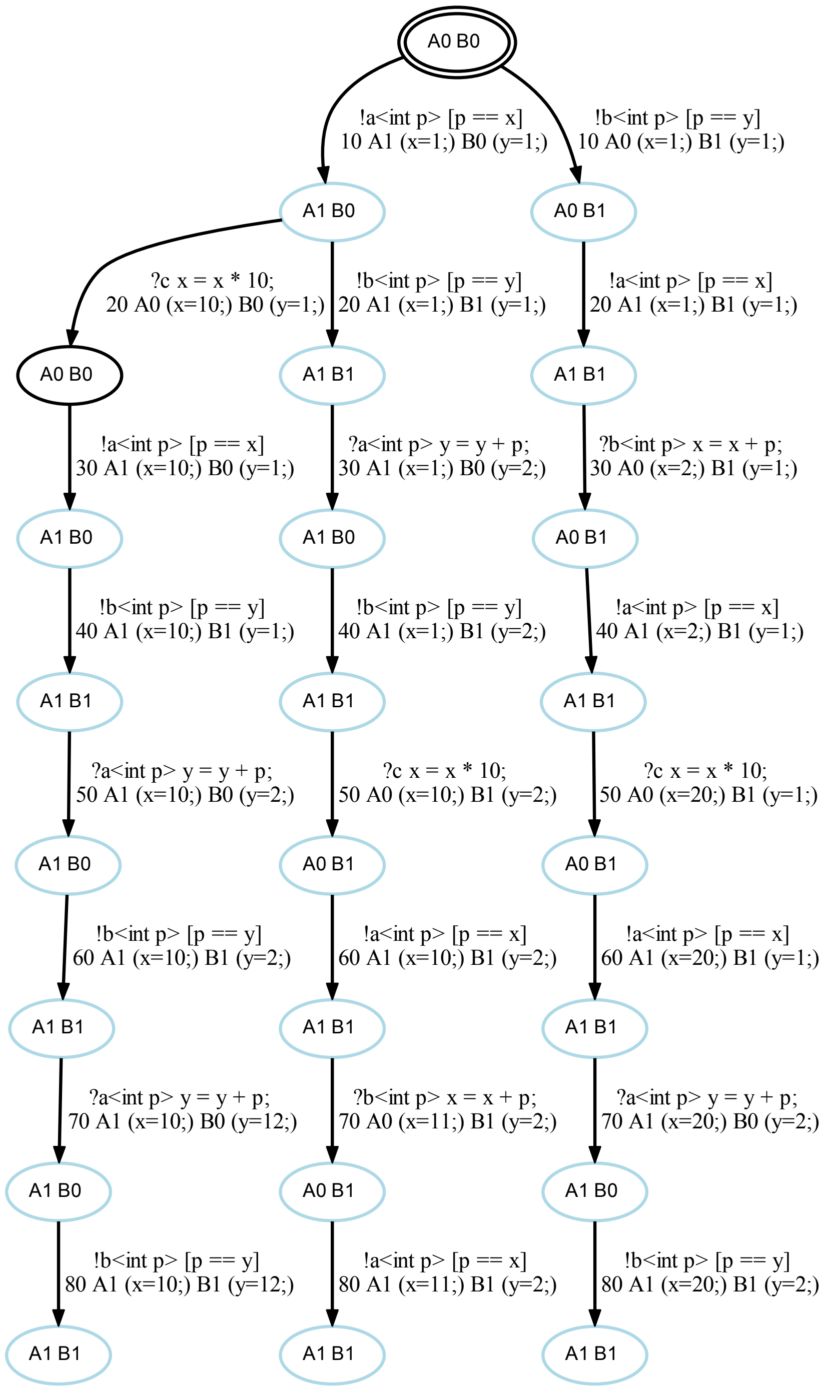


Figure 3.7.: Full interleaving simulation of two continuous simulators.

3.3.3. Full Parallel Simulator

Full parallel simulation simulates the parallel execution of ESTSs. That means the models will be simulated as if every model has its own processor. Due to this fact more than one transition can be executed at the same time. Therefore, deterministic ESTSs leads to a deterministic execution tree.

Definition 3.15. (Extended Execution Tree Node)
The execution tree node has to be extended by replacing the executed transition to a set of executed transitions and by adding a set of running transitions, to be able to simulate parallel executions of transitions. □

The set of running transitions is a set of instantiated transitions which were executed, but the execution is not finished at the moment of the node's simulation clock valuation.

The Algorithm 3.1 which is the base of every simulator has to change to be able to deal with the set of running transitions. In the case this set is not empty and the highest calculated execution duration is not greater than the maximum simulation time, the transitions in the set will be executed. The enabledness of the transitions of the state will not be checked if transitions are in the set of running transitions.

Depending on the execution duration of the transitions, there are two possibilities the generated execution tree can look like. These possibilities are true for enabled transitions of different ESTSs.

1. Execution duration of the enabled transitions are equal:
   This behavior leads to a deterministic execution tree, where the list of executed transitions holds more than one transition. Every enabled transition will be executed and all executions will lead to the same node.
2. Execution duration of the enabled transitions are different:
   The enabled transition with the lowest execution duration will be executed first and the other enabled transitions are added to the set of running transitions. After that, the running transition with the lowest execution duration will be executed. This will be continued until the set of running transition is empty. Due to the execution of the transitions was started at the same time, the simulation clock valuation will be not always increased by the whole execution duration of the transitions.

This simulator uses in general the same algorithm as the full interleaving simulator. The only change in Algorithm 3.5 is the merging of the created child nodes after every execution of a simulator. In the case two or more transitions of different ESTSs are executed the resulting nodes have to be merged together.

In Algorithm 3.6 the merging of two nodes is described. The parameters are the set of child nodes of the current node and the ESTS of the simulator that was executed last. The model is used to know which child nodes were added to the set and have to merge. As a result this algorithm returns a set of nodes which hold the merged child nodes of the given set. In the case no node was merged the given set of child nodes will be returned.

There are two nested loops, where the first is for the child nodes which exist in the set before the last simulator was executed. In the algorithm these are the `old child nodes`. The second one handles the `new child nodes` which are created from the last executed simulator and does the merging. That means every old child node is merged with every new child node. The execution duration of the executed transitions and the corresponding simulation clock valuation of the child nodes are the key factors how to merge. There are three different possibilities which have to be considered:

1. Simulation clock valuations are equal:
   The two child nodes really melt together to one node. This means the executed transition of the new created child node is added to the set of executed transitions of the old child node. Also, the set of states, the list of messages, the attribute valuation and the clock valuation are merged. Therefore, the values which were changed by executing the transition of the new child node are adapted.
2. Simulation clock valuation of the new child node is lower:
   The two child nodes cannot be merged because the execution durations are not equal. The new child node is the one with the lowest simulation clock valuation and therefore this node will be used as base child node for merging. The executed transitions of the old child node are converted into a set of instantiated transitions which are added to the set of running transitions of the new child node. During the next execution of the simulators the running transitions will be executed again. Also, the enabled transitions of the state that is the target of the executed transition of the new child node will be executed.
   If the new child node will be inserted in execution path before the old child node, it will not be longer a leaf node. Therefore the simulator can not execute the enabled transitions on the correct time. In this case, the resulting execution tree will not be correct.
3. Simulation clock valuation of the new child node is greater:
   The old child node has the lower simulation clock valuation and therefore

the old child node is the base for merging. The executed transition of the new child node will be converted into an instantiated transition and will be added to the running transitions set of the old child node.

Simulating deterministic ESTSs where only one old and one new child node exists, are merged into one child node and therefore the execution tree stays deterministic. In a non-deterministic environment all old child nodes are merged with all new child nodes and therefore the corresponding number of child nodes is the product of the number of old and new child nodes.

The Algorithm 3.6 does not handle nodes which does elapse time without an executed transition. In the case a child node was created to elapse time and the simulation clock valuation is greater or equal to the merging child node, the node for time elapsing will be ignored. Otherwise, the executed transitions will be added to the set of running transitions contained by the time elapsing node.

Example 3.6. (Merge different execution durations)
In this example the full parallel simulation will be illustrated. We simulate the ESTSs which are displayed in Figures 3.8a and 3.8b twice, but with different execution durations.

The first time the execution duration of both output transitions is 10. The resulting execution tree is shown in Figure 3.8c. These transitions were executed and because of the equality of the execution duration both were added to the set of executed transitions. Therefore, only one new node was added to execution tree.

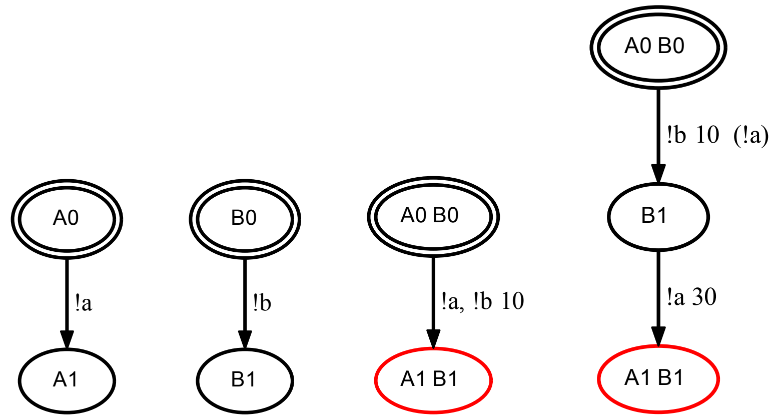
For the second simulation the execution duration of the output transition a was increased to 30. This leads to a different execution tree that is shown in Figure 3.8d. Both output transition started execution at same time. The output b is ready in 10 time units and therefore a node was created, where the output a was added to the running transition set, which is the reason the second node has only one state. After 30 time units a is also ready with execution and therefore a second node was created. □

Example 3.7. (Full parallel simulation)
The ESTSs of Figure 3.4 are also simulated with the full parallel simulator. The resulting execution trees are shown in Figure 3.9. Due to the simulating ESTSs are deterministic the generated execution trees are also deterministic.

The order of the single ESTS simulators is the same as in the Examples 3.4 and 3.5. Furthermore, at simulation time 10 a message was sent for the external input transition c from the environment.

Algorithm 3.6 Merge child nodes

```
1: function MERGE_CHILD_NODES($child\_nodes, ests$)
2:     $new\_child\_nodes \leftarrow \emptyset$
3:     for all $old\_child \in child\_nodes$ do
4:         $first\_transition \leftarrow old\_child.executed\_transitions$.GET_FIRST( )
5:         if $\begin{pmatrix} (old\_child.executed\_transitions.\text{GET\_SIZE}(\,) = 1) \wedge \\ (ests.\text{CONTAINS\_TRANSITION}(first\_transition)) \end{pmatrix}$ then
6:             continue
7:         end if
8:         for all $new\_child \in child\_nodes$ do
9:             $transition \leftarrow new\_child.executed\_transitions$.GET_FIRST( )
10:            if $\begin{pmatrix} (new\_child.executed\_transitions.\text{GET\_SIZE}(\,) > 1) \vee \\ \neg(ests.\text{CONTAINS\_TRANSITION}(transition)) \end{pmatrix}$ then
11:                continue
12:            end if
13:            if $old\_child.sim\_clock = new\_child.sim\_clock$ then
14:                $new\_node \leftarrow old\_child$
15:                $new\_node.executed\_transitions$.ADD($transition$)
16:                $new\_node$.MERGE_NODE_FIELDS($new\_child.attributes, ests$)
17:            else if $old\_child.sim\_clock > new\_child.sim\_clock$ then
18:                $new\_node \leftarrow new\_child$
19:                $running\_trans\_list \leftarrow$ CREATE_RUNNING_TRANSITIONS(
                       $old\_child.execution\_transitions$)
20:                $new\_node$.ADD_RUNNING_TRANSITIONS($running\_trans\_list$)
21:            else
22:                $new\_node \leftarrow old\_child$
23:                $running\_trans \leftarrow$ CREATE_RUNNING_TRANSITION($transition$)
24:                $new\_node$.ADD_RUNNING_TRANSITION($running\_trans$)
25:            end if
26:            $new\_child\_nodes$.ADD($new\_node$)
27:        end for
28:    end for
29:    if $new\_child\_nodes = \emptyset$ then
30:        return $child\_nodes$
31:    else
32:        return $new\_child\_nodes$
33:    end if
34: end function
```



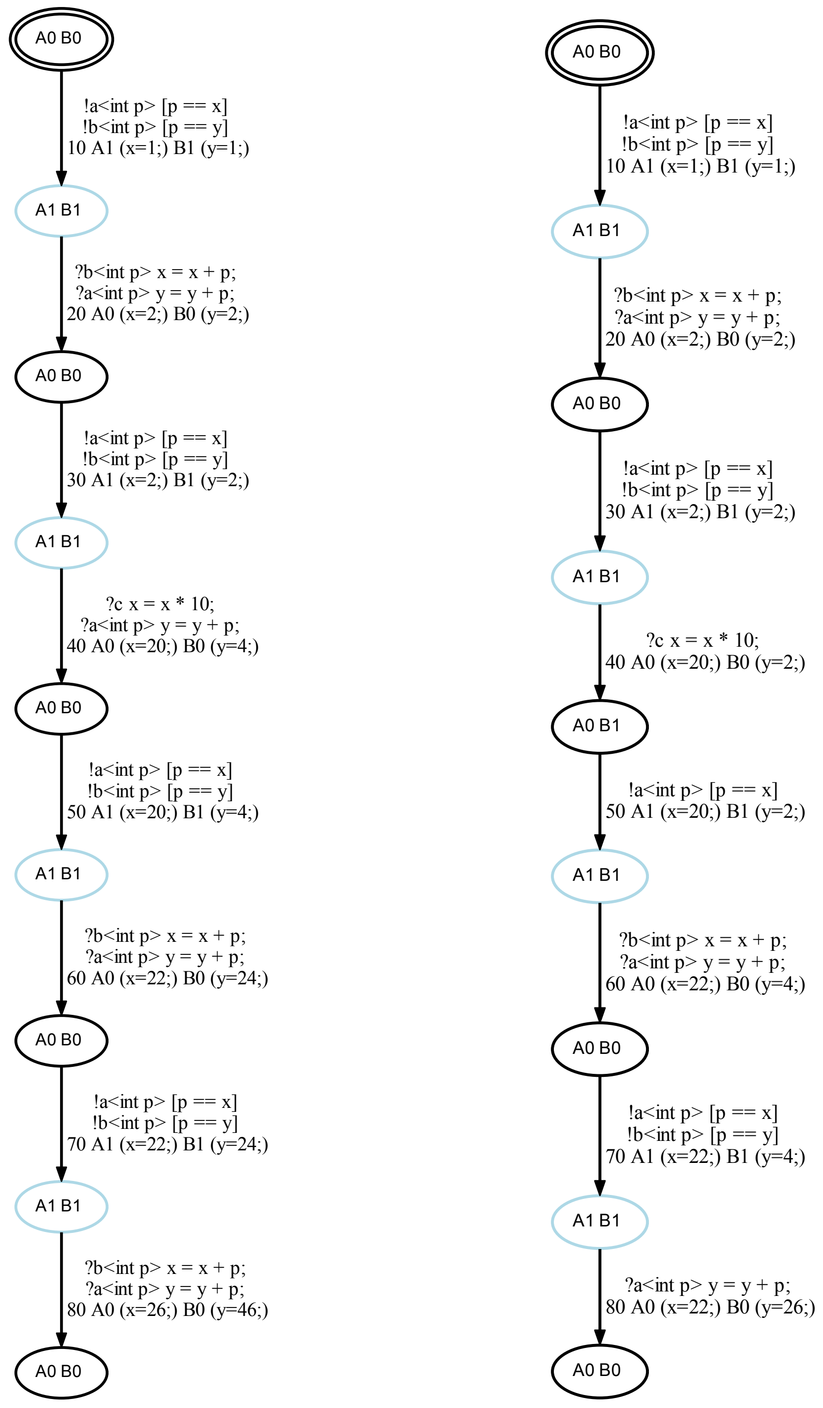
(a)ESTS 1 (b)ESTS 2 (c)Exec. tree 1 (d)Exec. tree 2

Figure 3.8.: Example full parallel simulation

The first full parallel simulation were made with two single step simulators, the result is shown in Figure 3.9a. The execution duration of every transition of both ESTSs are equal and both states of the nodes have always enabled transitions. This leads to an execution tree where every set of executed transitions held two entries. Therefore, the number of executed transitions has been doubled in the same time as in the other examples which uses the same ESTSs as before.

In Figure 3.9b the resulting execution tree is shown, where two continuous simulators were simulated. At the begin every continuous simulator only executed one transition which were merged afterward. At the second execution of the simulators more transitions were enabled. Therefore, the first simulator executed `?b`, `!a`, `?c` and `!a` and the second simulator executed `?a` and `!b`. The first two executed transitions are merged and for the last two executed transitions of the first simulator there are no other executed transitions to merge. Due to the continuous simulators are used the second simulator cannot be executed after termination. It has to wait until the first simulator terminated before both are executed again.

Due to the parallel execution of transitions the resulting execution trees are completely different than the once where a deterministic or full interleaving simulator were used. The resulting execution trees of the full parallel simulator have executed much more transitions in the same simulation time. Therefore, no execution path of the runs before matches the parallel once. □



(a)Two single step simulators.

(b)Two continuous simulators.

Figure 3.9.: Execution trees of two full parallel simulations.

4. Nested Simulation

The multiple ESTS simulators are for simulating systems, including more than one ESTS, with one specific communication mode. There are also systems which uses different communication modes and therefore the multiple ESTS simulators need to be nestable.

4.1. Nesting

The multiple ESTS simulators can be nested which means that one simulator contains other simulators. For this reason, the simulators are hierarchically organized in a tree like structure. The hierarchy is top down organized as shown in Figure 4.1. Such a simulator represents a subsystem of a system that includes two or more simulators. The nested simulators can be single or multiple ESTS simulators. In the case of a multiple ESTS simulator, the subsystem has a child. The nesting ends with a leaf subsystem that has only single ESTS simulators.

The algorithms of the single and multiple ESTS simulators have to be adapted to be able to handle both types of simulators nested in a multiple ESTS simulator. We use the signature of the single ESTS simulators, where a leaf node and a maximum simulation time are the parameters. The changes in the multiple ESTS simulators are quite simple. An execution tree has to be created with the given leaf node as root node and the list of simulators have to be a global list. Therefore, the simulators are added to the list before starting simulation.

Example 4.1. (Nested System)
In Figure 4.1 a system with nested subsystems is shown. The subsystems 2 and 3 are connected through subsystem 1 that do not have any other ESTSs. All subsystems have different connection modes and the four ESTSs will be simulated in a single step simulator.

The simulating ESTSs are displayed in Figure 4.2. The models of subsystem 2 send the values of their attributes `a` and `b` to subsystem 3. In ESTS 3 the

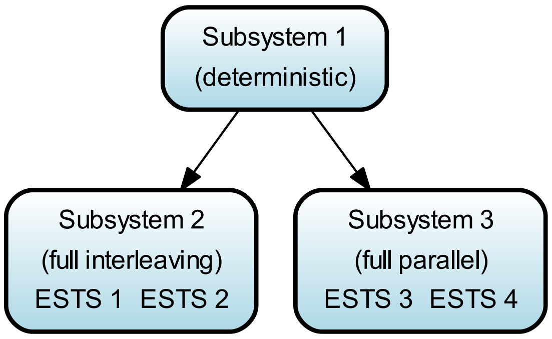
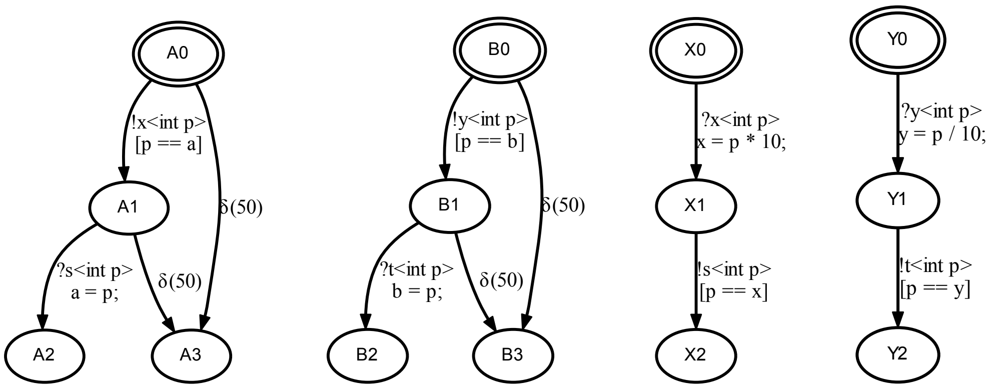


Figure 4.1.: Example of nested simulators.



(a)ESTS 1 (b)ESTS 2 (c)ESTS 3 (d)ESTS 4

Figure 4.2.: ESTSs for nested simulation.

value will be multiplied by 10 and in ESTS 4 the value will be divided by 10. Both ESTSs send the calculated values back to subsystem 2. In subsystem 2 the calculated values can be received until the delay transitions become enabled.

Figure 4.3 shows the generated execution tree of the nested simulation. The nodes hold one state of every ESTS, where the first in the list are the states of subsystem 2 and after them there are the states of subsystem 3. First, the enabled transitions of subsystem 2 were executed. There were two transitions enabled and because of the full interleaving simulator they were executed in two different execution paths. Afterward, subsystem 3 received the messages from the output transitions with the corresponding input transitions. On the second turn of the simulator of subsystem 2 on every execution path the second enabled output transition which was not executed on the execution path at first turn was executed. In subsystem 3 the newly sent message was received and an output transition was executed. Due to the simulator of subsystem 3 is a full

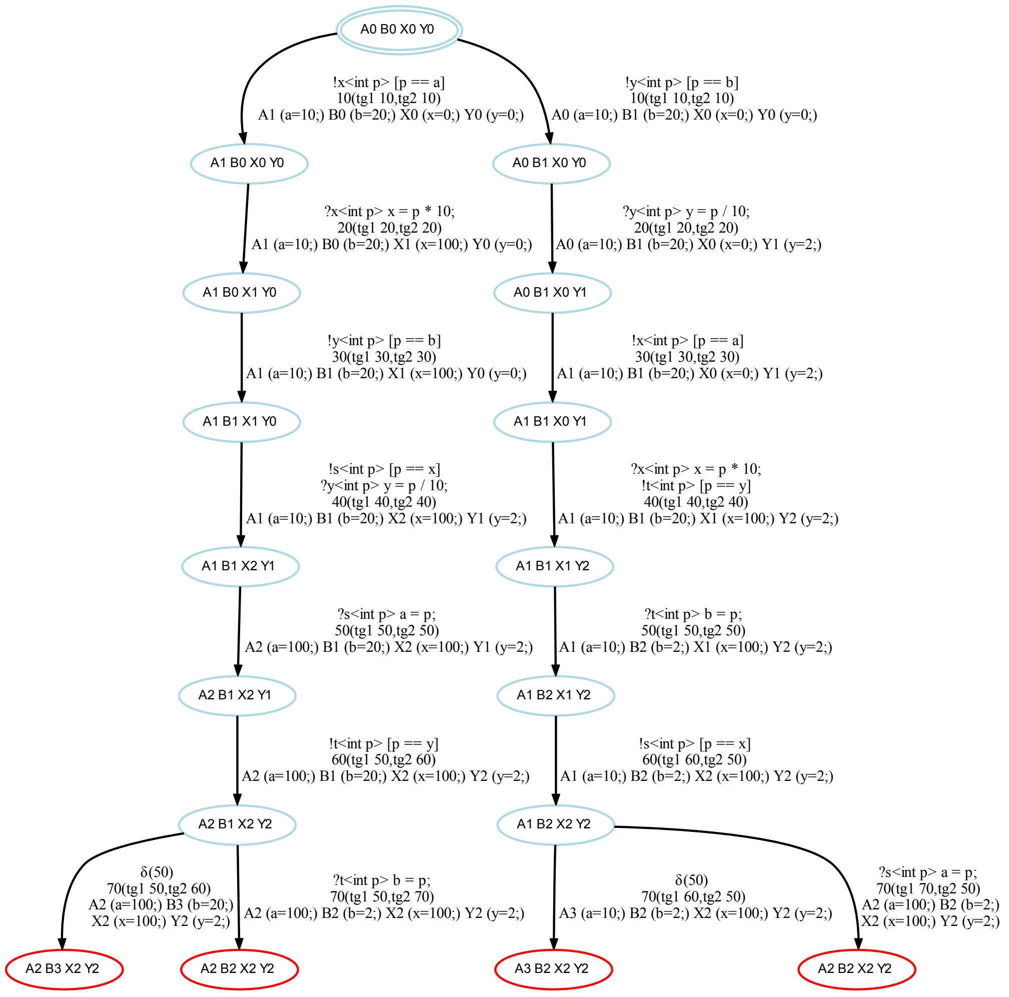


Figure 4.3.: Execution tree of a nested simulation.

parallel simulator more than one transition can be executed at same time. On turn three the message from subsystem 3 can be received from subsystem 2 before the delay transition became enabled. In subsystem 3 the second output transition was executed in this term. On the last turn in both execution paths a delay and an input transition were enabled and therefore they are executed and two new execution paths are created.

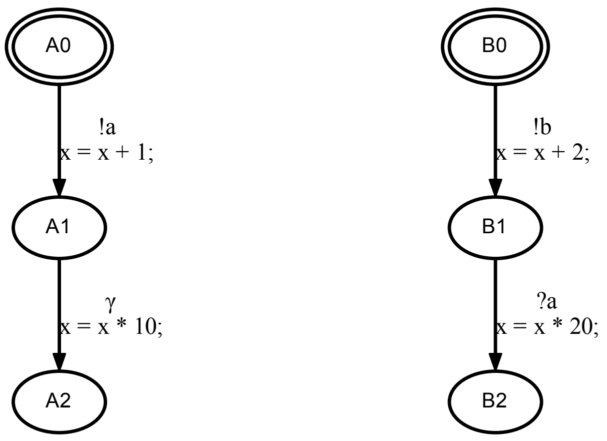
In this example the simulation of the nested simulations leads to four different execution paths with three different constellations of states and attributes at the leaf nodes. These constellations are, first subsystem 2 is able to receive all messages with the calculated values from subsystem 3. Second, and third subsystem 2 can only receive either one of the two messages from subsystem 3 because of the execution of the delay transition. □

4.2. Shared Attributes

A shared attribute is an attribute which is used in more than one ESTS. Therefore, changes of the shared attribute value have effect to all ESTSs sharing the attribute. In comparison to a programming language like C a shard attribute is similar to a global variable.

A shared attribute is defined in multiple ESTS simulators and all nested simulators which ESTSs include an attribute with the same name shares the same value. In relation to the hierarchy structure of the multiple ESTS, the sharing of the attributes does not apply to simulators which are above in hierarchy. The attributes are shared from the defining simulator downwards.

The use of shared attributes impacts the merging of child nodes of the full parallel simulator. In the case two transitions which are executed at the same time with same execution duration changes the same shared attribute, the execution leads to a non-deterministic execution tree. The attribute can not have both values. Therefore, the simulation creates a new execution path. One path uses the value of the shared attribute of one transition and the other path uses the value of the second transition.



(a)ESTS 1 (b)ESTS 2

Figure 4.4.: ESTSs for shared attribute example.

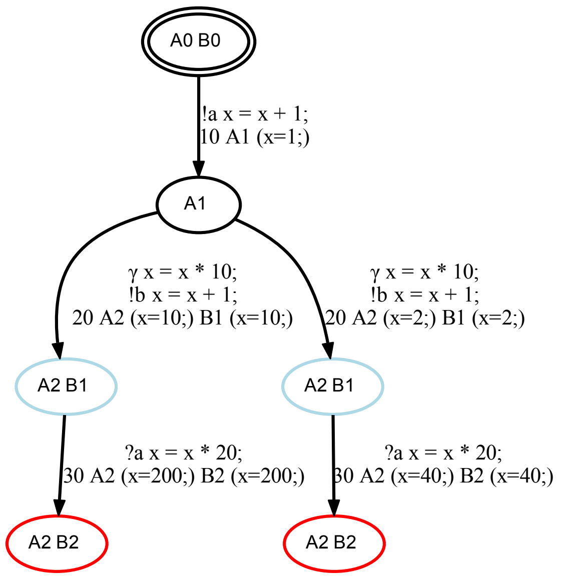


Figure 4.5.: Execution tree of a simulation with a shared attribute.

Example 4.2. (Shared Attribute)
The two ESTSs in Figure 4.4 are simulated with a full parallel simulator in this example, where the attribute `x` is a shared attribute. The transitions of the two ESTSs have an execution duration of 10 time units, except the output transition `b`. This transition needs 20 time units for execution. Every transition changes the value of the shared attribute `x` to show the impact of changing the value of shared attributes.

The resulting execution tree is displayed in Figure 4.5. The output transitions of the start states `A0` and `B0` are both enabled and were executed. The output transition `b` needed more time for execution as output transition `a`. Therefore, the first created node holds only one state and the output transition is part of the set of running transitions. In the next step the output transition `b` and the completion transition of state `A1` were executed and both were ready with execution at the same simulation clock valuation. Due to the parallel simulator executed both transitions at the same time and both change the value of the shared attribute, a new execution path was added to execution tree. In one path `x` holds the value of the completion transition and in the other path `x` holds the value of the output transition `b`. At the end the input transition was executed in both execution paths. The shared attribute has already different values in every execution path. □

4.3. Nested Simulation State

Another kind of nesting are Nested Simulation States (NSSs), where a state can contain a number of further ESTSs. That means a state can include nested functionality which will be activated by reaching the state.

Definition 4.1. (Termination State)
A state which has no outgoing transitions is a termination state. □

A NSS is a state which includes a single or multiple ESTS simulator. The simulator is starting by reaching the state and is executed until all states of the ESTSs of the simulator are termination states. In the case all current states of the simulator are terminated the simulator terminates and returns to the corresponding NSS.

The Algorithm 3.2 for executing a transition has to adapt to be able to handle such states. If the new state of a new child node is a NSS the nested simulator becomes active and will be started. Therefore, the start states of the nested simulator's ESTSs replace the NSS in the new child node. In further consequence the transitions of the ESTSs of the nested simulator will be executed instead of the enabled transitions of the NSS. Due to the nested simulator can be terminated, this behavior has also to check at this point. In the case all states of the ESTSs of the nested simulator are terminated, the states are replaced through the corresponding NSS. Thereby the enabled transitions of this state can be executed after the nested simulator is terminated. The activation and deactivation of the nested simulators run in a loop until neither a nested simulator is activated nor deactivated. This is done because states of the nested simulators can be also NSSs and therefore by activating a nested simulator another one can become active and vice versa.

Example 4.3. (Simulation with NSSs)
In this example the functionality of NSSs is shown. In Figure 4.6 a model with two NSSs is displayed. This model consists of five different ESTSs which are identifiable by the start state (double framed) and the termination state (gray filled). The NSSs in this model are `A1` and `C1` where every state includes two ESTSs. In addition, the used attributes `x` and `y` are shared attributes.

The ESTSs are part of a simulator and in this example we show the impact of using two different multiple ESTS simulators. The single ESTSs are in both runs simulated with a single step simulator.

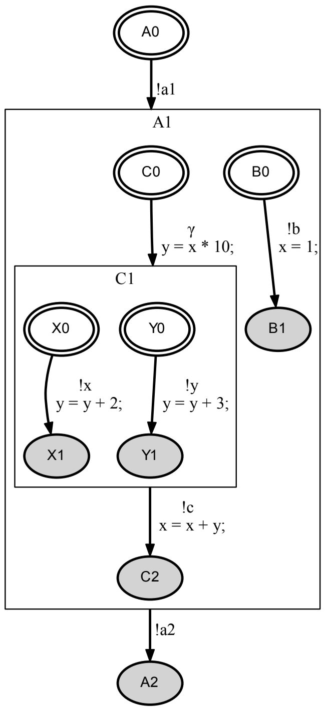


Figure 4.6.: Model with NSSs consisting of different ESTSs.

In Figure 4.7 the execution tree is shown for a simulation where the nested simulators are both full parallel simulators. At the beginning the output transition a was executed. Therefore, the NSS A1 was reached and the first nested simulator was started. Instead of the state A1 the node holds the start states of the nested simulator's ESTSs B0 and C0. Both states have an enabled transition which were executed parallel. Again, a NSS was reached and the second nested simulator was started by replacing the state C1 by the start states X0 and Y0. The enabled transitions were executed and because two transitions change the value of the same shared attribute a second execution path was created. After execution of the output transitions x and y the termination states X1 and Y1 were reached. The nested simulator terminated and the corresponding NSS C1 replaced the termination states. The next step executed the output transition c and entered the termination state C2. At this point all states of the first nested simulation's ESTSs are termination states which leads to the termination of the nested simulator. The NSS A1 replaced the termination states and the last transition was executed.

The second simulation of the ESTSs in Figure 4.6 is made with two different types of multiple ESTS simulators for the nested simulators. The ESTSs of the

NSS A1 are simulated with a full interleaving simulator and the ESTSs of NSS C1 are simulated with a full parallel simulator. A fragment of the generated execution tree is shown in Figure 4.8. The whole execution tree is too big and therefore some execution paths are not shown.

The execution of the first transition and the start of the first nested simulation is equal to the simulation before. The created child node has two enabled transitions and because of the full interleaving simulator both were executed, but in different execution paths. Therefore, in the different execution paths the second nested simulator is initialized at different times with different values for the attribute y. The second nested simulator executed the two output transitions parallel. The output transitions changed the same shared attribute at the same time and therefore a new execution path had to be created. This execution paths were removed from the execution tree because the execution tree is too big and these execution paths do not matter in this example.

On a closer look of this execution tree in Figure 4.8, the point in time of the execution of the output transition b deciding for the value of the attribute x in the leaf nodes. In the first execution path b was executed before the completion transition between the states C0 and C1. In further consequences the second nested simulator started with the value 10 for attribute y and in the leaf node the attribute x holds the value 13. In the second and third execution paths b was executed after the completion transition. Attribute y had at start of the second nested simulator the value 20 and x had the value 23 at the leaf nodes. This two execution paths terminate with the same value for attribute x similar to the first execution path of the execution tree in Figure 4.7. However, it has to be considered that the simulation clock valuation is not equal. The simulation clock valuation at the leaf nodes of the second execution tree are greater than the simulation clock valuation at the first execution tree's leaves. This difference arose from the use of different multiple ESTS simulators for the first nested simulator. Last but not least, the fourth execution path of the second execution tree gets our attention. In this execution path the output transition b was executed after the output transition c and therefore the attribute x got the value 1. □

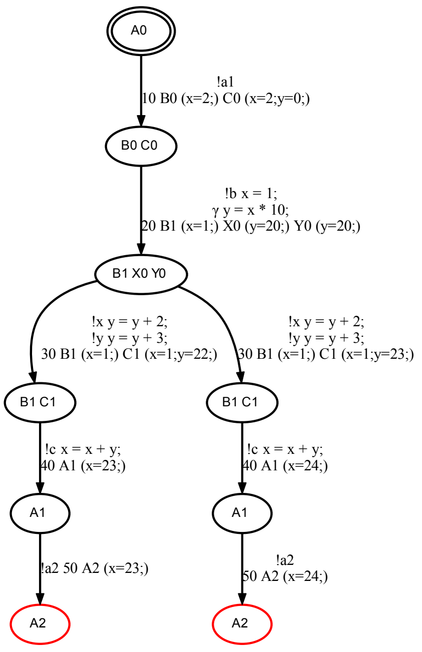


Figure 4.7.: Execution tree of a simulation with NSSs (full parallel).

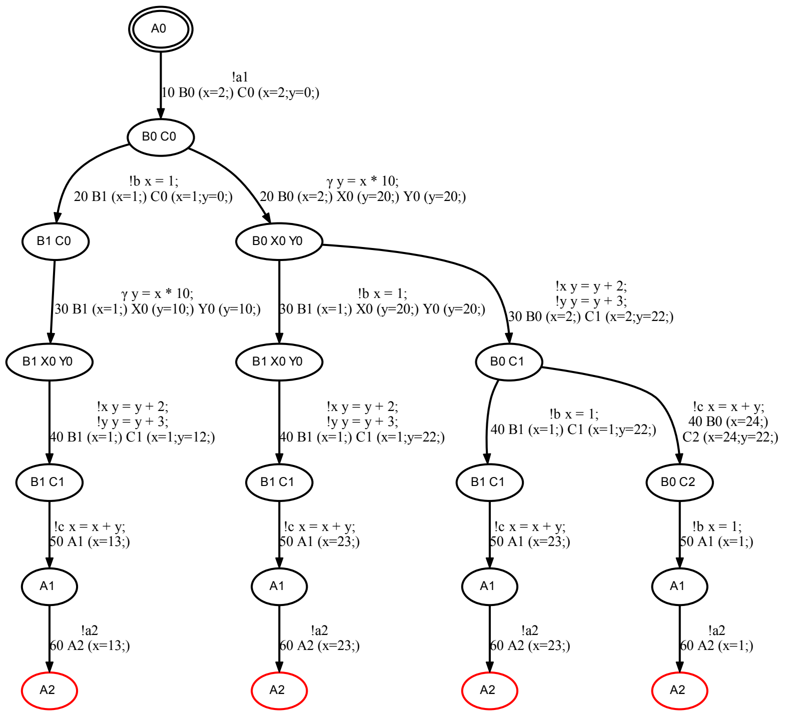


Figure 4.8.: Fragment of an execution tree of a simulation with NSSs (full interleaving).

5. Simulation Executors

Simulation executors are programs which simulate an environment for ESTSs simulators. They are responsible for sending messages to the simulators and for receiving the outputs from them.

The messages for the input transitions especially for the external input transitions (Definition 3.14) have to be generated for the simulators. In this section there are three different approaches how messages can be created. The first approach is a random one, where the messages are generated randomly, the second works with precalculated values. The third one works like a wrapper which gets inputs from external and passes them to the simulators.

5.1. Random Walk

The random walk is a simulation executor which generates messages randomly for the executing simulator. Therefore, a constraint solver is needed, which is able to calculate a parameter assignment to get a true evaluating guard. Only in the case such a parameter assignment can be found a message is sent to the simulator. The values for the parameters are randomly generated with respect to the restriction of the guard. Also, the send time of the message can be randomly calculated with the lower bound of the current simulation time. The simulator has to elapse the time difference to enable the input transition by adding a message with a send time higher than the simulation time.

Not only input transitions are of interest for the random walk also the delay transitions are important. In the case a delay transitions has a guard evaluating to true the random walk can signal the simulator to elapse time to enable a delay transition.

The leaf nodes of the execution tree are considered by the random walk. If there are more than one blocked node or a node has more than one input or delay transitions one leaf node and one input transition or delay transition is chosen randomly. After every stop of the simulator one message is sent to the simulator or the simulator is signaled to wait for a delay transition.

Algorithm 5.1 Random Walk

```
1: function RANDOM_WALK($simulator, max\_time$)
2:     $execution\_tree \leftarrow simulator$.GET_INITIALIZED_EXECUTION_TREE( )
3:     $sim\_state \leftarrow EXECUTED$
4:     while $sim\_state = EXECUTED$ do
5:         $leaf\_nodes \leftarrow execution\_tree$.GET_BLOCKING_LEAF_NODES( )
6:         SHUFFLE($leaf\_nodes$)
7:         for all $current\_leaf\_node \in leaf\_nodes$ do
8:             $message \leftarrow$CREATE_MESSAGE_OR_WAIT($current\_leaf\_node$)
9:             if $message = \emptyset$ then
10:                continue
11:            else if $message \neq EMPTY\_MESSAGE$ then
12:                for all $leaf\_node \in leaf\_nodes$ do
13:                    $leaf\_node$.ADD_MESSAGE($message$)
14:                end for
15:            end if
16:            break
17:        end for
18:        $sim\_state \leftarrow$DO_SIMULATION($simulator, max\_time$)
19:    end while
20:    return $execution\_tree$
21: end function
```

In Algorithm 5.1 the random walk is defined. This algorithm gets as parameter a simulator that can be a single or multiple ESTS simulator and a maximum simulation time. The simulator is executed as long enabled transitions exists and the maximum simulation time is not exceeded. The generated execution tree will be returned after one of these exit criterion is fulfilled.

At the beginning the execution tree is initialized with the initialized execution tree of the given simulator. This execution tree consists only of the root node which holds all start states of all ESTSs of the simulator. In a loop the core of the random walk is proceeded as long the simulation state has the value `EXECUTED`. The blocking leaf nodes of the execution tree are obtained and the leaf nodes are shuffled in the list to get a random order of the leaf nodes. These leaf nodes are checked if a message can be generated or a delay transition will become enabled in the future. If no message was returned the next leaf node will be checked. In the case an empty message is returned the decision is fallen to wait for a delay transition. Otherwise, a randomly chosen and created message is returned which will be added to the list of messages of every leaf node. Afterward, the simulator is executed for all leaf nodes, where the added message will be processed or the time will be elapsed to enable a delay transition. In the case no transitions of all leaf nodes can be executed `NO_ENABLED_TRANSITIONS` is returned or `INSUFFICIENT_TIME` if the maximum time is reached and therefore no more transitions can be executed. Otherwise, transitions are executed `EXECUTED` is returned and the loop will be continued.

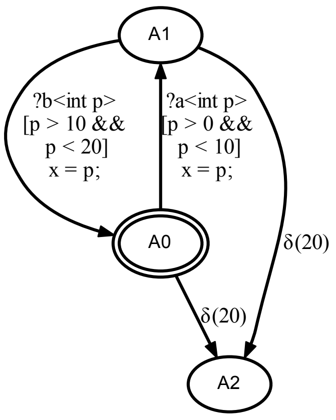
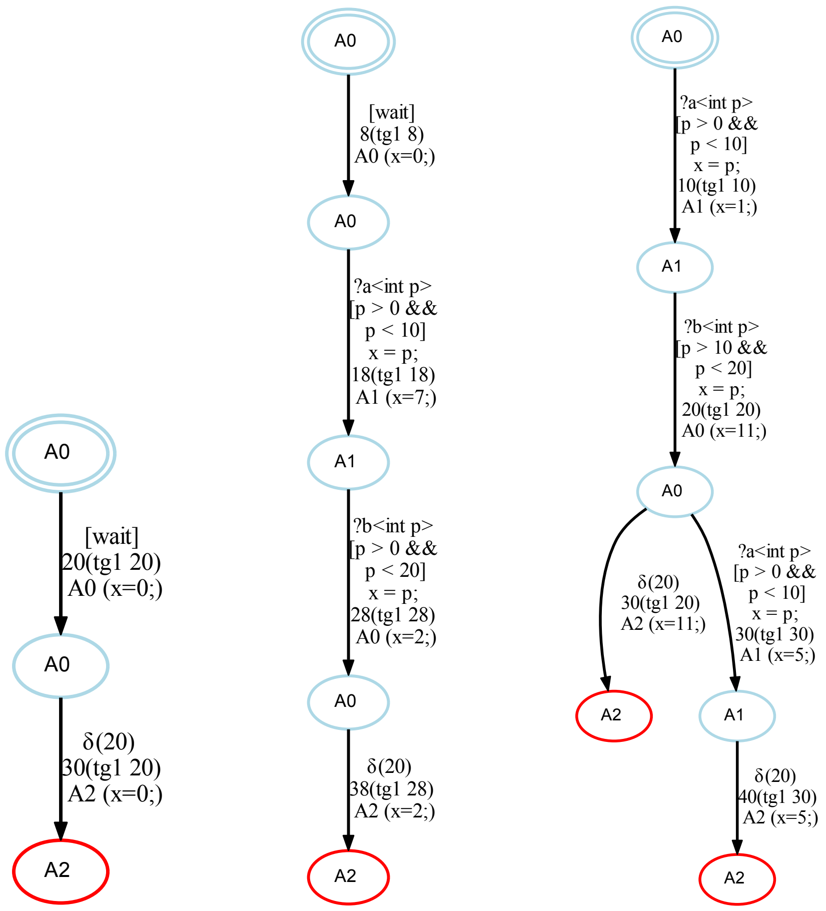


Figure 5.1.: ESTS for random walk example.

Example 5.1. (Random Walk)
This example shows three runs of a random walk for the same ESTS which is simulated by a single step simulator. The simulated ESTS is shown in Figure 5.1 which consists of three states, two input transitions, two delay transitions and one timing group. The states `A0` and `A1` are part of the timing group and the two input transitions `a` and `b` are external input transitions.

The three different execution trees of the three random walk runs are shown in Figure 5.2. In the first execution tree the random walk decided to wait for the delay transition at the root node. Therefore, 20 time units are elapsed and the delay transition became enabled and was executed.

At the second run a message for the input transition was sent with the send time 8 and with the value 7 for the parameter `p`. This value was created from a constraint solver with respect to the guard $p > 0 \land p < 10$. Due to the send time of the message is 8 these time units were elapsed. This is done to be able to receive the message by executing input transition `a`. Afterward, a message for input transition `b` was sent to the simulator which received it. The last transition which was executed was the delay transition. This has two possible causes, first



(a)Run 1 (b)Run 2 (c)Run 3

Figure 5.2.: Execution trees of random walks

the random walk chose the delay transition or second the random walk sent a message with send time higher than 28. Therefore, the delay transition was the only enabled transition at simulation time 28 and was executed.

The third run generated a non-deterministic execution tree. After sending the message for input transitions `a` and `b` the simulation held at state `A0`. Another message for input transition `a` was sent to the simulator with a send time 20. Therefore, the delay and the input transition were enabled and both were executed. After executing input transition `a` the delay transition was also executed at the second execution path. The parameter values of the created messages were always generated by a constraint solver with random values, where the upper and lower bounds of the guard were considered.

In this example three different random walk runs are shown with different results. There is a range of possible parameter values for the input transitions a and b. In combination with different waits there are a number of possible execution trees which a random walk can generate. □

5.2. Linear Walk

The linear walk does not generate messages for input transitions, but works with precalculated values instead. It gets an ordered list of messages which can have different origins. This list can consists for example of the created messages of a random walk. Another example could be a manual created list of messages which represents an accepted order of messages which the simulator should be able to process.

The aim of the linear walk is the possibility to be able to compare the created messages of the output transitions with the expected once for a given list of inputs. Furthermore, the messages can be sent to different variations of simulators for the given ESTSs. Therefore, the impact of different conditions for the same input can be revealed.

This simulation executor gets a simulator that can be a single or multiple ESTS simulator, an ordered list of messages and the maximum simulation time as input. The messages are added to the list of messages of the root node of the initialized execution tree of the simulator. Then the simulator is executed until the maximum simulation time is reached or the leaf nodes of the execution tree do not have enabled transitions.

5.3. Manual Walk

In contrast to, the simulation executors before this one does not make an automatic run. This is because the manual walk does not generate messages or gets the messages before starting. The inputs for the ESTSs have to send from external to initiate transition executions at the time they occur. The manual walk is a wrapper, where messages are created for the given inputs and they are sent to the simulator. The outputs from the simulator are received from the manual walk and are sent to the external program.

The manual walk is an interface to a simulator that can be a single or multiple ESTS simulator, where an external program is able to send inputs to the simulator and receive the outputs. This allows an interaction between an external program and the simulator. Due the ESTSs simulators need a maximum simulation time the simulation stops at the latest at this time.

6. Interface for ICOS

6.1. Independent Co-Simulation (ICOS)

ICOS[26] is a co-simulation framework where different simulation tools and HiL components are coupled with advanced coupling algorithms. The aim of ICOS is the simulation of an overall system which includes a vehicle, a driver and the environment [28].

There are a number of domain-specific simulation tools like Simulink®[1], Dymola®[2], SystemC®[3] or ASCET®[4] which have specific heterogeneous simulation environments. ICOS couples systems of different simulation tools with various engineering domains to an overall system. This enables the co-simulation of a cross-domain system.

In the Graphical User Interface (GUI) of ICOS a system which consists of a number of subsystems can be assembled. The inputs and the outputs of the subsystems have to specified and also how they are linked. The output of a subsystem can be the input of another subsystem. It must be ensured that all inputs of the subsystems are linked with an output. Therefore, at least one subsystem must not use inputs.

The co-simulation simulates the system for a defined number of durations where a duration represents a defined time unit. This means for example that one second elapses after each simulation step if one second is used as duration. At any duration the subsystems are applied with the defined input values. There are different modes which values the subsystems use. By default the inputs get the calculated values of the outputs of the previous duration.

[1]http://www.mathworks.com/simulink
[2]http://www.dymola.com
[3]http://www.systemc.org
[4]http://www.etas.com/ascet

6.2. Interface

Definition 6.1. (Wrapper)
A wrapper is a program which provides an interface to call a second program. The inputs of the wrapper are converted to be valid inputs for the second program and the outputs of the second program are also converted for the calling program. The calling program does not have to know anything about the second program it only has to know how to use the wrapper. □

The large collection of couplings to existing simulation tools does not include a simulation tool which simulates a transition system. To close this gap an interface to the ESTS simulator was implemented. On the basis of this interface ESTSs and transformed UML State Machines can be co-simulated with ICOS.

Due to the use of different program languages ICOS is implemented in C++ and the ESTS simulator is implemented in Java two wrappers are developed. One wrapper is written in C++ and is called from ICOS and the other one is written in Java and is called from the C++ wrapper and calls the ESTS simulator. To be able to call a Java method from C++ Java Native Interface (JNI) is used. JNI enables programs written in a non-Java language to call Java code which runs in a Java Virtual Machine (JVM).

A file specification was developed to be able to save a definition of an ESTS in a file. In the GUI a file is selected which contains the ESTS to simulate. For ICOS only deterministic ESTSs are supported because it is a deterministic simulation where every output can only have one value at a time.

6.2.1. C++ Wrapper

This wrapper starts a JVM and in this JVM the Java wrapper. Afterward, the ESTS simulator is initialized with the ESTS file that was defined in the GUI before, by passing the file path and name to the Java wrapper. This is done via JNI which is able to call Java methods which are currently available in the JVM.

At every duration the C++ wrapper gets the inputs from ICOS. These will be prepared for the JNI call and are passed to the Java wrapper. Additionally the simulation time is passed which will be the maximum simulation time for the simulator. The output transitions which are executed while simulating are fetched from the Java wrapper and they are returned to ICOS.

The simulator has to deliver values for every defined output at the end of the simulation time of the duration. In the case not all required output transitions were executed the simulation terminates with a failure.

6.2.2. Java Wrapper

This wrapper is the counterpart of the C++ wrapper on the Java side to execute the ESTS simulator. More precisely the manual walk is used to execute the ESTS simulator. The Java wrapper gets the file path and name from the ESTS file from the C++ wrapper. An ESTS is created with the given file and with this model an ESTS simulator is created to initialize the manual walk.

The inputs from the C++ wrapper are received and are sent to manual walk. After receiving the inputs for one duration, the simulation is started where the given simulation time is used as maximum simulation time for the simulator.

The main reason for using the Java wrapper instead of calling directly the manual walk is the handling of the outputs. ICOS specifies that on the end of the duration every defined output has to have a value. Value means in this context the valuation of the output parameters. The execution path can be seen as time line where an output has at any time the same value as the execution of the output before. Before the first execution, the value of an output is undefined and after the first execution the output has a specific value. This value is valid until another execution of the output is done. After simulating the ESTS all output value changes which are done in the current duration are returned where the label of the output transition, the parameter valuations and the simulation time after the execution are delivered to the C++ wrapper. Additionally the output values at the simulation time of the duration are returned to fulfill the ICOS specification.

Example 6.1. (Electricity Controller)
In this example an electricity controller of a hybrid electric vehicle which has an internal combustion engine and additionally an electric motor using a battery is simulated. The electricity controller calculates the percentage share of the usage of the electric motor based on the charge level of the battery. The higher the charge level the more the electric motor is used to support the internal combustion engine.

The ESTS which defines the electricity controller is shown in Figure 6.1. It has one input transition `battery` and one output transition `electricity`. The completion transitions are used to set the attribute `electricity` based on the

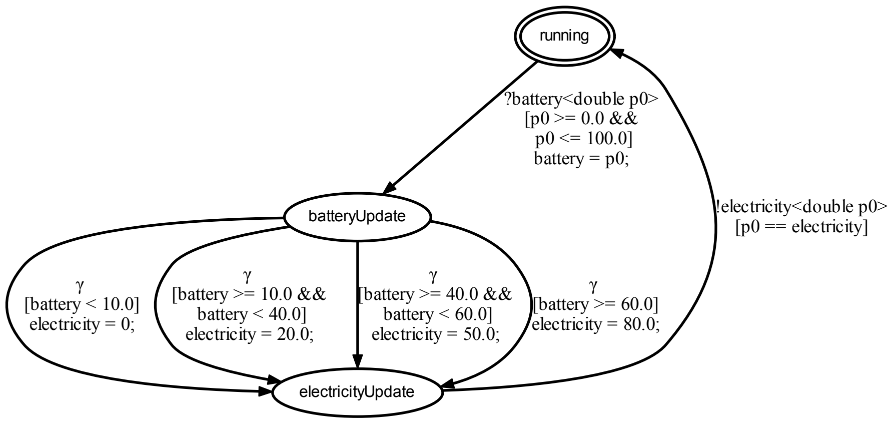


Figure 6.1.: ESTS of the electricity controller.

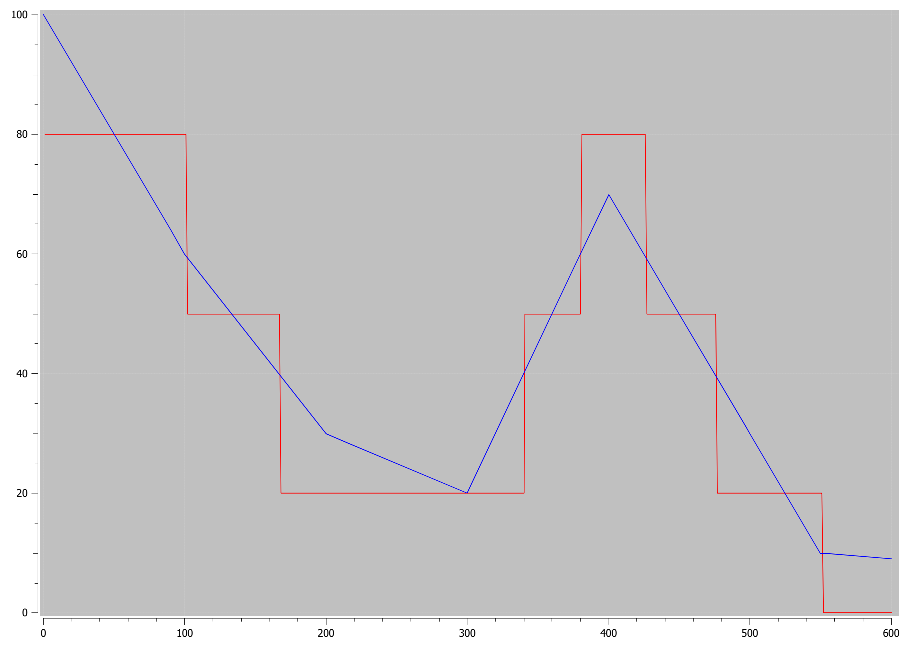


Figure 6.2.: Plot of the calculated electricity usage (red) and the battery charge condition (blue).

value of the attribute `battery`. In the case the battery charge level is below 10% the electric motor will not be used. Is the battery charge level between 10% and 40% than the electric motor is used for 20% and so on.

The battery charge level is a predefined static curve that is used as input for the ESTS. This curve is plotted in blue in Figure 6.2. The plot displays the battery charge level in percent over time in seconds.

ICOS was executed for 600 durations where a duration was one second. The red curve in Figure 6.2 is the output of the simulated ESTS which is the calculated percentage share of the usage of the electric motor. At the begin the battery charge level was 100% and the car used 80% electrical power and 20% from the internal combustion engine. The battery charge level decreased and at the point it is under 60% the ESTS returned for the electrical usage 50%. After the charge level decreased under 40% only 20% electrical power is used. At second 300 the battery was charged and at the point the charge level reached 40% again more power of the electrical motor was used. The charging ended at second 400 thenceforth the charge level decreased and at second 550 the charge level is under 10%. Therefore, the electronic motor is not used at the last 50 seconds. □

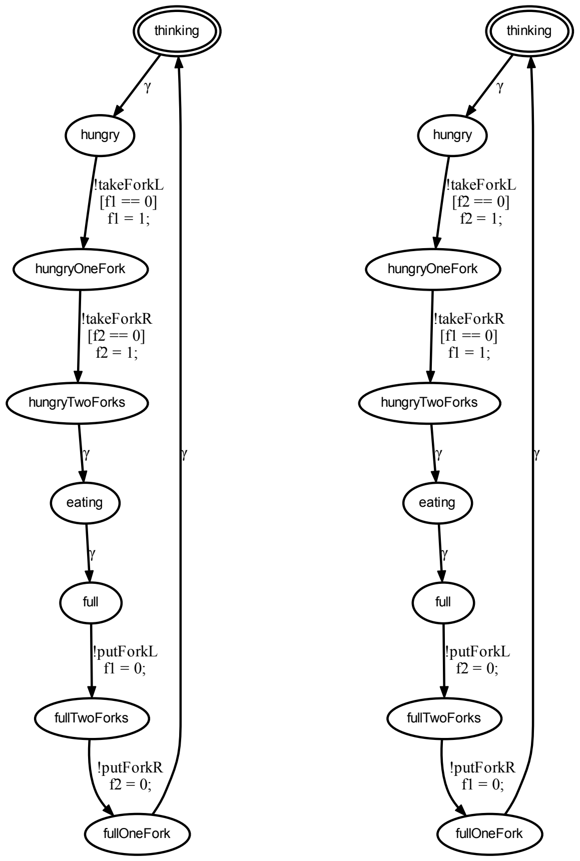
7. Experimental Results

In this chapter the applicability of simulating ESTSs is shown. The resulting execution tree of a simulation gives information about the behavior of simulating in different environments. The full interleaving simulator executes all enabled transitions of a state and therefore all possible execution orders for a specific initialization and inputs results. Due to this fact the execution tree can be checked for unexpected behaviors. This can be caused for example by faulty models, unexpected occurrences of race conditions or unexpected time behavior. The full parallel simulator provides information about the behavior of the models in a parallel environment. Not all models are ready for full parallel executions and therefore probably arising unexpected behaviors can be seen in the execution tree.

There is also an example which shows the co-simulation of Simulink® models and an ESTS in ICOS. An air controller for a physical plant is simulated where the physical plant is simulated with Simulink® and the air controller with an ESTS simulator.

7.1. Deadlock Detection

A situation where two or more actions wait for each other to continue is called deadlock. A famous problem to show the occurrence of deadlocks is the dining philosophers problem. We use it to show the ability to detect deadlocks by simulating ESTSs. In this example two philosophers are sitting on a table where each of them has a plate, but there are only two forks placed. In the middle of the table is a bowl of spaghetti where the philosopher is allowed to serve himself in the case he has both forks. The philosophers alternately think and eat. However, because of the limitation of the forks only one philosopher is able to eat at a time. A hungry philosopher takes a fork if it is available unless he waits for it. A deadlock occurs in the case every philosopher has taken one fork and waits for the second.

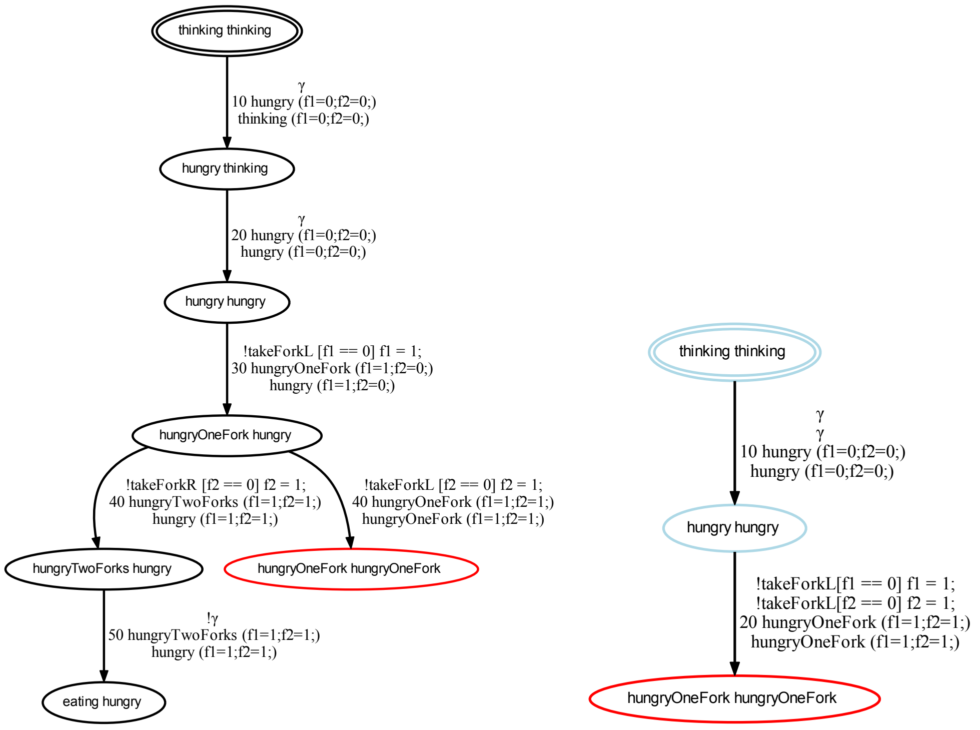


(a)Philosopher 1 (b)Philosopher 2

Figure 7.1.: ESTSs of two philosophers.

In Figure 7.1 two ESTSs are shown which are the models for the two philosophers. They are thinking and after they become hungry they take the forks. With both forks they eat and when they are full they put the forks back on the table and think again. The attributes `f1` and `f2` are shared attributes which can have the values 0 and 1. 0 means the fork is available and 1 means the fork is in use. The main difference between these two models is that philosopher 1 takes first fork 1 and then fork 2 and philosopher 2 takes first fork 2 and then fork 1. After eating, they put the forks back on the table in the same order they have taken them.

A simulation with the deterministic simulator will not result in a deadlock because only the first added ESTS will be simulated. The models do not terminate and therefore the deterministic simulator simulates the model until the maximum simulation time is reached.


thinking thinking
γ
10 hungry (f1=0;f2=0;)
thinking (f1=0;f2=0;)
hungry thinking
γ
20 hungry (f1=0;f2=0;)
hungry (f1=0;f2=0;)
hungry hungry
!takeForkL [f1 == 0] f1 = 1;
30 hungryOneFork (f1=1;f2=0;)
hungry (f1=1;f2=0;)
hungryOneFork hungry
!takeForkR [f2 == 0] f2 = 1;
40 hungryTwoForks (f1=1;f2=1;)
hungry (f1=1;f2=1;)
!takeForkL [f2 == 0] f2 = 1;
40 hungryOneFork (f1=1;f2=1;)
hungryOneFork (f1=1;f2=1;)
hungryTwoForks hungry
hungryOneFork hungryOneFork
!γ
50 hungryTwoForks (f1=1;f2=1;)
hungry (f1=1;f2=1;)
eating hungry
thinking thinking
γ
γ
10 hungry (f1=0;f2=0;)
hungry (f1=0;f2=0;)
hungry hungry
!takeForkL[f1 == 0] f1 = 1;
!takeForkL[f2 == 0] f2 = 1;
20 hungryOneFork (f1=1;f2=1;)
hungryOneFork (f1=1;f2=1;)
hungryOneFork hungryOneFork

(a)Full Interleaving (b)Full Parallel

Figure 7.2.: Execution trees of the simulation of two dining philosophers.

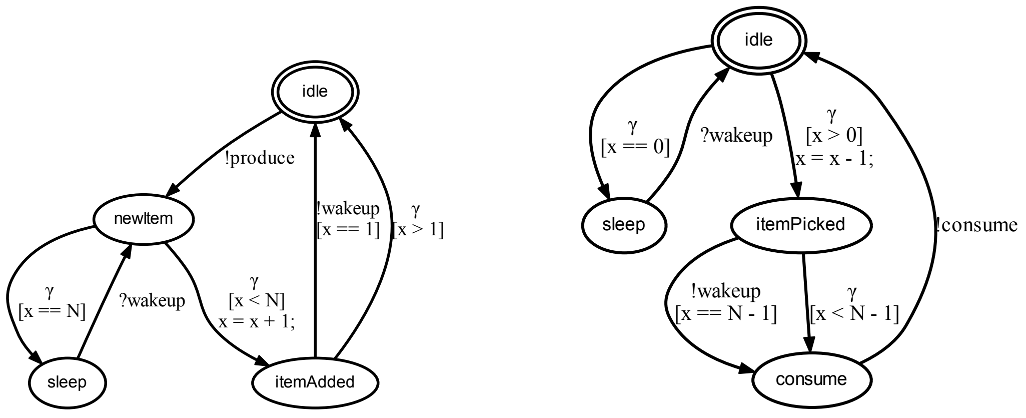
More interesting are the simulation results of the full interleaving and full parallel simulator which are shown in Figure 7.2. The full interleaving simulator generates a non-deterministic execution tree where a fragment of this tree is displayed in Figure 7.2a. There are two execution paths shown, in one the philosopher 1 is eating and in the other one the execution results in a deadlock. The second execution tree, shown in Figure 7.2b, is the result of a full parallel simulation. The deadlock occurs pretty soon because the two philosophers did everything simultaneously. They became hungry and took a fork always at the same time. Afterward, they wait for the second fork which never will be available in this situation.

7.2. Race Condition

A race condition occurs if the time at which inputs are received lead to different states during the simulation. In the case the inputs are received in an order the programs are not implemented for an unexpected behavior can occur. There are different ways which create race conditions, where three of them are discussed in more detail. First, we discuss an unexpected update of a shared resource due to a race condition. Second the write-write conflict where two or more simultaneously executed programs write to the same shared resource. Therefore, an update can be replaced by another without considering the previous update which is called lost update. The third race condition is the parallel update of a shared resource which leads to a non-deterministic behavior and can be also an unexpected behavior.

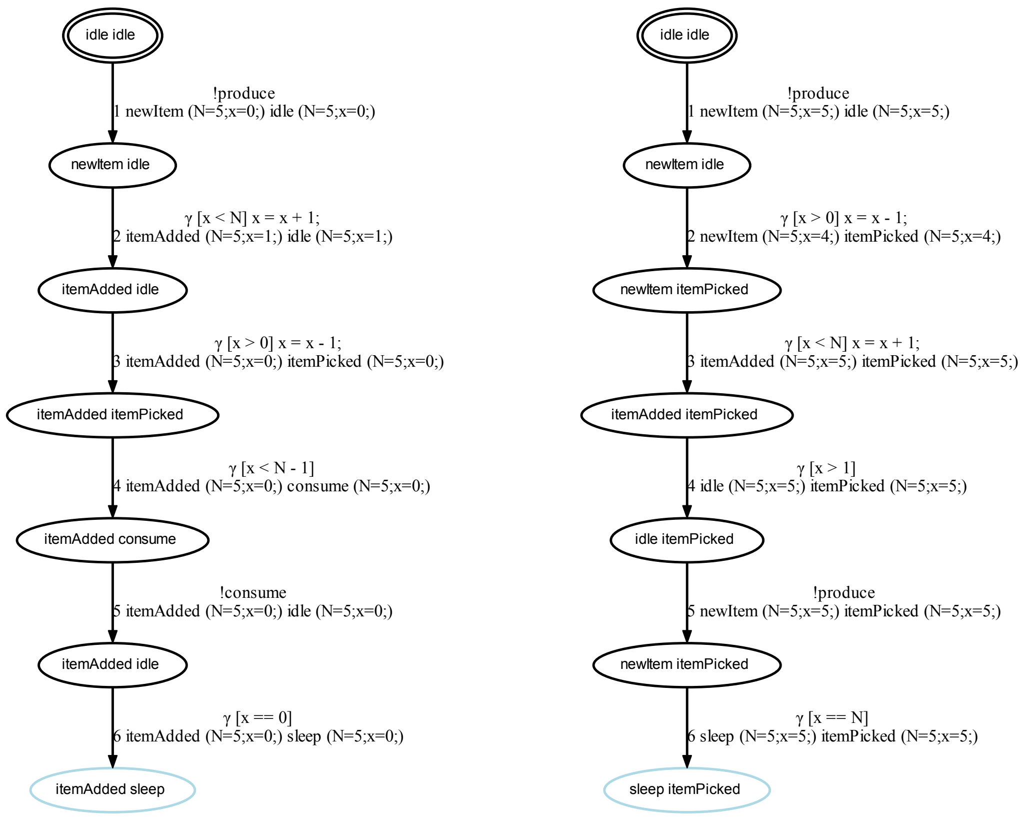
7.2.1. Unexpected Update

The producer consumer problem is a classic example to show different problems of a multi-process environment. The producer and consumer share a buffer with a fixed size where the producer fills the buffer and the consumer empties it. In Figure 7.3 the ESTSs for producer and consumer are displayed. Instead of a buffer the models increase and decrease the shared attribute `x` and the shared attribute `N` which defines the maximum value of `x`. The producer produces an item and `x` is increased by 1 if $x < N$. In the other case where `x` has reached its maximum the producer goes to state `sleep`. At this state it waits for a `wakeup` of the consumer. After increasing `x` the producer goes to state `idle` and in the case $x = 1$ a `wakeup` is sent to the consumer. This is done because before the increasing of `x` it is 0 and in this case the consumer could be in state `sleep`. The customer decrease the shared attribute `x` in the case it is greater than 0. In state `itemPicked` a `wakeup` is sent to the producer in the case $x = N - 1$ because `x` has reached the maximum before decreasing and the producer could sleep.



(a)Producer (b)Consumer

Figure 7.3.: ESTSs of a producer and a consumer.


idle idle
!produce
1 newItem (N=5;x=0;) idle (N=5;x=0;)
newItem idle
γ [x < N] x = x + 1;
2 itemAdded (N=5;x=1;) idle (N=5;x=1;)
itemAdded idle
γ [x > 0] x = x - 1;
3 itemAdded (N=5;x=0;) itemPicked (N=5;x=0;)
itemAdded itemPicked
γ [x < N - 1]
4 itemAdded (N=5;x=0;) consume (N=5;x=0;)
itemAdded consume
!consume
5 itemAdded (N=5;x=0;) idle (N=5;x=0;)
itemAdded idle
γ [x == 0]
6 itemAdded (N=5;x=0;) sleep (N=5;x=0;)
itemAdded sleep
idle idle
!produce
1 newItem (N=5;x=5;) idle (N=5;x=5;)
newItem idle
γ [x > 0] x = x - 1;
2 newItem (N=5;x=4;) itemPicked (N=5;x=4;)
newItem itemPicked
γ [x < N] x = x + 1;
3 itemAdded (N=5;x=5;) itemPicked (N=5;x=5;)
itemAdded itemPicked
γ [x > 1]
4 idle (N=5;x=5;) itemPicked (N=5;x=5;)
idle itemPicked
!produce
5 newItem (N=5;x=5;) itemPicked (N=5;x=5;)
newItem itemPicked
γ [x == N]
6 sleep (N=5;x=5;) itemPicked (N=5;x=5;)
sleep itemPicked

(a)Bug in producer. (b)Bug in consumer.

Figure 7.4.: Execution paths showing unexpected race conditions which are leading to a bug.

A full interleaving simulation with single step simulators for producer and consumer generates an execution tree. This tree consists of all possible execution orders of these two ESTSs for a given simulation time and for a specific initial attribute valuation. That means the generated execution tree contains all possible race conditions for the given conditions. In this example the simulation is started once with $x = 0$ and once with $x = 5$. The attribute `N` is in both simulations 5. The resulting execution tree contains two execution paths which are shown in Figure 7.4. These execution paths are displayed because they identify two bugs in the ESTSs caused by unexpected race conditions.

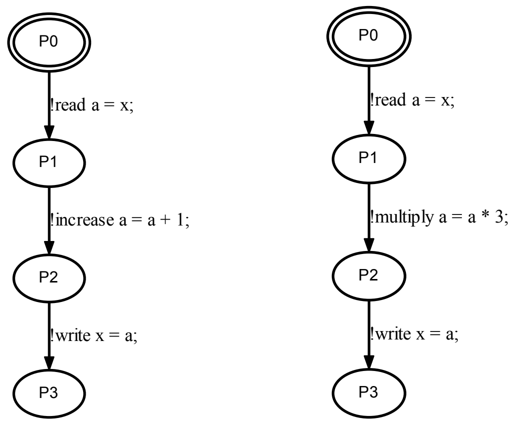
The first simulation starts with `x` is 0 and therefore the producer can execute the output transition `produce` and the increasing of `x` by 1. After the increasing, the consumer was allowed to execute its transitions. Due to `x` was 1 the consumer decreased `x` by 1 and executes the output transition `consume`. The consumer is again in state `idle` but this time `x` was 0 and therefore it changed its state to `sleep`. The consumer waited for a `wakeup` of the producer, but the producer was not able to execute a transition. This was caused by an incorrect guard $x > 1$ of the completion transition of the state `itemAdded`. Due to the state `itemAdded` is the state after the increase of `x`, `x` should be greater or equal to 1. In the case `x` is 1 the output transition `wakeup` and in the other case the completion transition is executed. Due to the race condition the shared attribute `x` was 0 and no guard could evaluate to true.

The second simulation was initialized with $x = N = 5$. In the execution path (Figure 7.4b) producer and consumer alternated at the begin. The producer produced an item and the consumer picked one. Therefore, the producer was able to add the new item by increasing `x` by 1 and so `x` reached again the maximum. The producer output transition `produce` was executed a second time and because $x = N$ it went to sleep. In this situation a second bug was found, the guard of the state `itemPicked` of the consumer are also incorrect with respect to the occurrence of race conditions.

7.2.2. Write–Write Conflict

A write-write conflict occurs when two programs write to the same shared resource without taking note of previous updates. One program updates a shared variable and the second program overwrite the value with a new one. The second program does not note the previous update of the first program. The race conditions are also responsible for this unexpected behavior and therefore it can be detected in the generated execution tree of a full interleaving simulator. To illustrate the possibility to detect such write-write conflicts two ESTSs are simulated with full interleaving simulator. The individual ESTSs are simulated with single step simulator to get an execution tree which includes all possible race conditions.

In Figure 7.5 two ESTSs are shown which read and write to the same shared attribute `x`. First, they read the value of `x` and save the value in a non-shared attribute `a`. The value of `a` is in one ESTS increased by 1 and in the other ESTS multiplied by 3. After the calculations the new values of `a` are written to shared value `x`. In the case the models are simulated deterministically no unexpected behavior will be arise. The expected values for `x` after execution are 6 and 4. The value 6 is the result if ESTS 1 is executed before ESTS 2 and the value 4 is the expected result for the reverse order of execution. The situation is different when the models are simulated simultaneously because in this case race conditions can be happening.



(a)ESTS 1 (b)ESTS 2

Figure 7.5.: ESTSs to show write-write conflict.

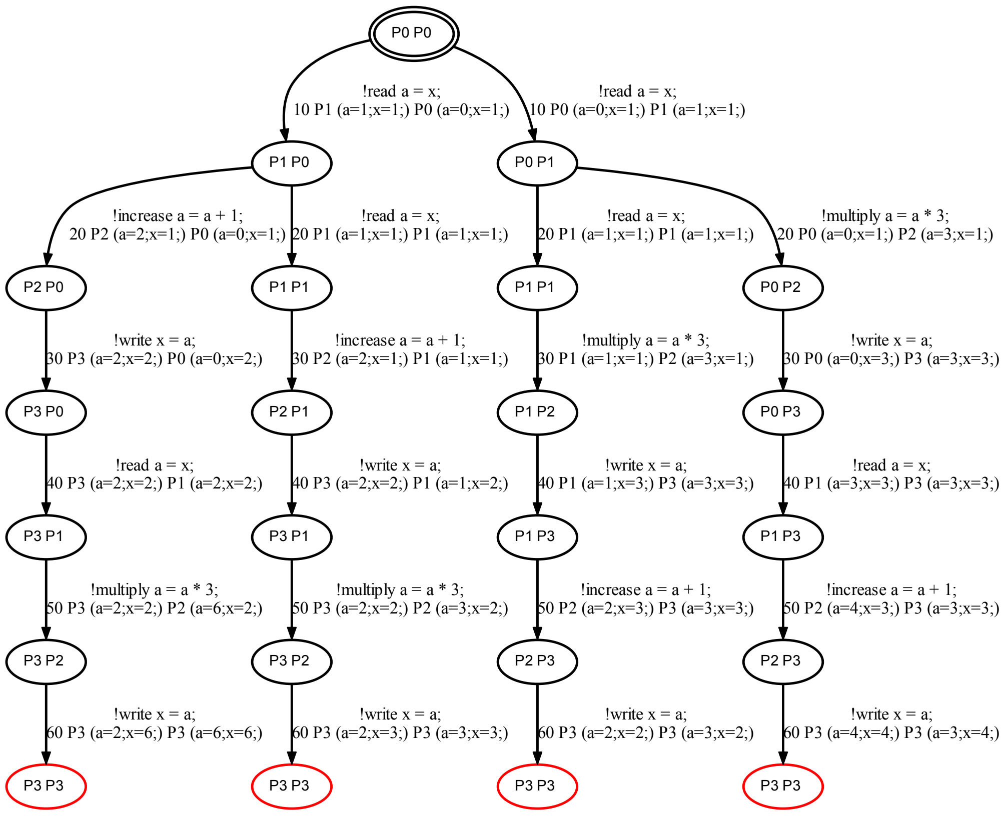

P0 P0
!read a = x;
10 P1 (a=1;x=1;) P0 (a=0;x=1;)
!read a = x;
10 P0 (a=0;x=1;) P1 (a=1;x=1;)
P1 P0
P0 P1
!increase a = a + 1;
20 P2 (a=2;x=1;) P0 (a=0;x=1;)
!read a = x;
20 P1 (a=1;x=1;) P1 (a=1;x=1;)
!read a = x;
20 P1 (a=1;x=1;) P1 (a=1;x=1;)
!multiply a = a * 3;
20 P0 (a=0;x=1;) P2 (a=3;x=1;)
P2 P0
P1 P1
P1 P1
P0 P2
!write x = a;
30 P3 (a=2;x=2;) P0 (a=0;x=2;)
!increase a = a + 1;
30 P2 (a=2;x=1;) P1 (a=1;x=1;)
!multiply a = a * 3;
30 P1 (a=1;x=1;) P2 (a=3;x=1;)
!write x = a;
30 P0 (a=0;x=3;) P3 (a=3;x=3;)
P3 P0
P2 P1
P1 P2
P0 P3
!read a = x;
40 P3 (a=2;x=2;) P1 (a=2;x=2;)
!write x = a;
40 P3 (a=2;x=2;) P1 (a=1;x=2;)
!write x = a;
40 P1 (a=1;x=3;) P3 (a=3;x=3;)
!read a = x;
40 P1 (a=3;x=3;) P3 (a=3;x=3;)
P3 P1
P3 P1
P1 P3
P1 P3
!multiply a = a * 3;
50 P3 (a=2;x=2;) P2 (a=6;x=2;)
!multiply a = a * 3;
50 P3 (a=2;x=2;) P2 (a=3;x=2;)
!increase a = a + 1;
50 P2 (a=2;x=3;) P3 (a=3;x=3;)
!increase a = a + 1;
50 P2 (a=4;x=3;) P3 (a=3;x=3;)
P3 P2
P3 P2
P2 P3
P2 P3
!write x = a;
60 P3 (a=2;x=6;) P3 (a=6;x=6;)
!write x = a;
60 P3 (a=2;x=3;) P3 (a=3;x=3;)
!write x = a;
60 P3 (a=2;x=2;) P3 (a=3;x=2;)
!write x = a;
60 P3 (a=4;x=4;) P3 (a=3;x=4;)
P3 P3
P3 P3
P3 P3
P3 P3

Figure 7.6.: Execution tree showing expected and unexpected behavior because of race conditions.

The generated execution tree of the full interleaving simulator consists of different execution paths which terminate with four different valuations of the shared attribute x. A fragment of the generated execution tree is shown in Figure 7.6 which contains four execution paths where each end with a different value for x. The two outer execution paths are executions which hold an expected value of x at the termination states. The left one executes first ESTS 1 and then ESTS 2 and the right one executes the ESTSs in a reverse order. These execution paths are out of race conditions and therefore x has an expected result.

The write-write conflict can be seen in the execution paths in the middle of the execution tree. At the begin both models read the value of x and therefore both calculated with the value 1. After calculating the models assigned the new value to x. Due to this execution order x holds always the value of the last update and the update before is lost.

7.2.3. Contemporaneous Update

In a parallel environment an update of a shared attribute by two different ESTSs at the same time is possible. The value of the attribute after such contemporaneous updates is undefined. That means the attribute holds one of the two values, but which value is unknown. Therefore, in such a case the full parallel simulation leads to a non-deterministic execution tree. However, in each execution path an update is lost and this can lead to unexpected results.

The previously used ESTSs of the Figures 7.3 and 7.5 are simulated with a full parallel simulator to illustrate contemporaneous updates. The result of the full parallel simulation of the write-write conflict ESTSs is shown in Figure 7.7. The two models had read, calculated and wrote always at the same time. Due to the simultaneously write the execution tree became non-deterministic. The shared attribute x got in one execution path the calculated value of the first model and in the other execution path the value of the second model. The resulting values for x are 2 and 3. This execution tree shows the parallel behavior of the models and in the case the resulting attribute valuations are incorrect the models are not ready of parallelism.

The second full parallel simulation uses the producer and consumer ESTSs (Figure 7.3) which generated the result shown in Figure 7.8. The producer produced a new item and the consumer went to sleep on the first state change. Afterward, the producer increased the shared attribute x and sent a `wakeup` to the consumer. Then the producer and consumer can execute transitions again parallel. The producer produced another item and the consumer received the

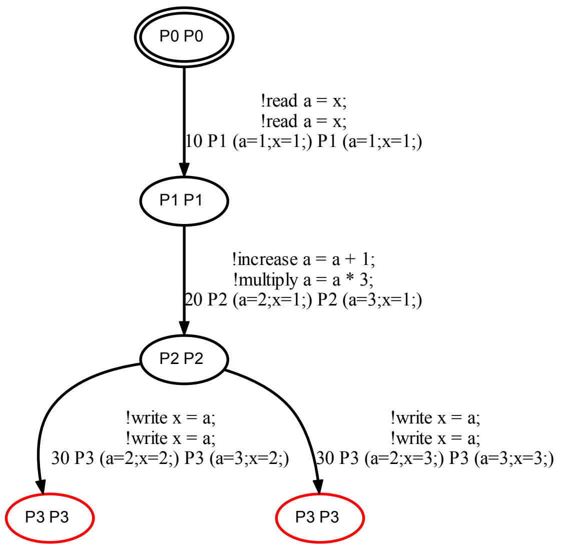


Figure 7.7.: Execution tree showing unexpected outputs because of race conditions.

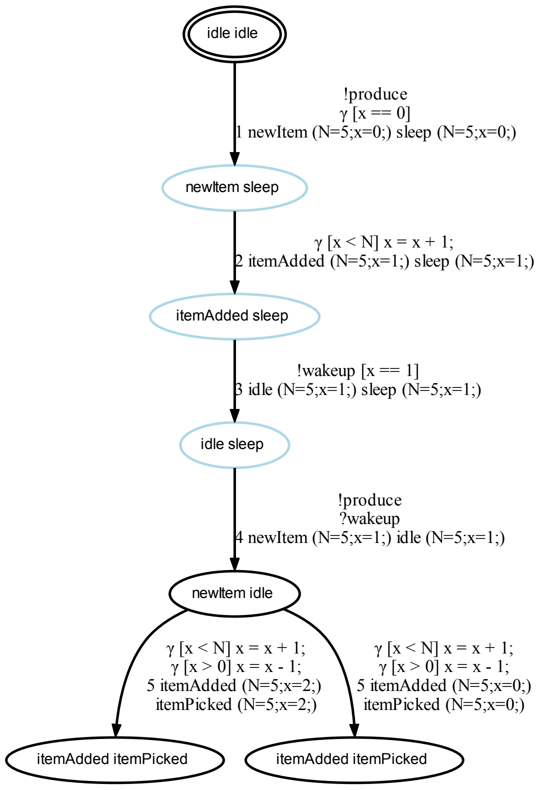
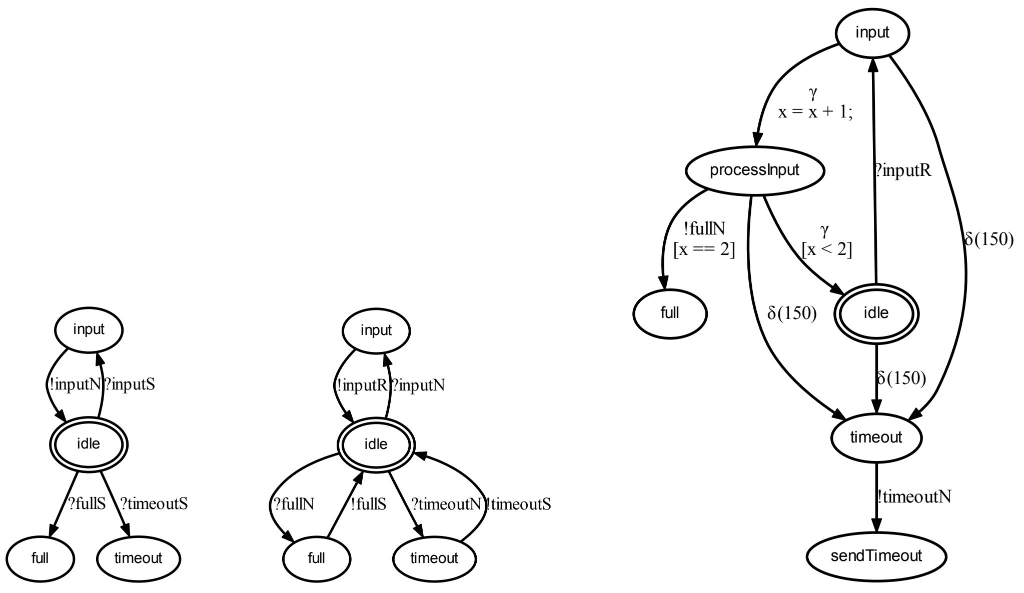


Figure 7.8.: Execution tree fragment of the simulation of consumer and producer.

wake up. The shared attribute x was increased because of the new produced item and the consumer decreased x because items were available. These also were done at the same time which resulted in a non-deterministic execution tree. In the first execution path x is 2 and in the second one x is 0. Both values are incorrect due to the increasing and decreasing of x whereas the value of x should be 1. Assuming that x is an index of a buffer the access with the value of the first execution path can lead to an index out of bounds exception. The second execution path will ignore the newly added item because x is 0 which means the buffer is empty.

7.3. Time Behavior

There are many reasons which can affect the execution time of a program. For example, the processor can also process other programs at the same time or the network is busy. Therefore, the different time behaviors can be investigated by simulating ESTSs with different execution durations. To get a meaningful information about the influence of different time behaviors the models have to simulate with the same inputs and the same initialization. In this example a system with three models (Figure 7.9) are simulated with different simulators and different time behaviors.



(a)Sender (b)Network (c)Receiver

Figure 7.9.: ESTSs for time behavior example.

A `sender` receives user inputs and sends the inputs over a `network` to the `receiver`. The receiver is able to process 2 inputs which have to be received within 100 time units. After receiving 2 inputs the receiver sends `full` over the network back to the sender. A timing group in the receiver is responsible to change after 150 time units in the `timeout` state to send a `timeout` to the sender. The transitions of the sender and the receiver takes 10 time unit and the transitions of the network can take between 1 and 25 time units. The aim of this example is to show the influence of different workloads of the network to the whole system.

The ESTSs in Figure 7.9 are simulated four times where they are simulated twice with the deterministic simulator and the single ESTSs with the continuous simulator. The other two simulations are made with the full parallel simulator and the single ESTSs are simulated with the single step simulator. Every multiple ESTS simulator is executed once with the setup to use the minimum execution duration and once to use the maximum execution duration. This effects the execution duration of the network where either 1 or 25 time units are used. The environment sent an input for the external input transition `inputS` three times at simulation time 1, 2 and 3.

In the Figures A.1 and A.2 the generated simulation tree from the deterministic simulator are shown. As mentioned before the single ESTSs are simulated with the continuous simulator. The first simulation is done using the minimum execution duration. The sender starts with the reception of the input and sends it to the network. The network received and sent the input to the receiver. Due to the minimum execution duration this only needed 2 time units. The receiver processed the input by increasing the attribute `x` by 1. Afterward, the sender got its second turn and received the second input from the environment and sent it via the network to the receiver. The receiver again processed the input and due to the attribute `x` reached the maximum value of 2, `fullN` was sent to the sender. Before the network can forwarding, the message from the receiver the sender got its third input which had no effect because the receiver is `full`.

The result of the simulation using the maximum execution duration is quite different than the previous one. This execution tree is shown in Figure A.2 which shows three execution paths. The begin of the execution tree is similar to A.1 the only difference is the used execution duration for the transitions of the network. The network took much more time and therefore the receiver got its second turn at simulation time 170. Due to the delay is less than the simulation time the delay transition became enabled. Besides the delay transition the input transition `inputR` is also enabled leading to non-determinism. In one execution path the receiver sent a timeout to the sender. The other execution path splits again because the delay transition became enabled at state `processInput`. Thus, the execution path which executes the delay transition sent a timeout to the sender and the other one sent a full message.

These two execution trees show the impact of the time to a system. In the case the network is not busy the inputs are received before the timeout occurs. Though the network is busy this is not necessarily the case.

This system was also simulated with a full parallel simulator which used single step simulators for the single ESTSs. The resulting execution trees are shown in the Figures A.3 and A.4. Due to the parallel execution of transitions the inputs are always received before the timeout occurs. The sender and receiver are able to execute transitions while the network is busy. Therefore, the inputs are earlier processed and the receiver got the two inputs before the delay transitions became enabled.

7.4. ICOS Air Controller

This example illustrates the functionality of the implemented ICOS interface by a co-simulation of Simulink® models and an ESTS. In Figure 7.10 a Simulink® model is shown which includes three different main models: the Switch Signal, the Air Controller and the Physical Plant. The Switch Signal is able to turn the Air Controller on and off. The clock signal does not influence this example and therefore it is ignored. The Air Controller includes a nested State Machine, which is shown in Figure B.1. It consists of two fans which are activated if specific temperatures are reached. The inputs for the Air Controller are the switch (on or off) and the temperature of the Physical Plant. The Physical Plant needs the airflow of the fans for cooling.

The idea is to replace the Air Controller through an ESTS and ICOS co-simulates the Switch Signal, Physical Plant and the created ESTS. Therefore, the Switch Signal and Physical Plant are adapted to work with ICOS. Due to a State Machine can be transferred to an ESTS an equivalent ESTS for the Air Controller Simulink® State Machine was created, which is shown in Figure B.2. ICOS sends both inputs (switch and temperature) always at the same time and the ESTS has to send the airflow back within the simulation time of the duration. Therefore, an `airflow` output transition will be sent also when the Air Controller is turned off, but in this case it is always zero. Is the Air Controller turned on the value of the airflow depends on the received temperature.

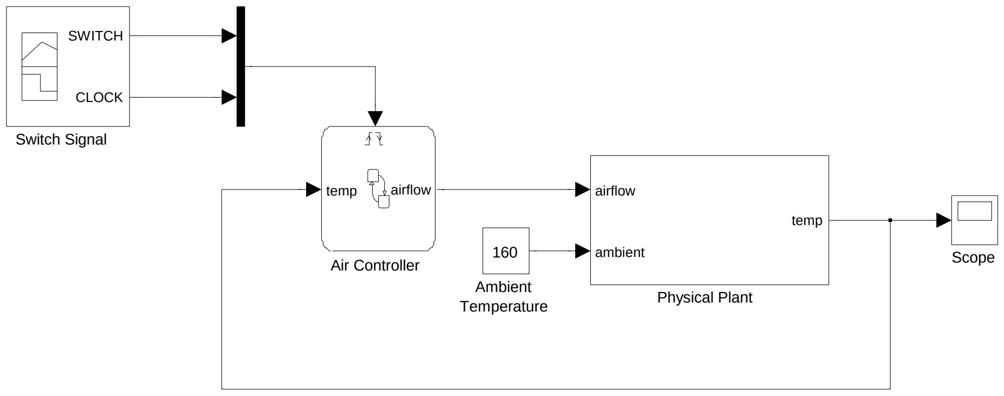
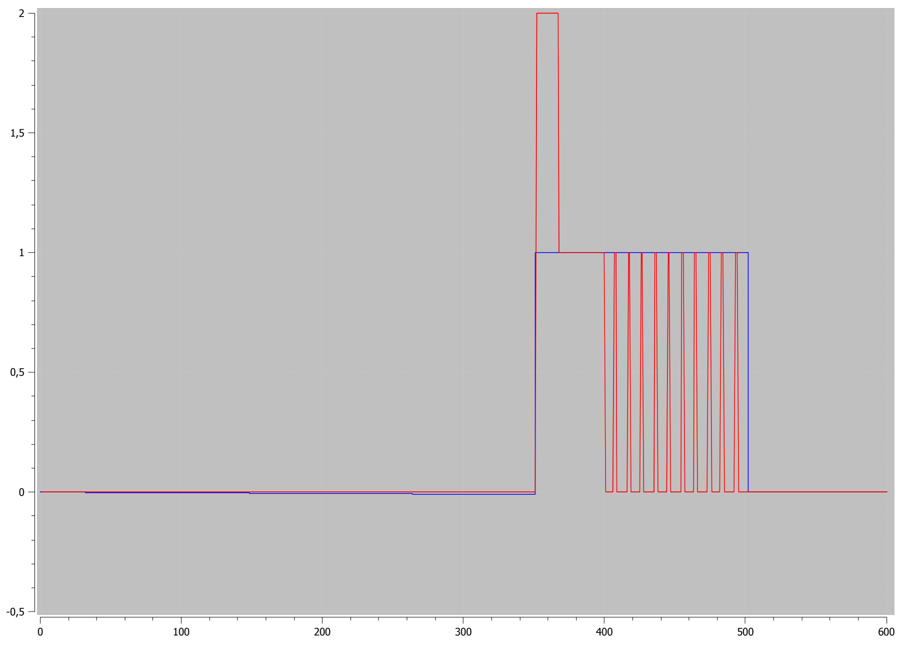


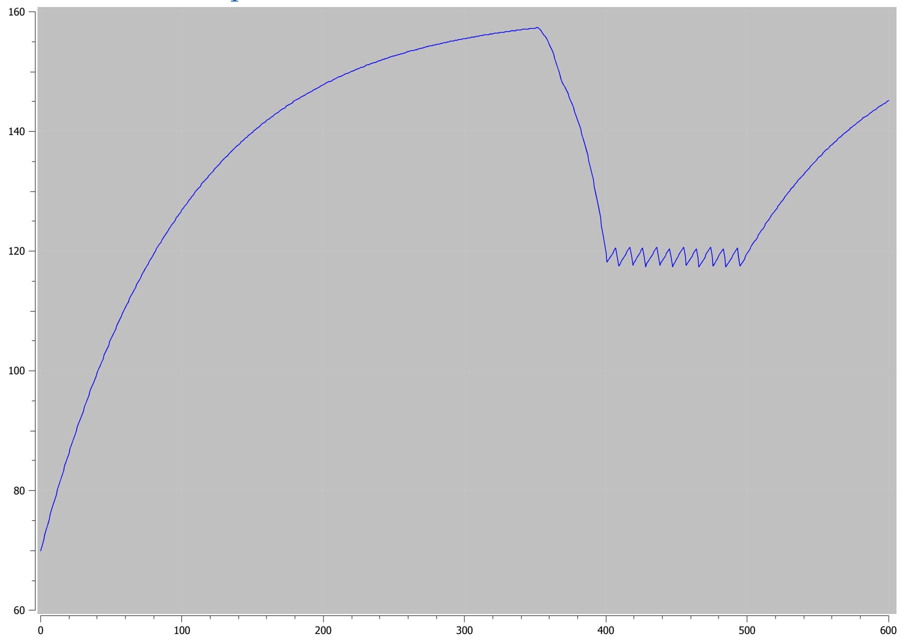
Figure 7.10.: Simulink® model of the air controller.

ICOS co-simulates the Simulink® models and the ESTS for 600 seconds. The plots shown in Figure 7.11 are the result of the co-simulation. The Switch Signal sends the values 1 and 0 to turn the Air controller on and off. In Figure 7.11a the sent switch signal is plotted where the Air Controller is only turned on in the time between 350 and 500. The temperature of the Physical Plant increases while the Air Controller is turned off which can be seen in Figure 7.11b. At second 350 it has a temperature of about 158°C at this time the Air Controller was turned on and both fans started to work. The temperature decreased under 150°C and therefore one fan stopped working. After 50 seconds, the second fan also stopped its operation because the temperature is under 120°C. The next 100 seconds the temperature reached the 120°C bound a few times and therefore one fan was stared always for a short time. The Switch Signal turned the Air Controller off at second 500 and this led to an increasing of the temperature until the simulation was stopped.

The results of the ICOS co-simulation are basically equal in comparison with the Simulink® simulation, which is shown in Figure B.3. The Simulink® simulation used the Clock signal of the Switch Signal, which is 2Hz. The ICOS simulation was made with a duration of 1s and therefore the results are not exactly equal.



(a)Plot of the input switch (blue) and the calculated airflow (red).



(b)Plot of the temperature of the Physical Plant.

Figure 7.11.: ICOS plots of the co-simulation of Simulink® and ESTS.

8. Final Remarks

8.1. Related Work

The simulation of transition systems is used in a lot of scientific tools in particular for automatic generation of test cases. There are some tools using Labeled Transition Systems (LTSs) [24] like TGV [17, 16] and TorX [3, 25]. These tools are generating test cases from formal specifications of reactive systems. TGV generates tests cases for a System Under Test (SUT) with respect to a given test purpose where the system and the test purpose are given as LTSs. TorX takes a different approach than TGV by generating test cases on-the-fly, where the random generated inputs are executed on the SUT and the outputs are checked immediately.

Modeling large systems with LTS is impractical and therefore specification languages are used. One of these is LOTOS [15] which is an International Organization for Standardization (ISO) standard and it is a practical way to specify a LTS. However, it is often a laborious task to create such a model because of the treatment of data. In a LTS there exists no variables and parameter so these names and values have to encode in the state and action names. This intensifies the state space to state space explosion and limits the usability.

The tool STG [7] uses a symbolic version of transition systems the Input-Output Symbolic Transition System (IOSTS). Symbolic means that data are separated from the model where the data are stored in variables. The process of the STG is similar to that from TGV. A tool that is using STS [11, 12] for on-the-fly test case generation like TorX is the STSimulator. This simulator is used in Jambition [10] which generates test cases automatically for web applications.

The approaches above are not considering time and for this reason the models have to be improved for time-critical systems. One of the first approaches dealing with real-time was the Timed Automata (TA) [1] which is the base for many further approaches. There are a lot of real-time extensions which define timed versions of LTS [18, 6, 21]. The tool TorX was extended to support a timed version of a LTS [5]. The combination of time and a STS was defined in [27].

They described the Symbolic Timed Automata (STA) which is an amalgamation of a STS and a TA.

A related and much older approach are the Finite State Machines (FSMs) [13] which consists of states and transitions. The main difference in comparison with transition systems is that a transition consists of input and output. The explosion of the state space is a problem for LTS as well as for FSM. Therefore, a symbolic approach the Extended Finite State Machine (EFSM) was introduced. There are also approaches to handle real-time systems based on FSM variants like Timed Finite State Machines (TFSM) [9] and Timed Extended Finite State Machines (TEFSM) [20]. The simulation of FSMs is also done for example for verification [14].

Uppaal [4, 2] is a model checker for real-time systems which is developed by Uppsala University and Aalborg University. Systems can be modeled, validated and verified with this tool. The base of the underlying model is a TA which is extended with data types.

8.2. Future Work

The presented ESTS simulators generate execution traces with respect to the given inputs. An input holds for every parameter exactly one value. An extension of this approach could be extent the parameter valuation to a parameter valuation range. Therefore, a simulation can generate the execution traces for the value range. This would lead to the possibility to create test cases for boundary checks. Furthermore, test cases could be generated based on the execution traces using for example constraint solver.

8.3. Conclusion

A system like a CPS has versatile components which communicate in different ways with each other. Therefore, ESTSs can be simulated with three different communication modes namely Deterministic, Full Interleaving and Full Parallel. This communication modes consist of sequential execution of the simulation ESTSs to simultaneously in parallel execution. The ability to use different communication modes for one simulation underlies the possibility of nesting ESTSs in a hierarchy in a tree like structure.

The simulation executors send inputs to a simulator and receive the outputs. The origin of the sent inputs is in every simulation executor different. The Random Walk generates inputs randomly by using a constraint solver. Precalculated inputs are used in Linear Walk where the outputs of different simulations with different options can be compared for the same inputs. The Manual Walk works like a wrapper and gets the inputs from an external program. This allows an interaction between an external program and the simulator.

The intention to create an embedded control unit for an existing system of physical components can be supported by the ESTS simulator. The provided communication modes and the considering of time fulfill the requirements to simulate such systems.

An interface to the co-simulation framework ICOS was introduced. ICOS is able to couple different simulation tools to co-simulate a system with subsystems of different domains. This enables the ESTS simulator to be a part of a simulation of for example a virtual vehicle.

Acknowledgments

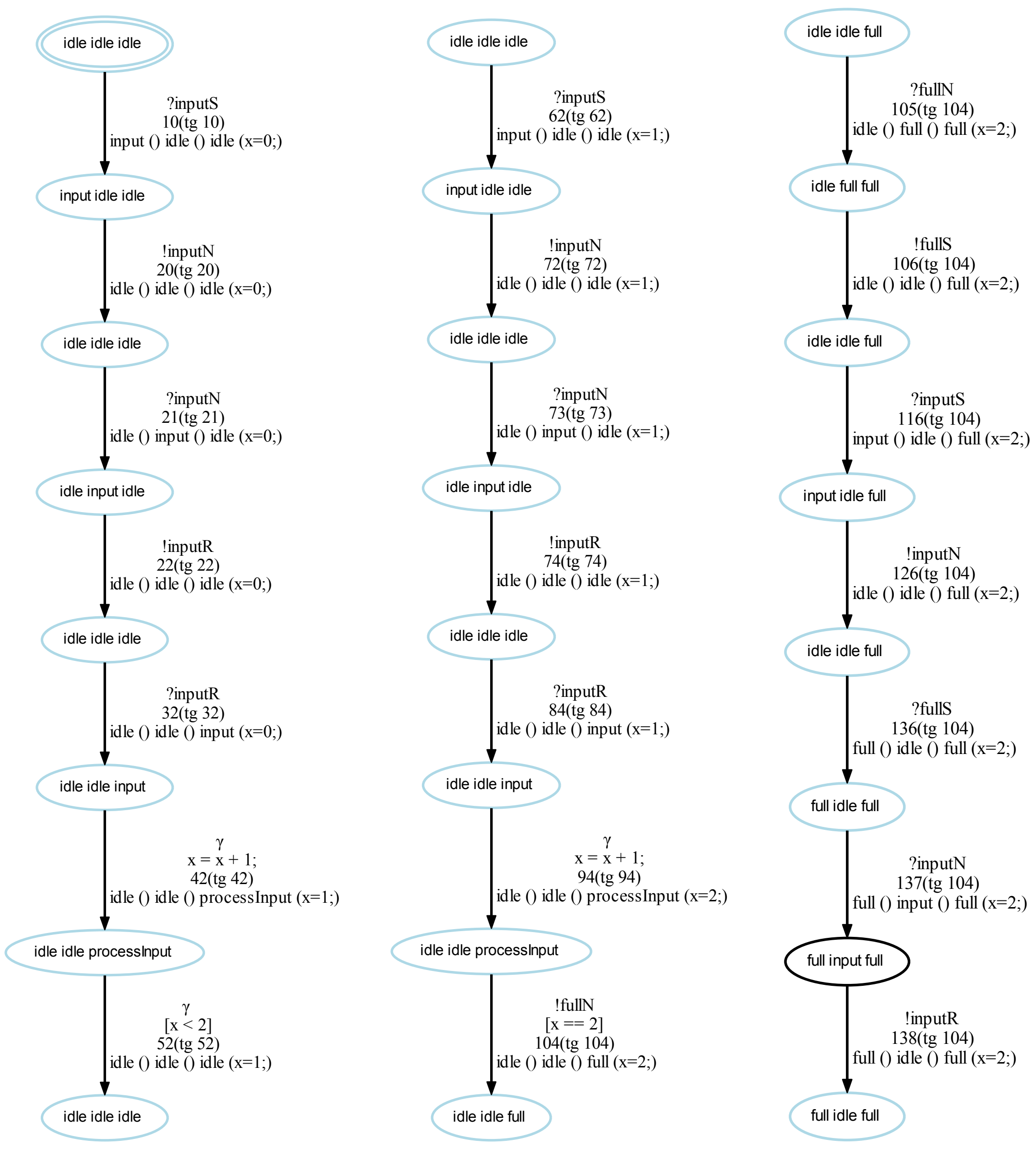
The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 269335 (see Article II.9. of the JU Grant Agreement) and from the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ) and the Austrian Research Promotion Agency (FFG).

I would like to express my gratitude to my supervisors Franz Wotawa and Christian Schwarzl for the useful comments and engagement through the entire process of this master thesis. Furthermore, I would like to thank my family for supporting me and their love.

Appendix A.

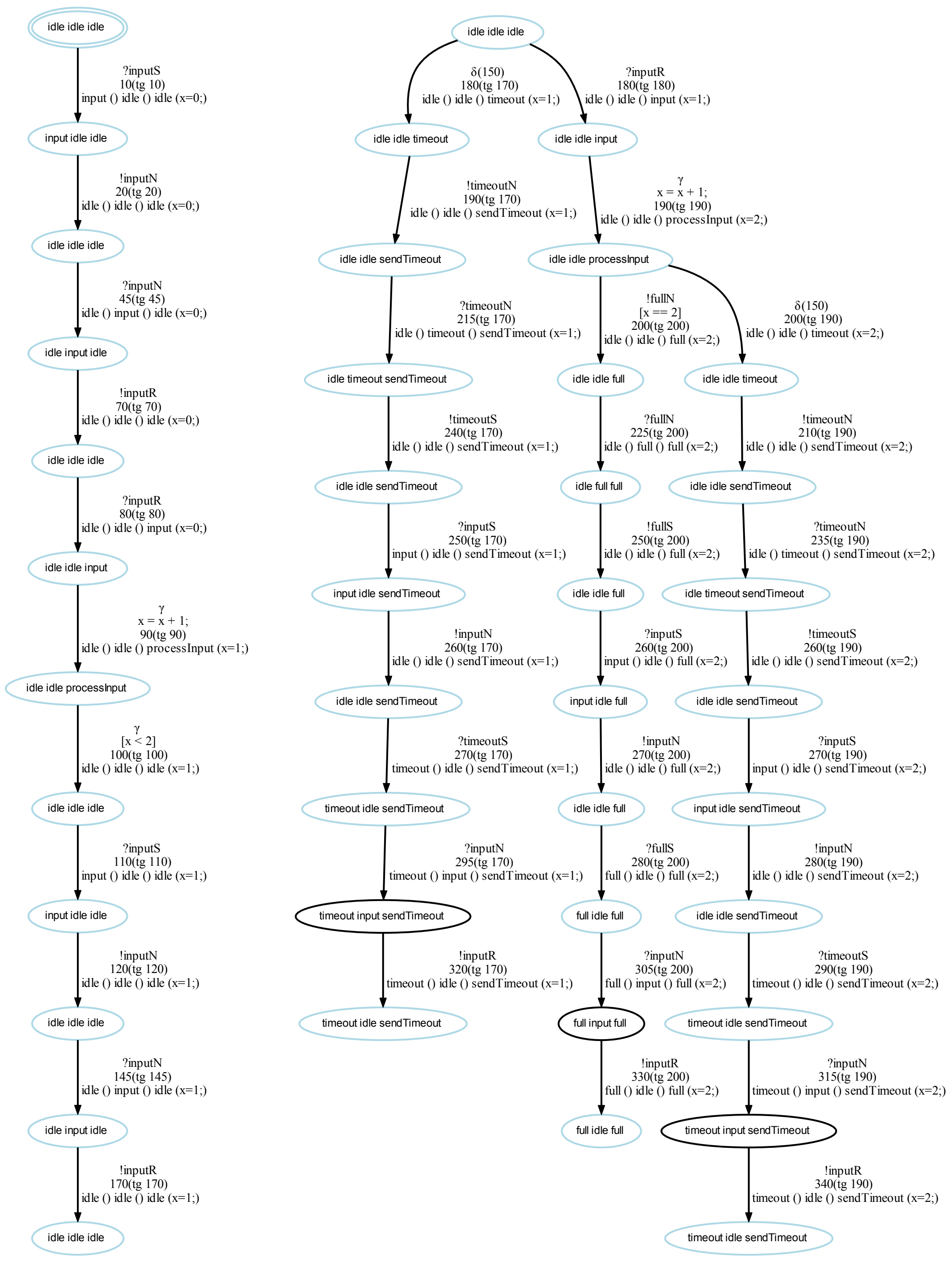
Network Example

The execution trees which are described in Section 7.3 are shown in this appendix.



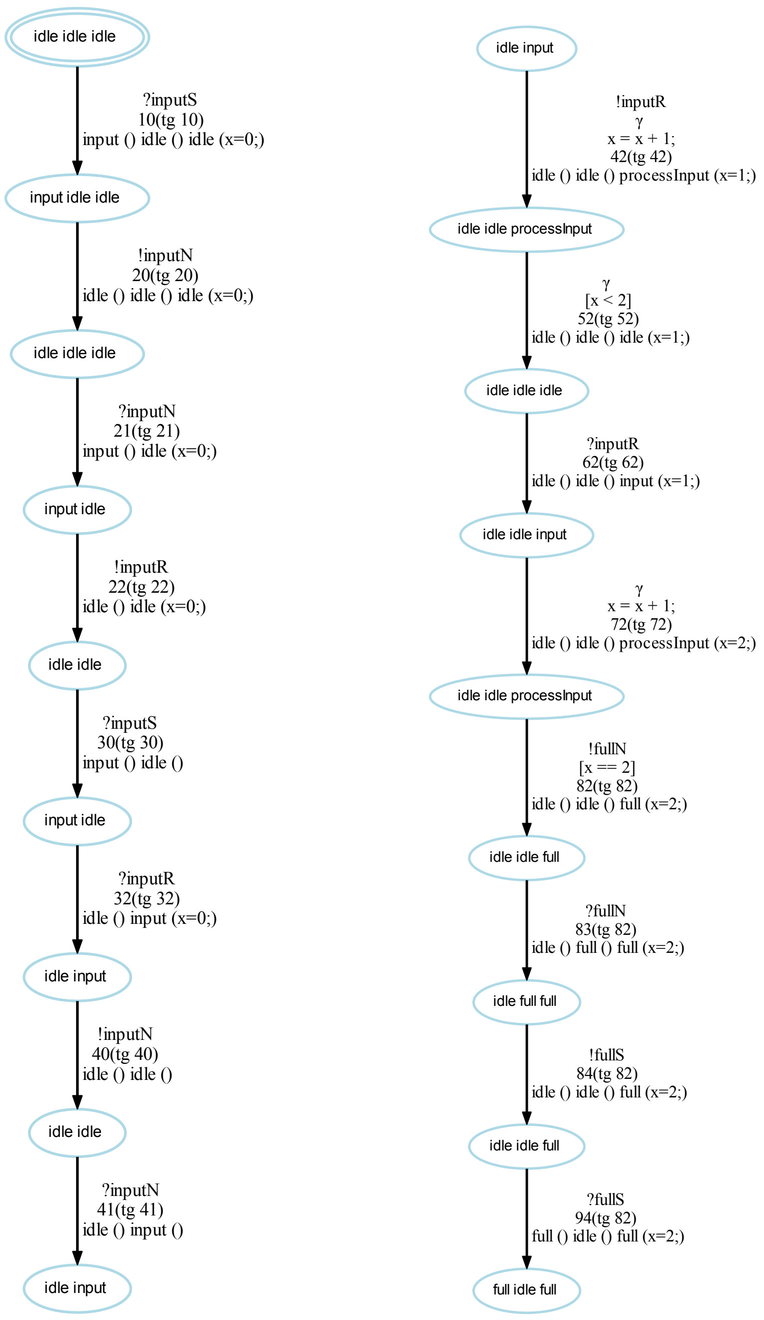
(a)Part 1 (b)Part 2 (c)Part 3

Figure A.1.: Execution tree for minimum execution duration and deterministic simulator.


idle idle idle
?inputS 10(tg 10) input () idle () idle (x=0;)
input idle idle
!inputN 20(tg 20) idle () idle () idle (x=0;)
idle idle idle
?inputN 45(tg 45) idle () input () idle (x=0;)
idle input idle
!inputR 70(tg 70) idle () idle () idle (x=0;)
idle idle idle
?inputR 80(tg 80) idle () idle () input (x=0;)
idle idle input
γ x = x + 1; 90(tg 90) idle () idle () processInput (x=1;)
idle idle processInput
γ [x < 2] 100(tg 100) idle () idle () idle (x=1;)
idle idle idle
?inputS 110(tg 110) input () idle () idle (x=1;)
input idle idle
!inputN 120(tg 120) idle () idle () idle (x=1;)
idle idle idle
?inputN 145(tg 145) idle () input () idle (x=1;)
idle input idle
!inputR 170(tg 170) idle () idle () idle (x=1;)
idle idle idle
idle idle idle
δ(150) 180(tg 170) idle () idle () timeout (x=1;)
?inputR 180(tg 180) idle () idle () input (x=1;)
idle idle timeout
idle idle input
!timeoutN 190(tg 170) idle () idle () sendTimeout (x=1;)
γ x = x + 1; 190(tg 190) idle () idle () processInput (x=2;)
idle idle sendTimeout
idle idle processInput
?timeoutN 215(tg 170) idle () timeout () sendTimeout (x=1;)
!fullN [x == 2] 200(tg 200) idle () idle () full (x=2;)
δ(150) 200(tg 190) idle () idle () timeout (x=2;)
idle timeout sendTimeout
idle idle full
idle idle timeout
!timeoutS 240(tg 170) idle () idle () sendTimeout (x=1;)
?fullN 225(tg 200) idle () full () full (x=2;)
!timeoutN 210(tg 190) idle () idle () sendTimeout (x=2;)
idle idle sendTimeout
idle full full
idle idle sendTimeout
?inputS 250(tg 170) input () idle () sendTimeout (x=1;)
!fullS 250(tg 200) idle () idle () full (x=2;)
?timeoutN 235(tg 190) idle () timeout () sendTimeout (x=2;)
input idle sendTimeout
idle idle full
idle timeout sendTimeout
!inputN 260(tg 170) idle () idle () sendTimeout (x=1;)
?inputS 260(tg 200) input () idle () full (x=2;)
!timeoutS 260(tg 190) idle () idle () sendTimeout (x=2;)
idle idle sendTimeout
input idle full
idle idle sendTimeout
?timeoutS 270(tg 170) timeout () idle () sendTimeout (x=1;)
!inputN 270(tg 200) idle () idle () full (x=2;)
?inputS 270(tg 190) input () idle () sendTimeout (x=2;)
timeout idle sendTimeout
idle idle full
input idle sendTimeout
?inputN 295(tg 170) timeout () input () sendTimeout (x=1;)
?fullS 280(tg 200) full () idle () full (x=2;)
!inputN 280(tg 190) idle () idle () sendTimeout (x=2;)
timeout input sendTimeout
full idle full
idle idle sendTimeout
!inputR 320(tg 170) timeout () idle () sendTimeout (x=1;)
?inputN 305(tg 200) full () input () full (x=2;)
?timeoutS 290(tg 190) timeout () idle () sendTimeout (x=2;)
timeout idle sendTimeout
full input full
timeout idle sendTimeout
!inputR 330(tg 200) full () idle () full (x=2;)
?inputN 315(tg 190) timeout () input () sendTimeout (x=2;)
full idle full
timeout input sendTimeout
!inputR 340(tg 190) timeout () idle () sendTimeout (x=2;)
timeout idle sendTimeout

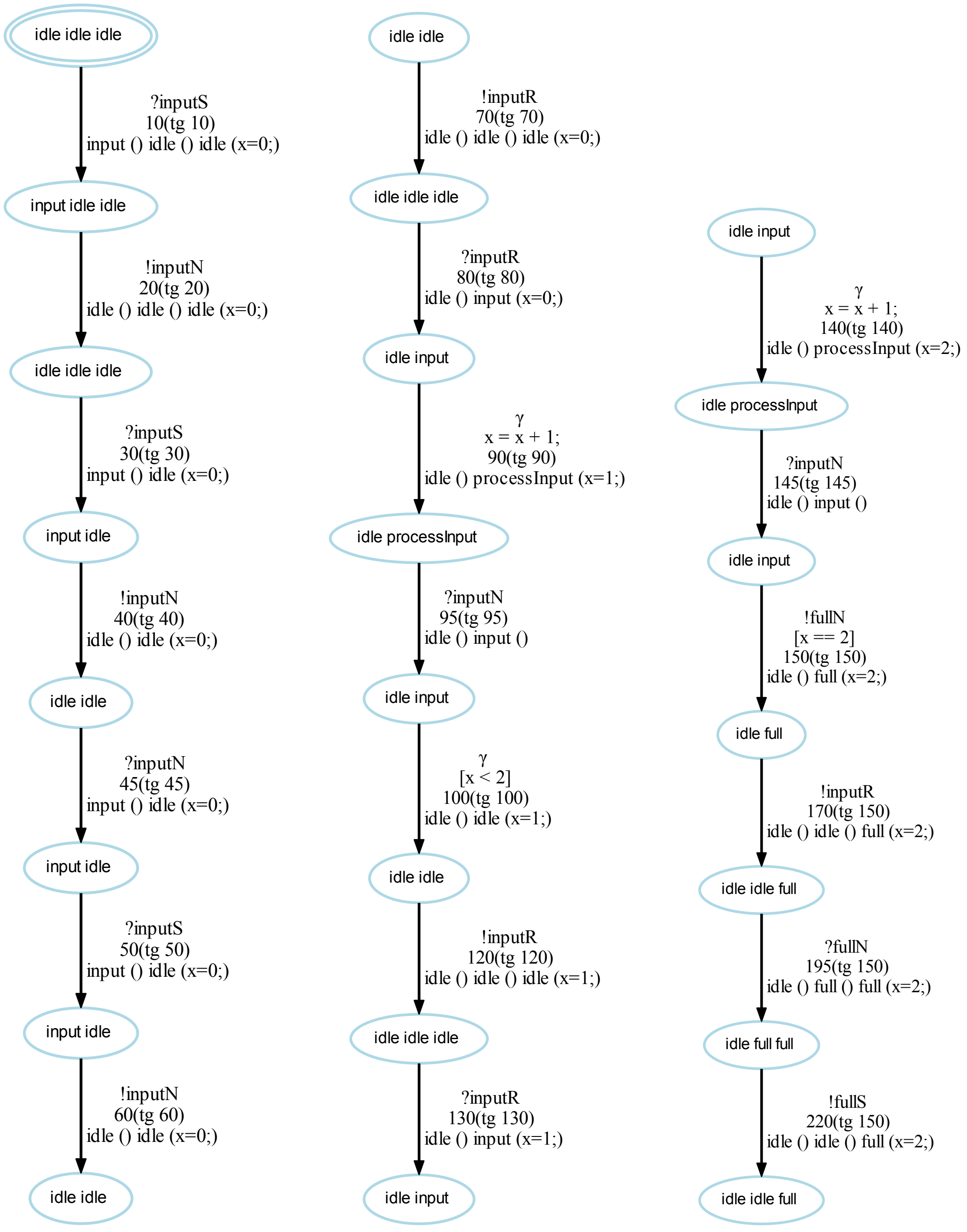
(a)Part 1 (b)Part 2

Figure A.2.: Execution tree for maximum execution duration and deterministic simulator.



(a)Part 1 (b)Part 2

Figure A.3.: Execution tree for minimum execution duration and full parallel simulator.



(a)Part 1 (b)Part 2 (c)Part 3

Figure A.4.: Execution tree for maximum execution duration and full parallel simulator.

Appendix B.

Air Controller Example

The Simulink® State Machine and the equivalent ESTS of the air controller are shown here. Also, the reference result of the Simulink® simulation is displayed in this section.

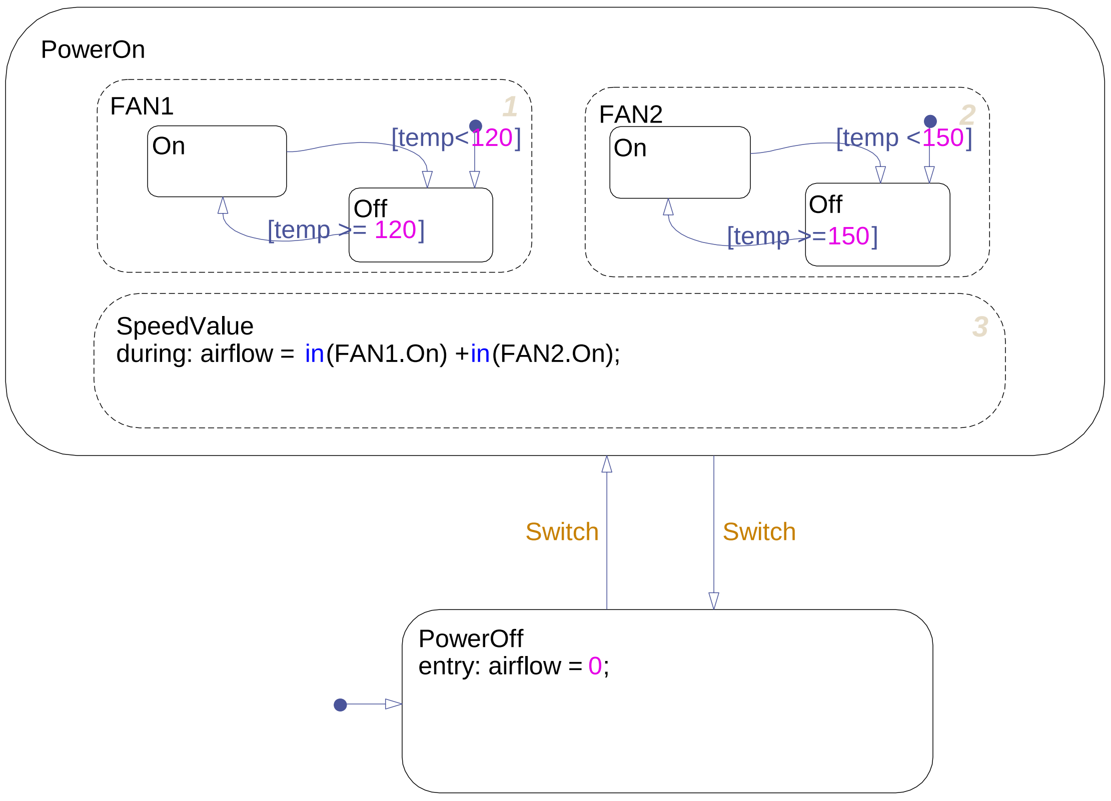


Figure B.1.: Simulink® State Machine of the air controller.

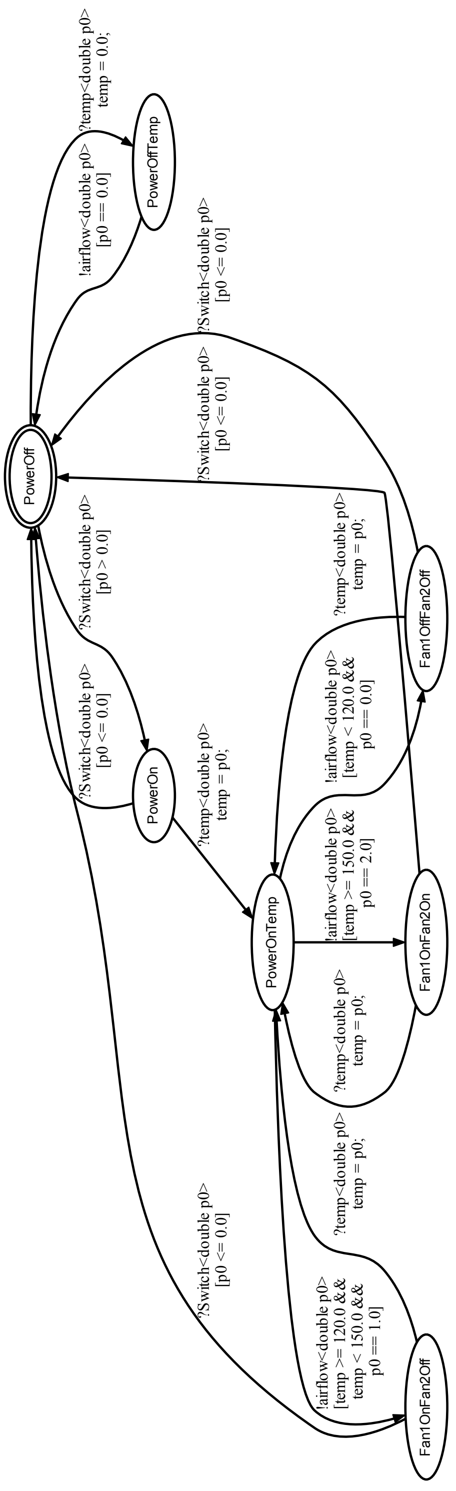

PowerOff
PowerOffTemp
PowerOn
PowerOnTemp
Fan1OffFan2Off
Fan1OnFan2On
Fan1OnFan2Off
?temp<double p0> temp = 0.0;
!airflow<double p0> [p0 == 0.0]
?Switch<double p0> [p0 <= 0.0]
?Switch<double p0> [p0 <= 0.0]
?Switch<double p0> [p0 > 0.0]
?Switch<double p0> [p0 <= 0.0]
?temp<double p0> temp = p0;
?temp<double p0> temp = p0;
!airflow<double p0> [temp < 120.0 && p0 == 0.0]
!airflow<double p0> [temp >= 150.0 && p0 == 2.0]
?temp<double p0> temp = p0;
?temp<double p0> temp = p0;
?Switch<double p0> [p0 <= 0.0]
!airflow<double p0> [temp >= 120.0 && temp < 150.0 && p0 == 1.0]

Figure B.2.: ESTS of the air controller.

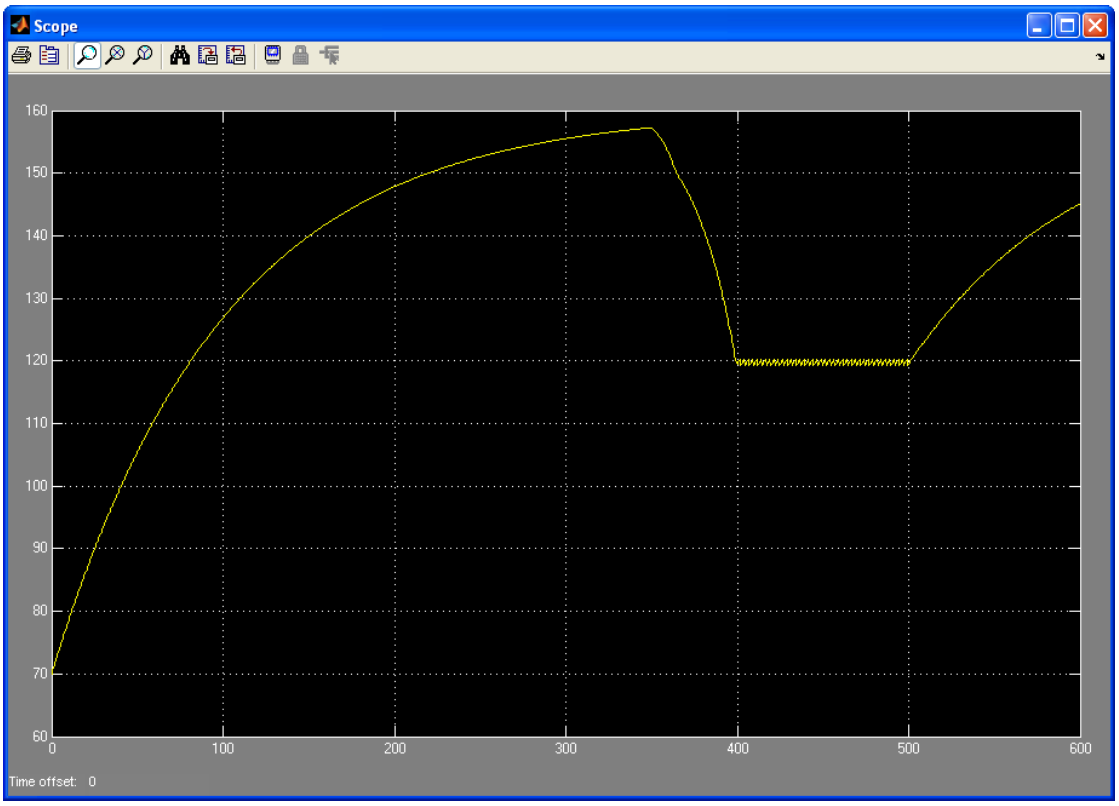


Figure B.3.: Simulink® plot of the Air Controller simulation.

List of Acronyms

CPS	Cyber-Physical System
EFSM	Extended Finite State Machine
ESTS	Extended Symbolic Transition System
FSM	Finite State Machine
GUI	Graphical User Interface
HiL	Hardware in the Loop
ICOS	Independent Co-Simulation
IOSTS	Input-Output Symbolic Transition System
ISO	International Organization for Standardization
JNI	Java Native Interface
JVM	Java Virtual Machine
LTS	Labeled Transition System
NSS	Nested Simulation State
STA	Symbolic Timed Automata
STS	Symbolic Transition System
SUT	System Under Test
TA	Timed Automata
TLTS	Timed Labeled Transition System
UML	Unified Modeling Language

Bibliography

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[2] G. Behrmann, R. David, and K. G. Larsen. A tutorial on UPPAAL. pages 200–236. Springer, 2004.

[3] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment, 1999.

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.

[5] H. Bohnenkamp and A. Belinfante. Timed testing with TorX. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2005.

[6] L. Brandán Briones. *Theories for model-based testing: real-time and coverage*. PhD thesis, Enschede, March 2007. CTIT number: 07-97.

[7] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer Berlin Heidelberg, 2002.

[8] M. Decker. Simulation of Extended Symbolic Transition Systems. 2013. Master-Project.

[9] K. El-Fakih, N. Yevtushenko, and H. Fouchal. Testing timed finite state machines with guaranteed fault coverage. In M. Núñez, P. Baker, and M. Merayo, editors, *Testing of Software and Communication Systems*, volume 5826 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin Heidelberg, 2009.

[10] L. Frantzen, M. las Nieves Huerta, Z. Kiss, and T. Wallet. On-the-fly model-based testing of web services with JAMBITION. In R. Bruni and K. Wolf, editors, *Web Services and Formal Methods*, volume 5387 of *Lecture Notes in Computer Science*, pages 143–157. Springer Berlin Heidelberg, 2009.

[11] L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[12] L. Frantzen, J. Tretmans, and T. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in Lecture Notes in Computer Science, pages 40–54. Springer, 2006.

[13] A. Gill. *Introduction to the Theory of Finite-state Machines*. New York: McGraw-Hill, 1962.

[14] Z. Hasan and M. Ciesielski. Functional verification and simulation of FSM networks. In *VLSI Test Symposium, 1993. Digest of Papers., Eleventh Annual 1993 IEEE*, pages 326–332, 1993.

[15] International Organization for Standardization. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989.

[16] C. Jard and T. Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, Aug. 2005.

[17] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 1999.

[18] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *In 11th International SPIN Workshop on Model Checking of Software (SPIN'04), volume 2989 of LNCS*, pages 109–126. Springer, 2004.

[19] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 2011. http://LeeSeshia.org/.

[20] M. G. Merayo, M. Núñez, and I. Rodríguez. Extending EFSMs to Specify and Test Timed Systems with Action Durations and Timeouts. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, volume 4229 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2006.

[21] J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In F. Cassez and C. Jard, editors, *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer Berlin Heidelberg, 2008.

[22] C. Schwarzl. *Symbolic Model-based Test Case Generation for Distributed Systems*. PhD thesis, Institute for Software Technology Graz University of Technology, 2012.

[23] C. Schwarzl, B. Aichernig, and F. Wotawa. Compositional Random Testing Using Extended Symbolic Transition Systems. In B. Wolff and F. Zaïdi, editors, *Testing Software and Systems*, volume 7019 of *Lecture Notes in Computer Science*, pages 179–194. Springer Berlin Heidelberg, 2011.

[24] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. Number TR-CTIT-96-26 in CTIT technical report series, Enschede, the Netherlands, 1996. University of Twente, Centre for Telematics and Information Technology (CTIT).

[25] J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[26] VIRTUAL VEHICLE Research Center. ICOS 3.0 User Manual, 2013.

[27] S. von Styp, H. Bohnenkamp, and J. Schmaltz. A conformance testing relation for symbolic timed automata. In K. Chatterjee and T. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *Lecture Notes in Computer Science*, pages 243–255. Springer Berlin Heidelberg, 2010.

[28] J. Zehetner, M. Benedikt, D. Watzenig, and J. Bernasch. Modern coupling strategies - is co-simulation controllable. In *NAFEMS Online-Magazin*. 2012.