A.2 Appendix

Algorithm 2 iterates over `States` belonging to the `System` in the ontology (using the "hasState" property) (Line 3). It extracts the related `StateValues` (Line 4), adds them to the corresponding component's implementation (Line 5), and records them in a state list (Line 6).

Algorithm 2 Transformation: Process States

1: **function** *processStates*(M, s, s_{impl})
2: $state\_list \leftarrow []$
3: **for** each *state* in $O(State)$, where $s.hasState(state)$ **do**
4: $stateValues \leftarrow$ getStateValues($state$)
5: addStates(M, s_{impl}, $stateValues$)
6: state_list.add($stateValues$)
7: **end for**
8: **return** *state_list*
9: **end function**

In Algorithm 3, for a `System`'s component s the algorithm iterates over all `Functions` performed by the `System` s (Line 3), identifying them as out event ports (Line 4). If a `Function` *sets* or *sends* data `Items` (Line 5), data output ports are generated within s (Line 8). In cases where a `Function` *receives* `Items` (Line 11), data input ports are created (Line 14). For `Functions` that *emit* `Flows` (Line 17), out data boolean ports are generated (Line 19). If `Functions` *ingest* `Flows` (Line 22), out data ports of boolean type are generated (Line 24). Algorithm adds all identified ports to the component and records their default values for each case (Line 9, 15, 20 & 25). Finally, an out data enum port named "function_state" is introduced (Line 29), to track function-specific states within the component.

Algorithm 3 Transformation: Process Ports

1: **function** *processPorts*(M, s)
2: $functionList \leftarrow$ ["no_function"]
3: **for each** $function$ in $O(Function)$, where $s.performs(function)$ **do**
4: addOutEventPort(M, s, $function$)
5: **if** $function.sets(item)$ or $function.sends(item)$ **then**
6: $port \leftarrow$ getItem($function$)
7: $values \leftarrow$ getItemValues($port$)
8: $default \leftarrow$ addOutDataPort(M, s, $port$, $values$) ▷ if $values$ is null, port type is boolean, with default value "false"
9: $function$.setFunctionEffect($port$, $default$)
10: **end if**
11: **if** $function.receives(item)$ **then**
12: $port \leftarrow$ getItem($function$)
13: $values \leftarrow$ getItemValues($port$)
14: $default \leftarrow$ addInDataPort(M, s, $port$, $values$) ▷ if $values$ is null, port type is boolean, with default value "false"

```
15:             function.setFunctionEffect(port, default)
16:         end if
17:         if function.emits(flow) then
18:             port ← getFlow(function)
19:             default ← addOutDataPort(M, s, port, null)      ▷ Port type is
    boolean, with default value "false"
20:             function.setFunctionEffect(port, default)
21:         end if
22:         if function.ingests(flow) then
23:             port ← getFlow(function)
24:             default ← addInDataPort(M, s, port, null)       ▷ Port type is
    boolean, with default value "false"
25:             function.setFunctionEffect(port, default)
26:         end if
27:         functionList.add(function + "_" + s + "_st")
28:     end for
29:     function_state ← addOutDataPort(M, s, "function_state",
    functionList)
30:     s.setFunctionState(function, function_state)
31: end function
```

Algorithm 4 focuses on establishing connections between subcomponents within the `System`'s implementation. The algorithm iterates over each subcomponent (Line 2), identifies shared ports (i.e. ports with the same name) and invoked event ports (e.g. where the corresponding `Function` invokes another) (Line 3, in which function getSubcomponentConnections is a SPARQL rule identifying shared ports and invocation connections), and creates connections within the component implementation (Line 7), effectively mirroring the ontology's connection relationships.

Algorithm 4 Transformation: Process Connections

```
1: function processPorts(M, s_impl)
2:     for each subcomponent in s_impl do
3:         connections ← getSubcomponentConnections(subcomponent)
4:         for each connection in connections do
5:             fromPort ← getInPort(connection)
6:             toPort ← getOutPort(connection)
7:             addConnection(M, s_impl, fromPort, toPort)
8:         end for
9:     end for
10: end function
```

Algorithm 5 synthesizes the nominal behavior of a system, in the form of transitions that are added to the `System`'s implementation. For every `Function` performed by the `System`'s component s (Line 2), the algorithm examines whether the function is connected to any system errors (Lines 7 - 13). If no error association is found, the algorithm delves into relationships like "isAvailableIn",

"setsTo" and "performs" to generate conditions and effects for transition triggers (Lines 27-42). Example 6 and Figure 14 illustrate the described transformation up to this point to a model skeleton in the SLIM formal language.

Algorithm 5 Transformation: Process Nominal Transitions

```
1: function processStateTransitions(M, s, s_impl)
2:     for each f in O(Function), where s.performs(f) do
3:         Tuple{item, value} ← getItemChangeByFunctionError(f)
4:         if not isFunctionRelatedToError(f) then
5:             fromStates ← []
6:             itemStateValueList ← []
7:             for each v in O(StateValue), where f.isAvailableIn(v) do
8:                 if isStateValueOfSystem(v, s) then
9:                     fromStates.add(v)
10:                else
11:                    itemStateValueList.add(v)
12:                end if
13:            end for
14:            if not Tuple{item, value} is null then
15:                itemStateValueList.add(item + "! = " + value)
16:            end if
17:            condition ← itemStateValueList
18:            function_states ← ""
19:            invoking_functions ← getFunctionInvokingList(f)
20:            for each invoking_function in invoking_functions do
21:                if function_states.equals("") then
22:                    function_states ← "function_state=" + s.getFunctionState(invoking_function)
23:                else
24:                    function_states ← function_states + " and " + "function_state="
   + s.getFunctionState(invoking_function)
25:                end if
26:            end for
27:            condition.add("(" + function_states + ")")
28:            toState ← null
29:            effect ← getFunctionEffect(f)
30:            if f.setsTo(v) or f.sets(itemOrState) then
31:                if itemOrState in O(State) then
32:                    toState ← v
33:                else
34:                    effect ← itemOrState + ":=" + v
35:                end if
36:            end if
37:            if effect is null then
38:                effect ← "function_state=" +s.getFunctionState(f)
39:            else
40:                effect ← effect + " ; " + "function_state=" + s.getFunctionState(f)
```

```
41:             end if
42:             addTransitions(M, s_impl, fromStates, condition, f, toState, effect)
    ▷ If fromStates is null, function will add transitions for all system states. If
    toState is null, then value is equal to the applicable system state
43:         end if
44:     end for
45: end function
```

Algorithm 6 manages the identification and processing of system errors in the ontology. It first retrieves `Error` information associated with each `System` (Line 2). If `Error`s are present (Line 3), it creates error models and implementations within the SLIM component model, adding any identified `Error` states (Line 6). If `Fault` exists that activates the error (Line 10), then it is added as event (Line 14), otherwise a unique event gets generated by combining the error name with "_FM" (Line 12). Finally, transitions between the states are established based on error "enter" and "exit" relationships (Lines 16 - 23).

Algorithm 6 Transformation: Process System Errors

```
1: function processErrors(M, s)
2:     errors ← getErrors(s)
3:     if not errors is empty then
4:         e ← createError(M, s)
5:         e_impl ← createErrorImplementation(M, s)
6:         no_error ← addErrorState(M, e, "no_error")
7:         current_state ← no_error
8:         for each error in errors do
9:             eState ← addErrorState(M, e, error)
10:            fault ← getFaultThatActivates(error)
11:            if fault is null then
12:                eEvent ← addErrorEvent(M, e_impl, error + "_FM")
13:            else
14:                eEvent ← addErrorEvent(M, e_impl, error)
15:            end if
16:            addEventTransition(M, e_impl, current_state, eEvent, eState)
17:            if enters(s, error) then
18:                current_state ← errorState
19:            end if
20:            if exits(s, error) then
21:                addEventTransition(M, errorImpl, eState, "reset", no_error)
22:                current_state ← no_error
23:            end if
24:        end for
25:    end if
26: end function
```

Algorithm 7 thoroughly analyzes the impact of `Error`s to `System` behavior and constructs transition conditions and fault effects accordingly (Lines 34 &

35). Particular attention is given to transitions to and from a "failed" state (Lines 36 - 53), considering function prefixes for entering or exiting error states. Additionally, it addresses reset conditions and effects for various states within the `System`'s state list (Lines 44 - 47), thereby capturing how the `System` responds to `Error` occurrences, their consequences and how the `System` handles recovery from this `Error`s. Example 7 and Figure 15 illustrate the described transformation to a SLIM error model.

Algorithm 7 Transformation: Process Fault Injections and Recovery Transitions

```
1: function processErrorTransitions(M, s, s_impl, state_list)
2:     e ← getError(s)
3:     e_impl ← getErrorImplementation(s)
4:     if not e_impl is null then
5:         addErrorModel(M, s_impl, e_impl)
6:         if not state_list is empty then
7:             port ← addOutDataPort(M, s, "pre_failed_state", state_list)
8:         end if
9:         for each eState in e.getErrorStates() do
10:            Tuple{item, value} ← getItemChangeByError(eState)
11:            addFaultEffect(M, s_impl, eState, item, value)
12:            functionsList ← functionsRelatedToError(s, errorState)
13:            for each functions in functionsList do
14:                for each function in functions do
15:                    for each v in O(StateValue), where f.isAvailableIn(v)
   do
16:                        if not isStateValueOfSystem(v, s) then
17:                            itemStateValueList.add(v)
18:                        end if
19:                    end for
20:                    if not Tuple{item, value} is null then
21:                        itemStateValueList.add(item + "! = " + value)
22:                    end if
23:                    condition ← itemStateValueList
24:                    reset_condition ← itemStateValueList
25:                    function_states ← ""
26:                    invoking_functions ← getFunctionInvokingList(function)
27:                    for each invoking_function in invoking_functions do
28:                        if function_states.equals("") then
29:                            function_states ← "function_state=" + s.getFunctionState(invoking_function)
30:                        else
31:                            function_states ← function_states + " and " +
   "function_state=" + s.getFunctionState(invoking_function)
32:                        end if
33:                    end for
34:                    condition.add("(" + function_states + ")")
```

```
35:                     effect ← getFunctionEffect(f)
36:                     if functionIsPrefixToExit(function, eState) then
37:                         if effect is null then
38:                             effect ← "function_state=" +s.getFunctionState(function)
39:                         else
40:                             effect ← effect + " ; " + "function_state=" + s.getFunctionState(function)
41:                         end if
42:                         addTransition(M, s_impl, "failed", condition, function,
   "failed", effect)
43:                         reset_condition.add("function_state=" +s.getFunctionState(function))
44:                         for each state in state_list do
45:                             reset_effect ← composeResetEffect(Tuple{item, value},
   state)
46:                             addTransition(M, s_impl, "failed", reset_condition,
   "reset", state, reset_effect)
47:                         end for
48:                     else if functionIsPrefixToEnter(function, eState) then
49:                         for each state in state_list do
50:                             errorEffect ← composeErrorEffect(effect, function,
   state)
51:                             addTransition(M, s_impl, state, condition, function,
   "failed", errorEffect)
52:                         end for
53:                     end if
54:                 end for
55:             end for
56:         end for
57:     end if
58: end function
```