Contents

1 Introduction 3

2 Basics of the SAML Language 6

3 Example Models 8
- 3.1 Vending Machine 8
- 3.2 R/S-Flipflop 9

4 Base Language Specification 11
- 4.1 Blocks 11
- 4.2 Instances 13
- 4.3 Wires 14
- 4.4 Initializing Cycles in the Instance-Wire-Graph 14

5 Translating Dataflow to SAML 16
- 5.1 Creating the Synchronization Component 16
- 5.2 Determining the Execution Order 16
- 5.3 Creating Components for Instances 17
- 5.4 Resolving Dataflow Input Qualifiers 18

6 Extension: Hierarchical Dataflow 20
- 6.1 Motivation 20
- 6.2 Grammar Modification 21
- 6.3 Transformation 22

7 Time Semantics 26
- 7.1 Dataflow Time Model 26
- 7.2 Spec Transformation 27

8 Analyzing Dataflow Models 29
- 8.1 Vending Machine 29
- 8.2 R/S Flipflop 30

9 Conclusion 32

Appendix A BNF Grammars 34
- A.1 Dataflow Grammar 34
- A.2 Hierarchical Extension Grammar 35

Appendix B Vending Machine Model 36
- B.1 Dataflow Model 36
- B.2 SAML Model 37

Appendix C Flipflop Model 39
C.1 Hierarchical Dataflow Model 39
C.2 Dataflow Model 40
C.3 SAML Model 41

Appendix D Implementation Notes 44

References 45

1 Introduction

The *System Analysis Modeling Language*[1] language was introduced by Matthias Güdemann as the core of a framework for model based safety analysis, that he developed in his dissertation. The unique feature of this framework, compared to other model based systems, is *SAML's* tool-independence. Instead of supplying an analysis software devoted to its own models, Güdemamn provided algorithms for the translation of *SAML* models into semantically equivalent *NuSMV* and *PRISM* models. This way both symbolic and probabilistic analyses of *SAML* models can be conducted [4, 9, 7].

Since then, *SAML* has been further evolved in the scope of the *VECS Project*[2] at Otto-von-Guericke University Magdeburg. On the one hand *SAML* has been enriched with many features, aiming to make the creation of complex models as efficient as possible. On the other hand, the modeling process still requires a large amount of expertise and care, because of the unfamiliar time semantics: In *SAML* all components of a model become active *at once* in every timestep. Most users are however accustomed to describing processes as *series of ordered steps*, similar to the description of algorithms.

This paper introduces an extension to SAML, which allows a more intuitive approach based on the *dataflow* design paradigm. The fundamental idea of *dataflow* is that technical systems can be represented as a series of interconnected components. Each of the components realizes a dedicated function, comparable to an electronic circuit. (Input-)values flow through the components in the order dictated by the network. They are repeatedly transformed, finally producing the system's output-values.

Other analysis frameworks that support dataflow-driven model creation already exist today. Particularly for safety critical applications, e.g. in aeronautics, the verification of the deployed systems is essential. *Esterel Scade* is a prominent representative of tools used in this field. It features a graphical editor for its dataflow-based modeling language, as well as formal model verification and a certified code generator[2]. Figure 1 shows an example of dataflow in *Scade's* graphical modeling language.

The aim of this work is the creation of a textual dataflow modeling language for the *VECS* framework. It should feature an intuitive syntax and expressive capabilities similar to *Scade's* graphical language. The dataflow extension is to be integrated into the *VECS* framework, by providing an algorithm for the translation of dataflow models to SAML models. These should then be processable by the existing toolchain, allowing the analysis with all available backends. Figure 2 gives an overview of the way the dataflow extension is aimed to be integrated into *VECS*.

---

[1] Abbreviated to SAML

[2] Verification Environment for Critical Systems

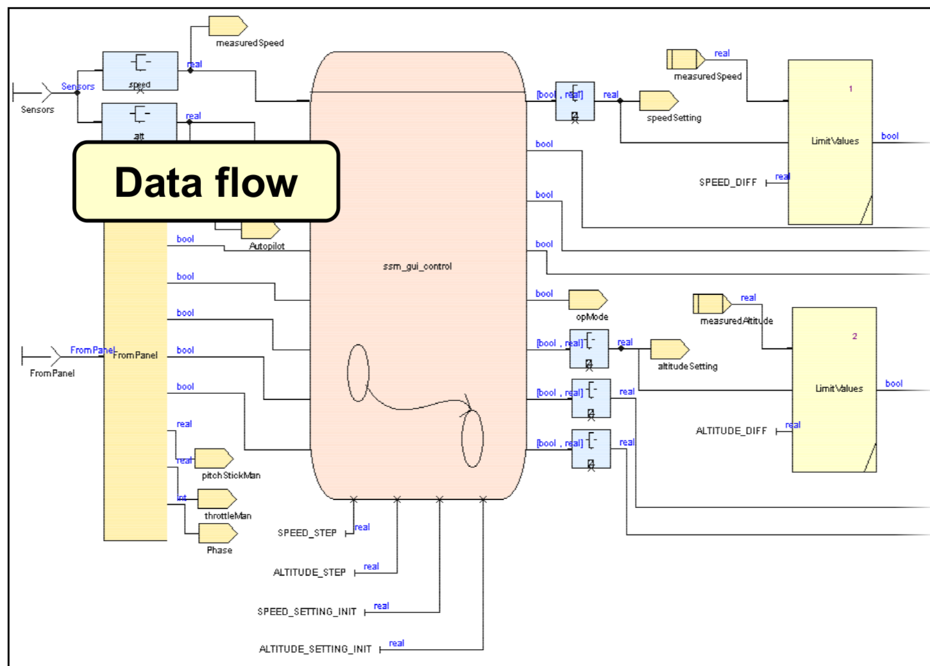

Figure 1: Dataflow in Scade's Graphical Modeling Language (Source [2])

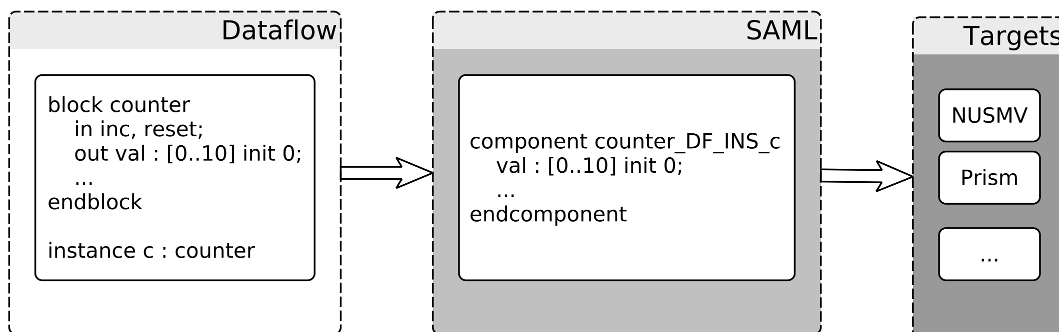

Figure 2: Integration of Dataflow into VECS

The paper is comprised of nine chapters. A basic understanding of the SAML language is required for the translation process. Therefore Chapter 2 presents SAML's essential syntax and semantics, followed by the introduction of two simple technical systems, which will be used as examples for the creation and translation of dataflow models in Chapters 4 and 5. In Chapter 6 a weakness of the dataflow language is highlighted, to motivate and explain the hierarchical-dataflow extension. Chapter 7 approaches the issues introduced by translating from dataflow's asynchronous to SAML's synchronous time semantics. Chapter 8 uses the models and algorithms introduced previously and demonstrates the workflow of analyzing dataflow models by use of the VECS framework. Finally, the Conclusion summarizes the paper, points out challenges and suggests additional extensions.

2 Basics of the SAML Language

This chapter introduces the subset of SAML, that will be used by the translation algorithm in Chapter 5: *components*, *declarations*, *update-rules* and *formulas*.

The *component* is the basic structure in any SAML model. Each component is a state-machine with at least one state-variable *declaration*. *Declarations* consist of the variable's identifier, the range of values it may roam and its initial value. Additionally each component requires a total transitionsrelation, comprised of a set of *update-rules*. In the simple, deterministic case, *update-rules* consist of a condition and assignments for the component's state variables. SAML can also express nondeterministic and probabilistic behavior, which will be covered at the end of the chapter. Three constraints need to be satisfied when writing *update-rules*:

1. Exactly one condition needs to be active in every timestep.
2. Assignments may only be made to variables declared in the same component. Other variables may be used in conditions, if they are qualified by prefixing their identifier with the declaring component's name, using a dot as separation token.
3. An assignment needs to be made to all variables declared in the component.

As stated before, all components become active at once in every timestep. When a component becomes active, the values of its state-variables change according to the update-rule, whose condition evaluates to true at that point of time. Listing 1 shows a component called *example*. It features two state variables, which are incremented in turns, until the first variable reaches its maximum value.

Listing 1: Basic SAML Syntax: Components

```
// Component Declaration
component example
    // State Variable Declarations
    a : [0..10] init 0;
    b : [0..10] init 0;

    // Update Rules
    (a <= b) & ((a < 10) & (b < 10)) -> choice: ((a' = a + 1) & (b' = b));
    (a > b)  & ((a < 10) & (b < 10)) -> choice: ((a' = a) & (b' = b + 1));
    !((a < 10) & (b < 10)) -> choice: ((a' = 0) & (b' = 0));
endcomponent
```

The first two conditions include the term $((a < 10) \wedge (b < 10))$, guarding the update rules from performing illegal assignments outside of the declared variable ranges. The third condition is the negated version of this term, defining the component's behavior, when the first variable reaches its upper boundary.

SAML has a syntax for the definition of *formulas*, which can be used to increase readability and reduce redundancy, when update-rules multiply contain the same term. Listing 2 shows the model, which was modified to use a *formula*, to avoid repeated use of the guard term.

Listing 2: Basic SAML Syntax: Formulas

```
// Component Declaration
component example
    // Formula Declaration
    formula RESET := !((a < 10) & (b < 10));

    // State Variable Declarations
    a : [0..10] init 0;
    b : [0..10] init 0;

    // Update Rules
    (a <= b) & !RESET -> choice: ((a' = a + 1) & (b' = b));
    (a > b)  & !RESET -> choice: ((a' = a) & (b' = b + 1));
    RESET -> choice: ((a' = 0) & (b' = 0));
endcomponent
```

Probabilistic Update-Rules assign different values to a component's state variables, based on a probability distribution. This is achieved by prefixing the assignment-sets $AS_1, AS_2, ..., AS_N$ with the probabilities $P_{AS_1}, P_{AS_2}, ..., P_{AS_N}$, with $\sum_{i=1}^{n} P_{AS_i} = 1$. Listing 3 illustrates the syntax for probabilistic update rules.

Listing 3: Basic SAML Syntax: Probabilistic Update Rules

```
condition -> choice: (0.5: ((var1' = a) & (var2' = b))
                    + 0.3: ((var1' = u) & (var2' = v))
                    + 0.2: ((var1' = x) & (var2' = y)));
```

Nondeterministic Update-Rules are used to assign different values to a component's state variables, when no probability distribution can be determined. Listing 4 illustrates the syntax for nondeterministic update rules.

Listing 4: Basic SAML Syntax: Nondeterministic Update Rules

```
condition -> choice: ((var1' = a) & (var2' = b))
           + choice: ((var1' = u) & (var2' = v))
           + choice: ((var1' = x) & (var2' = y));
```

Nondeterministic and probabilistic update rules may be combined to form hybrid update rules. In other words, these rules perform a nondeterministic selection between a set of probability distributions. This type of rule is rarely used due to the obscure semantics.

3 Example Models

In this chapter two example models used throughout this paper are introduced. The first is the simplified model of a vending machine, which will serve the purpose of introducing the dataflow syntax and of illustrating the translation to SAML in Chapters 4 and 5. The second is an r-s/flipflop, used in Chapter 6 to motivate the hierarchical-dataflow extension and to illustrate the transformation to classic dataflow. Both models will reappear as subject of the analyses in Chapter 8.

3.1 Vending Machine

Interaction with the simplified vending machine model can only take place in two ways: inserting coins or pressing a reset button to abort the purchase. When the target price of 2.50€ has been inserted, the machine grants a drink and returns to its initial state. The flowchart depicted in Figure 3 illustrates the mechanics of the vending machine.

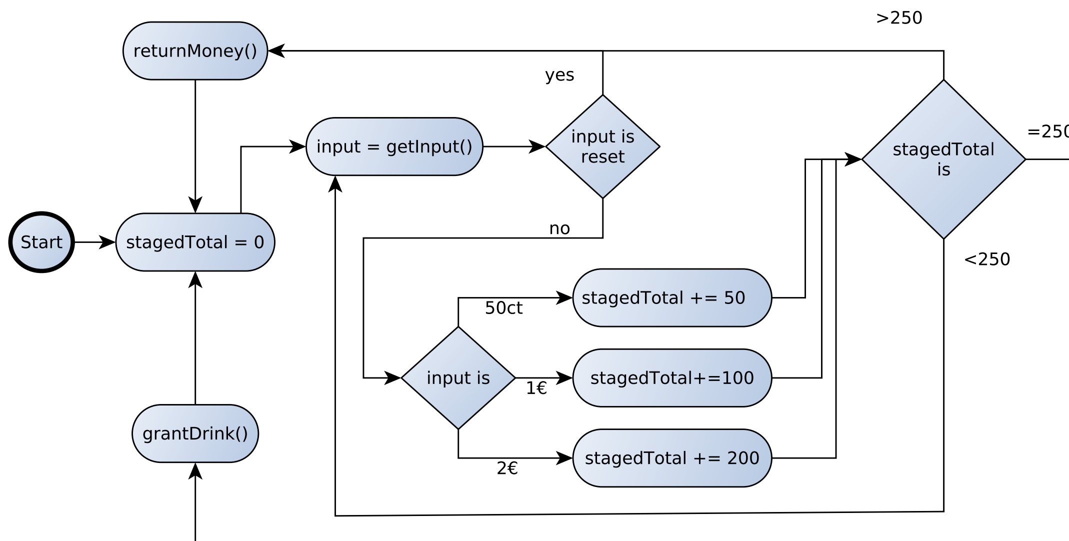

Figure 3: Flowchart: Mechanics of the Vending Machine

A Mealy Machine[5] is used to define the semantics in a more formal way. Its states encode the total money currently staged in the machine. Possible, mutually-exclusive inputs are *50* (50ct Coin), *100* (1€ Coin) and *200* (2€ Coin), as well as *r* (Reset). Possible, mutually-exclusive outputs are *No Operation* (0), *Give Drink* (1) and *Return Money* (2). Depending on the current state and selected input, exactly one transition is activated,

the associated output is made, and the following state becomes active [3]. As long as the target sum of 2.50€ is not reached, insertion of coins increases the staged total, producing the *No Operation* output. If the target sum is exactly reached, the machine returns to the initial state and produces the *Give Drink* output. If the target sum is exceeded, the machine returns to the initial state, producing the *Return Money* output. In any state, the *reset* input leads back to the initial state and yields the *Return Money* output. Figure 4 shows the graphical representation of the Mealy Machine.

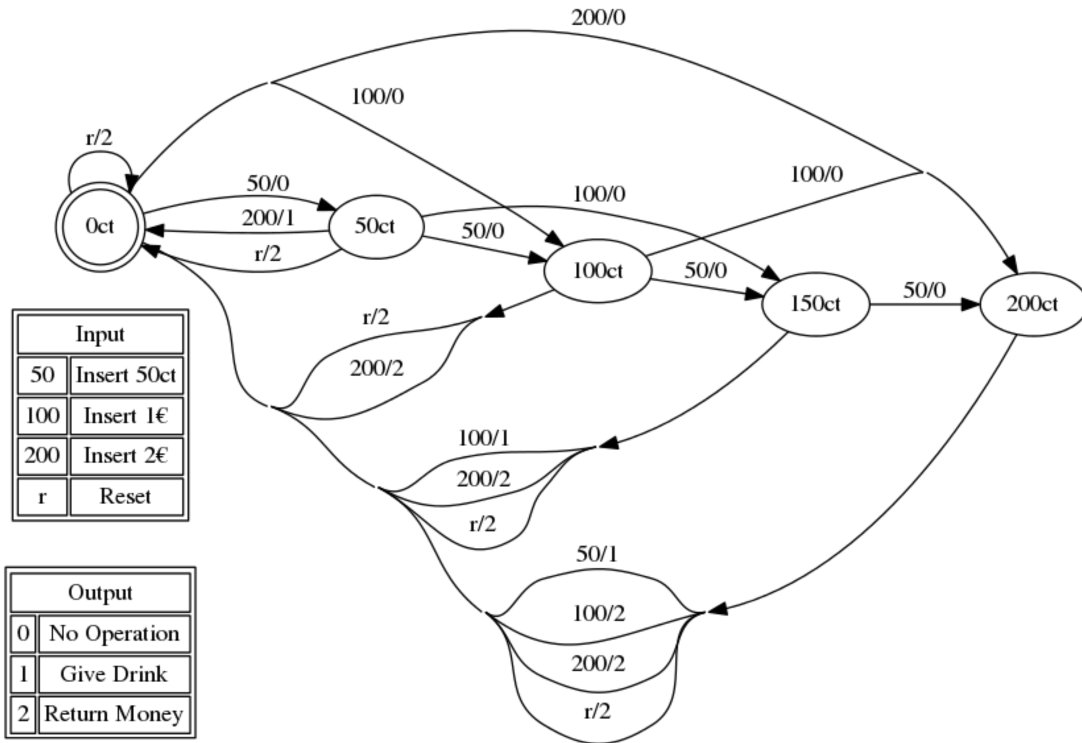

Figure 4: Mealy Machine: Semantics of the Vending Machine

3.2 R/S-Flipflop

In this section the structure and function of the most basic digital storage circuit are explained. The r/s-flipflop is a one-bit storage which can be built from two *nand gates*[4] in a feedback-system: The outputs of each of the gates are connected to the input of the opposite gate. The unconnected inputs of the two gates are called *set* and *reset*, their corresponding outputs are called $Q1$ and $Q2$. Figure 5 shows the schematic of the r/s-flipflop.

[3] The transitions are labeled in the format *input/output*

[4] Logical *and gate* with inverted output. Can be used to build any digital circuit.

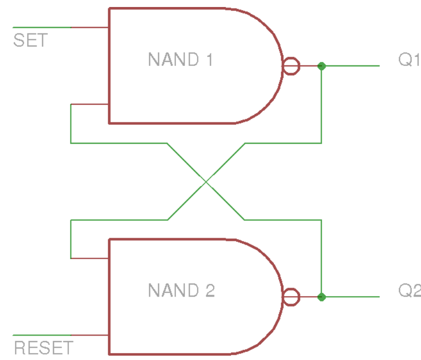

Figure 5: R/S-Flipflop: Schematic

The inputs of an r/s-flipflop built from nand gates are inverted. Therefore the levels of both inputs are *high* in the idle-state. In this state the current value of the *Q1* output is preserved until a falling edge on one of the inputs sets it to *high* (*set* input) or resets it to *low* (*reset* input). This makes an initial reset of the flipflop necessary, because otherwise the output level at startup would be undefined. The level of the $Q2$ output is always inverted to $Q1$, except for the invalid input configuration when both input levels are *low*. Figure 6 clarifies how the flipflop's output levels change according to its input configuration.

Set	Reset	Q1'	Description
0	0	1	invalid
0	1	0	set
1	0	1	reset
1	1	Q1	hold

Figure 6: R/S-Flipflop: Operation Table

4 Base Language Specification

In this chapter, the main elements of the dataflow language: *blocks*, *instances* and *wires* will be introduced by incrementally building the example vending-machine model from Chapter 3. The formal syntax definition of the dataflow language in BNF[5] as introduced by Saake and Sattler [10] is shown in Appendix A.1. In the following four sections the model will be incrementally built, by defining its (1) *blocks* and (2) *instances*, interconnecting the instances with (3) *wires* and (4) finding and initializing cycles in the instance-wire-graph. The complete model is also located in the Appendix B.1.

4.1 Blocks

The *block* is the central element of the dataflow specification language. *Blocks* relate to classes in OOP[6]. They define the interface and behavior of an arbitrary number of instances of their type[7].

Every block features an optional *input-*[8] and at least one *output-declaration*. Together these form the block's *interface*. Outputs require a range as well as an initial value. Apart from the interface, blocks require a total transitionsrelation represented as a set of update rules. The syntax and constraints for dataflow update rules are almost identical to those for SAML update rules described in Chapter 2. However, one further constraint needs to be satisfied: *Conditions may only depend on inputs and outputs declared in the block's interface.*

The dataflow model of the vending-machine is split into three blocks: *customer*, *stagingArea* and *actuator*.

The *customer* block simulates the environment of the vending machine. For this reason it has no inputs, but it does have two outputs called *coin* and *reset*. Its transitionsrelation represents a nondeterministic selection of either inserting one of the accepted coins, pressing the reset button, or doing nothing. Listing 5 shows the *customer* block.

[5] Backus Naur Form
[6] Object Oriented Programming
[7] In the following, the words *type* and *block* will be used synonymously
[8] Multiple inputs can be created in one declaration. Blocks without inputs are intentionally possible.

Listing 5: Example: Customer Block

```
block customer
        out coin : [0..3] init 0;
        out reset : [0..1] init 0;

        true -> choice : ((coin' = 0) & (reset' = 0))
                       + choice : ((coin' = 1) & (reset' = 0))
                       + choice : ((coin' = 2) & (reset' = 0))
                       + choice : ((coin' = 3) & (reset' = 0))
                       + choice : ((coin' = 0) & (reset' = 1));
endblock
```

The *stagingArea* block keeps track of the total of inserted money during each purchase process. It features the three inputs *coin*, *reset* and *reset2*. The first two correspond to the outputs of the *customer* block. The purpose of *reset2* is to signal an automatic reset after having granted a drink to the customer. Its first output *stagedTotal* is the actual memory, storing the amount of money inserted so far, encoded as a number of 50ct pieces. The second output, *returnMoney* indicates the vending-machine returning the staged money. The transitionsrelation is made up of ten update rules. The first seven cover the insertion of coins, either incrementing *stagedTotal* by the appropriate amount, or resetting it to zero and activating[9] *returnMoney*. The last three handle the two modes of reset. If *reset1* is active, *stagedTotal* is reset to zero and *returnMoney* is activated. In case of *reset2* becoming active, *stagedTotal* is reset to zero, but *returnMoney* is not activated. If both reset inputs become active at the same time *reset2* is priorized. Listing 6 shows the *stagingArea* block.

[9]For binary inputs/outputs 1 means active, 0 inactive

Listing 6: Example: Staging-Area Block

```
block stagingArea
        in coin, reset1, reset2;
        out stagedTotal : [0..5] init 0;
        out returnMoney : [0..1] init 0;

        reset1 = 0 & reset2 = 0 & coin = 0
                -> (stagedTotal' = stagedTotal) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 1 & stagedTotal < 5
                -> (stagedTotal' = stagedTotal + 1) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 1 & stagedTotal >= 5
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 0 & reset2 = 0 & coin = 2 & stagedTotal < 4
                -> (stagedTotal' = stagedTotal + 2) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 2 & stagedTotal >= 4
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 0 & reset2 = 0 & coin = 3 & stagedTotal < 2
                -> (stagedTotal' = stagedTotal + 4) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 3 & stagedTotal >= 2
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 0 & reset2 = 1
                -> (stagedTotal' = 0) & (returnMoney' = 0);
        reset1 = 1 & reset2 = 0
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 1 & reset2 = 1
                -> (stagedTotal' = 0) & (returnMoney' = 0);
endblock
```

The *actuator* block's purpose is to decide whether the correct amount of money has been inserted. Its single input *val* therefore corresponds to the *stagedTotal* output of the *stagingArea* block. If *val* is five, meaning that 2.50€ have been inserted, it grants a drink by activating its *giveDrink* output. Listing 7 shows the *actuator* block.

Listing 7: Example: Actuator Block

```
block actuator
        in val;
        out giveDrink : [0..1] init 0;

        val < 5 -> (giveDrink' = 0);
        val >= 5 -> (giveDrink' = 1);
endblock
```

4.2 Instances

Dataflow *instances* relate to objects in OOP. Their interface and behavior are defined by the instantiated block, similar to those of an object being defined by its class. Each instance has its own set of state variables. Their names, ranges and initial values are those of the outputs declared in the instantiated block. The configuration of an instance's

state variables at a specific point of time is called the instance's *internal state*. When an instance becomes active, its internal state changes according to the instantiated block's transitionsrelation. An instance's following internal state is a function of its current internal state and its configuration of inputs.

For the example model an instance of each of the blocks defined in the previous section is created. Note that multiple instances of the same block can be created in a single line, by appending further instance names using a comma as separation token. Listing 8 shows the three instance declarations for the vending machine.

Listing 8: Example: Instantiating Blocks

```
instance person : customer;
instance stage : stagingArea;
instance act : actuator;
```

4.3 Wires

Wires are the final language element required in order to create any non-trivial model. Wires connect the outputs of one instance to the inputs of another instance. Outputs may be connected to several inputs, but *only one output may be connected to each input*.

For our example model the first three wires simply connect instance outputs to the corresponding inputs of the following instance. A somewhat special case is the third wire, which creates a backlink to the *stage.reset2* input. This wire is needed in order to trigger the automatic reset upon a successful purchase. Listing 9 shows how wires are created to interconnect the instances of the vending-machine.

Listing 9: Example: Interconnection between Instances

```
wire person.coin to stage.coin;
wire person.reset to stage.reset1;
wire stage.stagedTotal to act.val; // Involved in Cycle
wire act.giveDrink to stage.reset2; // Involved in Cycle
```

4.4 Initializing Cycles in the Instance-Wire-Graph

The network of wires and instances can be represented as a directed, possibly cyclic graph. The latter constitutes a problem, because determining a topological sort of the instance-wire-graph is part of the translation process. This is impossible for cyclic graphs. For this reason the *init* keyword can be appended to a wire declaration. The *init* keyword severs the cycle at the annotated wire. Where to initialize the cycles in a model is up to the author, because the placement of the *init* keywords will determine the instance execution order during translation, as will be explained in Chapter 5. Translation is possible as soon as all cycles are initialized. Finding cycles in the instance-wire-graph

can be algorithmically realized in a number of ways, for example by conducting a depth-first-search for back-edges[10].

In the vending-machine model one cycle involving the instances *stage* and *act* exists. This cycle could be initialized by appending the *init* keyword to either one of the last two wires. As Listing 10 shows, in this case the last of the wires is chosen for initialization in order to ensure that the execution order *person*, *stage*, *act* is chosen during translation.

Listing 10: Example: Initializing Wire Cycles

```
wire person.coin to stage.coin;
wire person.reset to stage.reset1;
wire stage.stagedTotal to act.val;
wire act.giveDrink to stage.reset2 init; // Cycle initialized here
```

With the initialization of the wire cycle, the dataflow model of the vending machine is complete. Figure 7 gives a structural overview of the finished model. The initialized wire is displayed as a dashed arrow to distinguish it from the regular wires.

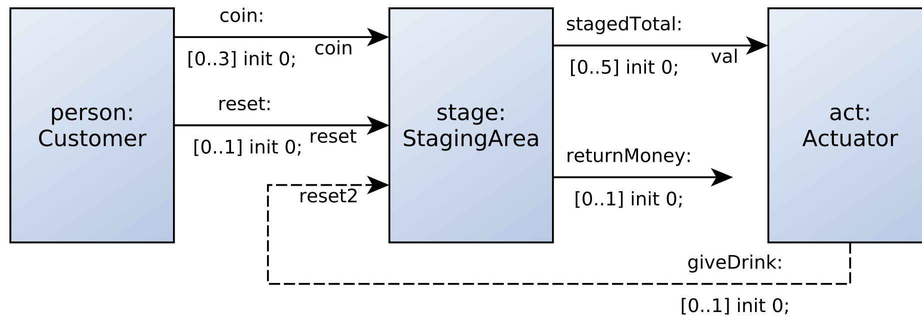

Figure 7: Example: Graphical Representation of the Vending-Machine Dataflow Model

5 Translating Dataflow to SAML

This chapter deals with the algorithmic translation of dataflow models to SAML models. As mentioned earlier, all components of a SAML model become active in parallel at each timestep. Dataflow models, on the other hand, are based on the notion that input values are fed through a series of instances in an order defined by their connecting wires, finally producing output values. Therefore, translating dataflow models to SAML models implies creating a SAML model, which simulates the structure, semantics and execution order of the source model. The algorithmic translation of dataflow models to SAML models takes place in four steps, which will be explained in the following sections. In this approach the SAML components will be synchronized by use of a special synchronization component. In order to illustrate the translation process, the steps will be applied to the vending-machine model, which was constructed in the previous chapter. The SAML model of the translated vending machine is located in Appendix B.2.

5.1 Creating the Synchronization Component

In the first step of the translation process the synchronization component, which is called *PC*[10], is created. The name *PC* was chosen because the purpose of this synchronization component resembles that of the eponymous register in processor architecture. The latter holds a pointer to the memory location, where the currently active CPU instruction is stored[11]. In a similar way, the *PC* component of a translated dataflow model is in charge of signaling each instance its turn to become active. This implies that a dataflow model with n instances will require a *PC* component with a state-variable ranging from 0 to *n-1*. The *PC* is the only component, whose state changes every timestep. Its value is incremented as long as the upper boundary is not reached, then, it is reset to zero. The actual SAML component is always called *_DF_PC_* and its single state variable is always named *state*.

Listing 11 shows the *PC* component which is generated for the vending machine.

Listing 11: Example: Program Counter

```
component _DF_PC_
        state: [0 .. 2] init 0;
        state < 2 -> state' = state + 1;
        state >= 2 -> state' = 0;
endcomponent
```

5.2 Determining the Execution Order

To determine a valid execution order, a graph representation of the instances and their connections in the dataflow model is needed. As one of various possible representations,

[10] Abbreviation for *Program Counter*

instance-adjacency-lists were chosen. Listing 12 outlines how these can be build for any dataflow model.

Listing 12: Pseudo Code: Creating Wire-Adjacency-Lists

```
instanceInEdges = Map<Instance, List>()
instanceOutEdges = Map<Instance, List>()
for instance in allInstances:
        instanceInEdges.put(instance, [])
        instanceOutEdges.put(instance, [])
for wire in allWires:
        inEdges = instanceInEdges.get(wire.getDest())
        outEdges = instanceOutEdges.get(wire.getSource())
        inEdges.add(wire.getSource())
        outEdges.add(wire.getDest())
```

The execution order can then be any topological sort of the graph. Several similarly complex algorithms for determining topological sorts are known, and in essence any of them could be chosen. In this case a variation of an iterative algorithm first proposed by Kahn [8] was selected because its simple implementation suits the adjacency lists nicely. Listing 13 shows the implementation in pseudo-code.

Listing 13: Pseudo Code: Determining Topological Sorts

```
topologicalSort = []
primaryNodes = []
for instance in instanceInEdges.getKeys():
    if instanceInEdges.get(instance).isEmpty():
        primaryNodes.add(instance)
while !primaryNodes.isEmpty():
        instance = primaryNodes.removeFirst()
        topologicalSort.add(instance)
        outEdges = instanceOutEdges.get(instance)
        for target in outEdges:
            inEdges = instanceInEdges.get(target)
            inEdges.remove(instance)
            if inEdges.isEmpty():
                primaryNodes.add(dest)
```

Applying these algorithms to the vending-machine model determines the following, anticipated execution order: (0) *person*, (1) *stage*, (2) *act*. This information will be used in the next step, in which the translation of the dataflow instances to SAML components is explained.

5.3 Creating Components for Instances

Each dataflow instance is independent in terms of its internal state. To preserve this individuality, each instance is translated to a SAML component. The component is named according to the scheme *block_DF_INS_instance* and receives an activation formula called *_DF_ACTIVE_*. This formula evaluates to *true*, whenever the synchronization

component's state value is equal to the instance's index in the execution order. The declarations for the component are derived from the corresponding block's output declarations by omitting the initial *out* tokens. In a similar way the block's update rules are adopted. Instead of removing a token, a check for the activation formula's boolean value is conjunctively prepended to the condition-side of each update rule. One more update rule is then appended, to define the behavior of the component, when the activation formula resolves to *false*. When this is the case, the new values of all state variables are set to their current values. This ensures that the simulated instance's internal state can only change, when it is in fact its turn to become active.

Listing 14 shows the SAML component created for the vending machine's *act* instance by applying the translation described above.

Listing 14: Example: Translated *act* instance before resolution of input qualifiers

```
component actuator_DF_INS_act
    formula _DF_ACTIVE_ := _DF_PC_.state = 2;
    giveDrink : [0..1] init 0;

    _DF_ACTIVE_ & (val < 5) -> (giveDrink' = 0);
    _DF_ACTIVE_ & (val >= 5) -> (giveDrink' = 1);
    !_DF_ACTIVE_ -> (giveDrink' = 1);
endcomponent
```

Note that the component still contains references to the instance's original inputs. These will be dealt with in the following section.

5.4 Resolving Dataflow Input Qualifiers

In this final step of the translation process, all occurrences of dataflow input qualifiers are replaced with appropriate SAML qualifiers by resolving the source model's wires. As explained earlier, each wire connects an *output* of a *source instance* to an *input* of a *destination instance*. Assuming the dataflow model was valid, the components for the source and destination instances of any wire were already created in the previous translation step. For each wire all occurrences of *input* in the *destination instance's* component are now replaced with the SAML qualifier, *<source-block>_DF_INS_<source-instance>.<output>*. The translation of the dataflow model is now complete.

Listing 15 shows the *act* instance's corresponding SAML component after the resolution of the dataflow input qualifiers.

Listing 15: Example: Translated *act* instance after resolution of input qualifiers

```
component actuator_DF_INS_act
    formula _DF_ACTIVE_ := _DF_PC_.state = 2;
    giveDrink : [0..1] init 0;

    _DF_ACTIVE_ & (stagingArea_DF_INS_stage.stagedTotal < 5) -> (giveDrink' = 0);
    _DF_ACTIVE_ & (stagingArea_DF_INS_stage.stagedTotal >= 5) -> (giveDrink' = 1);
    !_DF_ACTIVE_ -> (giveDrink' = 1);
endcomponent
```

6 Extension: Hierarchical Dataflow

6.1 Motivation

The dataflow language can be used to model digital circuits. This can be achieved, for example, by creating a block for the basic *nand* gate. Any more complex operation can then be represented by a network of instances of this block, because any digital circuit can be broken down to a network of *nand* gates[12]. As introduced in Chapter 3, an r/s-flipflop is commonly represented as two nand gates in a feedback system. In a dataflow model the circuit could thus be represented using two instances and two wires as shown in Listing 16.

Listing 16: Example: Non-Hierarchical R/S-Flipflop

```
block nand
    in a,b;
    out v  [0..1] init 0;

    (a <= 0) | (b<=0) -> (v' = 1);
    (a > 0) & (b > 0) -> (v' = 0);
endblock

instance nand1, nand2 : nand;
wire nand1.v to nand2.a;
wire nand2.v to nand1.b init;
```

Assuming the objective was modeling an 8-bit linear shift register[11]. This would imply the need for sixteen instances of the nand block as well as sixteen wires, only to create the flipflops. The interconnection would still be neglected. Obviously this approach does not scale.

The hierarchical extension to the dataflow language enables the definition of hierarchical blocks. This allows creating an hierarchical rs-flipflop block, which encapsulates the two instances and wires of the nand block. Only eight instances of this hierarchical block would then be required to implement the flipflops for the shift register, greatly reducing model size and code duplication.

In the first section of this chapter the syntax modifications required by the hierarchical extension are introduced. The formal syntax definition in BNF is located in the Appendix A.2. The second section will explain how hierarchical dataflow models are transformed to classical dataflow models. The syntax will be introduced by creating the hierarchical r-s/flipflop model, which will then be reused to illustrate the translation process.

After having applied this transformation, the classical dataflow model could then be transformed to a SAML model, using the translation introduced in the previous chapter.

[11]Each bit is stored in an r/s-flipflop.

6.2 Grammar Modification

The hierarchical extension to the dataflow language introduces an alternative to the kind of block known from the previous chapters. In order to distinguish between the two types, the classical block will be referred to as *atomic*, while the new block is called *hierarchical*. Hierarchical blocks are defined in the same way as atomic blocks, the major difference being that they do not contain update rules for their transitionsrelation. Instead they delegate the calculation of their output values to a set of *subblocks*, *subinstances* and *subwires*. These are defined below the block's interface, using the known syntax. The hierarchical block's interface is connected to its subinstances' interfaces using wires. This introduces two new kinds of wires. Classical wires always connect instance-outputs to instance-inputs. Wires in hierarchical blocks can now also connect block-inputs to instance-inputs and instance-outputs to block-outputs. Hierarchical blocks may be nested arbitrarily. Listing 17 demonstrates the creation of an r/s-flipflop with two levels of hierarchical blocks.

Listing 17: Example: Hierarchical R/S-Flipflop

```
block flipflop
    in r,s;
    out q1 : [0..1] init 0;
    out q2 : [0..1] init 0;

    block nand
        in a, b;
        out v : [0..1] init 0;

        block and
            in a, b;
            out v : [0..1] init 0;

            (a > 0) & (b > 0) -> (v' = 1);
            (a <= 0) | (b <= 0) -> (v' = 0);
        endblock

        block not
            in a;
            out v : [0..1] init 0;

            (a <= 0) -> (v' = 1);
            (a > 0) -> (v' = 0);
        endblock

        instance theAnd : and;
        instance theNot : not;
        wire a to theAnd.a;
        wire b to theAnd.b;
        wire theAnd.v to theNot.a;
        wire theNot.v to v;
    endblock
```

```
    instance nand1, nand2 : nand;
    wire s to nand1.a;
    wire r to nand2.b;
    wire nand1.v to nand2.a;
    wire nand2.v to nand1.b init;
    wire nand1.v to q1;
    wire nand2.v to q2;
endblock

instance theFlipFlop : flipflop;
```

6.3 Transformation

The fundamental idea of the transformation algorithm is that hierarchical blocks can be regarded as a template of preconnected instances and wires with an exposed interface. During the translation process, these templates are resolved by replacing instances of hierarchical blocks with appropriately wired networks of their contained atomic blocks. In order to be able to transform arbitrarily nested hierarchical models, the algorithm is applied recursively, starting with the innermost hierarchical blocks. The transformation steps are explained in detail by applying them to the hierarchical model from Listing 17.

Step 1: Rename Atomic Subblocks Find the innermost hierarchical blocks and replace them with copies of their atomic subblocks. These are prefixed with the names of their deleted parent blocks, using *_DF_SUB_* as separation token. In case of the example model the innermost hierarchical block is the *nand* block. Its atomic subblocks are the blocks *and* and *not*, which are appropriately renamed to *nand_DF_SUB_and* and *nand_DF_SUB_not*.

Step 2: Create Instances of Atomic Subblocks Let *M* be the carthesian-product *M1 x M2*, where *M1* are all instances of the hierarchical block and *M2* are all subinstances of the hierarchical block. For each tuple *(instance, subinstance)* in *M* create an instance called *instance_DF_SUB_subinstance* of the renamed type of *subinstance*. In the example building *M* yields the set {*(nand1,theAnd),(nand1,theNot),(nand2,theAnd),(nand2,theNot)*}. Listing 18 shows the instances created for the example model in this step.

Listing 18: Transformed Instances

```
instance nand1_DF_SUB_theAnd, nand2_DF_SUB_theAnd : nand_DF_SUB_and;
instance nand1_DF_SUB_theNot, nand2_DF_SUB_theNot : nand_DF_SUB_not;
```

Step 3: Create Wires from former Subwires For each subwire from *subinstance.output* to *subinstance.input* of the hierarchical block, create wires for all of the new instances. These are of the form *wire *_DF_SUB_subinstance.output to *_DF_SUB_subinstance.input*.

The asterisk is used as placeholder for the names of the original, hierarchical instance. Preserve any init keywords. Listing 19 shows the wires created for the example model in this step.

Listing 19: Transformed Subwires

```
// subwire
wire theAnd.v to theNot.a;
//produces
wire nand1_DF_SUB_theAnd.v to nand1_DF_SUB_theNot.a;
wire nand2_DF_SUB_theAnd.v to nand2_DF_SUB_theNot.a;
```

Step 4: Resolve References in Wires For each wire from *instance.output* to an *instance2.input* involving references to instances of hierarchical blocks, resolve these references:

A hierarchical *instance.output* is resolved to the *instance_DF_SUB_subinstance.output2* of the subwire from *subinstance.output2* to *output*. Listing 20 shows the wires of the example model, which were transformed this way.

Listing 20: Transformed Wires with Resolved Hierarchical Outputs

```
wire nand1.v to q1 // becomes
wire nand1_DF_SUB_theNot.v to q1;
wire nand2.v to q2 // becomes
wire nand2_DF_SUB_theNot.v to q2;
```

A hierarchical *instance.input* is resolved to the *instance_DF_SUB_subinstance.input2* of the subwire from *input* to *subinstance.input2*. Listing 21 shows the wires of the example model, which were transformed this way.

Listing 21: Transformed Wires with Resolved Hierarchical Inputs

```
wire r to nand1.a; // becomes
wire r to nand1_DF_SUB_theAnd.a;
wire s to nand2.b; // becomes
wire s to nand2_DF_SUB_theAnd.b;
```

For wires with references to hierarchical inputs and hierarchical outputs, resolve both references as explained above. Listing 22 shows the wires of the example model, which were transformed this way.

Listing 22: Transformed Wires with Resolved Hierarchical Outputs and Inputs

```
wire nand1.v to nand2.a; // becomes
wire nand1_DF_SUB_theNot.v to nand2_DF_SUB_theAnd.a;
wire nand2.v to nand1.b init; // becomes
wire nand2_DF_SUB_theNot.v to nand1_DF_SUB_theAnd.b init;
```

Step 5: Repeat until Convergence At this point the innermost hierarchical blocks have been transformed to a network of instances of atomic blocks and wires. If the transformed model still contains hierarchical blocks, the algorithm is applied again, until convergence is reached when the model no longer contains hierarchical blocks.

Listing 23 shows the example model after the first application of the transformation algorithm. The flipflop block itself now only contains atomic blocks. Therefore the transformation algorithm can be reapplied, rendering a model completely free of hierarchical blocks. This model is shown in Listing 24 and is ready for translation to SAML.

Listing 23: Example Model after Resolution of the *nand* block

```
block flipflop
    in r,s;
    out q1 : [0..1] init 0;
    out q2 : [0..1] init 0;

    // Start of Resolved Block
    block nand_DF_SUB_and
        in a, b;
        out v : [0..1] init 0;

        (a > 0) & (b > 0) -> (v' = 1);
        (a <= 0) | (b <= 0) -> (v' = 0);
    endblock
    block nand_DF_SUB_not
        in a;
        out v : [0..1] init 0;

        (a <= 0) -> (v' = 1);
        (a > 0) -> (v' = 0);
    endblock

    instance nand1_DF_SUB_theAnd, nand2_DF_SUB_theAnd : nand_DF_SUB_and;
    instance nand1_DF_SUB_theNot, nand2_DF_SUB_theNot : nand_DF_SUB_not;

    wire nand1_DF_SUB_theAnd.v to nand1_DF_SUB_theNot.a;
    wire nand2_DF_SUB_theAnd.v to nand2_DF_SUB_theNot.a;
    // End of Resolved Block

    // Resolved wire destinations
    wire r to nand1_DF_SUB_theAnd.a;
    wire s to nand2_DF_SUB_theAnd.b;
    // Resolved wire sources
    wire nand1_DF_SUB_theNot.v to q1;
    wire nand2_DF_SUB_theNot.v to q2;
    // Resolved wire sorces and destinations
    wire nand1_DF_SUB_theNot.v to nand2_DF_SUB_theAnd.a;
    wire nand2_DF_SUB_theNot.v to nand1_DF_SUB_theAnd.b init;
endblock

instance theFlipFlop : flipflop;
```

Listing 24: Example Model after Convergence of Transformation Algorithm

```
block flipflop_DF_SUB_nand_DF_SUB_and
        in a, b;
        out v : [0..1] init 0;

        ((a>0 ) & (b>0 ) ) -> (v' = 1);
        ((a<=0 ) | (b<=0 ) ) -> (v' = 0);
endblock

block flipflop_DF_SUB_nand_DF_SUB_not
        in a;
        out v : [0..1] init 0;

        (a>0 ) -> (v' = 0);
        (a<=0 ) -> (v' = 1);
.b;endblock

instance theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd,
    theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd
    : flipflop_DF_SUB_nand_DF_SUB_and;
instance theFlipFlop_DF_SUB_nand1_DF_SUB_theNot,
    theFlipFlop_DF_SUB_nand2_DF_SUB_theNot
    : flipflop_DF_SUB_nand_DF_SUB_not;

wire theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd.v
    to theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.a;
wire theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd.v
    to theFlipFlop_DF_SUB_nand2_DF_SUB_theNot.a;
wire theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v
    to theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd.a init;
wire theFlipFlop_DF_SUB_nand2_DF_SUB_theNot.v
    to theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd.b;
```

7 Time Semantics

7.1 Dataflow Time Model

Any dataflow model M can be considered as a function F. Based on an input vector $\overrightarrow{I}$ and the *models internal state*[12] S, it calculates a tuple consisting of the output vector $\overrightarrow{O}$ and the model's new internal state S':

$$M : F(\overrightarrow{I}, S) = (\overrightarrow{O}, S')$$

For the purpose of verifying that the function calculates the correct output tuples for a set of input tuples and internal model states, the concrete number and order of operations is not important. The time required for the calculation of the output tuple for any input and internal model state is therefore simply called one *macrostep*. This name is based on the naming of the timesteps in *Statemate's* asynchronous time model. It was chosen because dataflow timesteps resemble the description of *Statemate's macrostep* as listed by Thums [13]:

1. During a *macrostep* input-values do not change.
2. During a *macrostep* no external events are triggered.
3. During a *macrostep* no time passes.
4. Calculated output-values are exposed to the environment at the end of each *macrostep*.

The translation of dataflow models to SAML models introduces a second, synchronous time model. In the generated SAML model a fixed number of synchronous steps passes, in order to emulate each *macrostep*. To differentiate between the two types of steps, SAML's synchronous steps are called *microsteps*, referring to the naming of the timesteps in *Statemate's* synchronous time model[13].

When specifying verification properties for dataflow models, it is desirable to wire temporal logic formulas based on dataflow's asynchronous time model. However, the translation inevitably exposes SAML's synchronous timesteps. This violates the definition of the macrostep, because the model's output values are always exposed, when they should only be visible at the end of each macrostep. For this reason it is necessary to transform the verification properties, restricting their access to the model's output values to the end of the emulated macrosteps. The transformation of some of the most important building blocks of *CTL*[13] specifications is proposed, without formal proof, in the following section.

[12] The current cofiguration of all instances' internal states.
[13] Computation Tree Logic

7.2 Spec Transformation

CTL is a language for reasoning about the behavior of transition systems. Based on the definitions by Baier et al. [1] this section will introduce some of the most intuitive CTL formulas. For each of the formulas a proposition will be made, on how to adapt it to retain its semantics when the model is translated to SAML. This is necessary, because the formulas rely on the output values of the dataflow model, which are only accessible at the end of each macrostep. Accessing them during intermediate microsteps would yield false negatives, because the calculation of the output values was incomplete. The end of a macrostep is always signaled by the SAML model's program counter being reset to zero. How this information is incorporated into the CTL formulas is explained in the following paragraph*s. These transformations will be used in the case studies to adapt CTL formulas describing the dataflow example models to the different time semantics of the generated SAML models.

AG (All Paths, Globally) Formulas of the type $AG(\alpha)$ are interpreted as: On all possible paths the property α holds in every timestep. To avoid α being verified as false during the SAML model's microsteps, formulas of this type are adapted to:

$$AG((PC = 0) \implies \alpha) \equiv AG((PC \neq 0) \vee \alpha)$$

EF (A Path, Eventually) Formulas of the type $EF(\alpha)$ are interpreted as: On at least one path the property α holds at some point. To ensure that α is only verified as true at the end of macrosteps, formulas of this type are adapted to:

$$EF(PC = 0 \wedge \alpha)$$

AU (All Paths, Until) Formulas of the type $A[\alpha\ U\ \beta]$ are interpreted as: On all possible paths β holds at some point, up to which α holds. To avoid $alpha$ being verified as false and β being being verified as true during microsteps, formulas of this type are adapted to:

$$A[(PC \neq 0 \vee \alpha)\ U\ (PC = 0 \wedge \beta)]$$

EU (A Path, Until) Formulas of the type $E[\alpha\ U\ \beta]$ are interpreted as: On at least one path β holds at some point, up to which α holds. Formulas of this type are adapted similar to AU formulas:

$$E[(PC \neq 0 \vee \alpha)\ U\ (PC = 0 \wedge \beta)]$$

AW/EW (All Paths / A Path, Weak Until) Formulas of the type $A[\alpha\ W\ \beta]$ and $E[\alpha\ W\ \beta]$ are closely related to their AU and EU counterparts, exhibiting one major difference: For these *weak until* formulas β does not ever need to be verified as true, as long as α holds infinitely. AW formulas can be expressed as EU formulas and EW can be expressed as AU formulas[1]. Thereupon the adaptation described for AU and EU formulas can be applied.

$$A[\alpha\ W\ \beta] \equiv \neg E[(\alpha \wedge \neg\beta)\ U\ (\neg\alpha \wedge \neg\beta)]$$
$$\neg E[(PC \neq 0 \vee (\alpha \wedge \neg\beta))\ U\ (PC = 0 \wedge (\neg\alpha \wedge \neg\beta))]$$

$$E[\alpha\ W\ \beta] \equiv \neg A[(\alpha \wedge \neg\beta)\ U\ (\neg\alpha \wedge \neg\beta)]$$
$$\neg A[(PC \neq 0) \vee (\alpha \wedge \neg\beta))\ U\ (PC = 0 \wedge (\neg\alpha \wedge \neg\beta))]$$

AX Formulas of the type $AX(\alpha)$ are interpreted as: On all paths α holds in the next step. For formulas describing dataflow models, the next step is the next macrostep. Therefore the adaptation needs to ensure that α is not verified during the intermediate microsteps. This is achieved by applying the following adaptation to formulas of this type:

$$AX(A[(PC \neq 0)\ U\ (PC = 0) \wedge \alpha])$$

8 Analyzing Dataflow Models

In this chapter the workflow of analyzing dataflow models is demonstrated for the two example models. The vending machine and flipflop models will be dealt with in separate sections by applying the following procedure:

The first step covers the description of the desired verification properties as *CTL specifications* using dataflow's asynchronous time model. The specifications are then adapted to SAML's synchronous time model. In the second step the models are translated to SAML by applying the transformations and translations described in this paper. The model is then supplemented with the adapted specifications and analyzed using the VECS framework's *NuSMV* backend. For this purpose the VECS toolchain transforms the SAML model to an equivalent *SMV* model, which is analyzed with the modelchecker *NuSMV*[14][4, 7].

8.1 Vending Machine

In this section the vending machine model will be analyzed for three behavioral properties, desirable for any real vending machine:

8.1.1 Verification Properties and CTL Specs

A drink is granted if and only if it was completely paid for This property can be expressed by restricting the vending machine to two valid states it may be in at the end of a macrostep: (1) The cost of a drink was *not entirely paid* and a drink was *not granted*, or (2) The cost of a drink *was entirely paid* and a drink *was granted*. In CTL this property is expressed as:

```
AG (((stage.stagedTotal < 5) & (act.giveDrink = 0))
   | ((stage.stagedTotal = 5) & (act.giveDrink = 1)))
```

Adapting the spec to SAML yields:

```
AG (_DF_PC_.state != 0 | (
   ((stagingArea_DF_INS_stage.stagedTotal < 5) & (actuator_DF_INS_act.giveDrink = 1))
   | ((staginaArea_DF_INS_stage.stagedTotal = 5) & (actuator_DF_INS_act.giveDrink = 1))))
```

After granting a drink, the staging area is reset during the next step This property is a logical implication: *On all paths, globally: When a drink is granted, in the following step the staging area is reset to zero.* In CTL it is expressed as:

```
AG ((act.giveDrink = 1) -> (AX (stage.stagedTotal = 0)))
```

Adapting the spec to SAML yields:

[14] New Symbolic Model Verifer

```
AG (_DF_PC_.state != 0 |
((actuator_DF_INS_act.giveDrink = 1) ->
(AX (A [_DF_PC_.state != 0 U stagingArea_DF_INS_stage.stagedTotal = 0]))))
```

A drink can always be purchased This property ensures, that the machine cannot reach a state, in which no further purchases are possible: *On all paths, globally: There exists a path on which a drink is granted eventually.* In CTL it is expressed as:

```
AG (EF (act.giveDrink = 1))
```

Adapting the spec to SAML yields:

```
AG (_DF_PC_.state != 0 | (EF ((_DF_PC_.state = 0) & (actuator_DF_INS_act.giveDrink = 1)))
```

8.1.2 Translation and NuSMV Analysis

The translation of the vending machine model to SAML was covered in Chapter 5 and the resulting model is located in Appendix B.2. This model is now analyzed using VECS' NuSMV backend. After completing the analysis, NuSMV reports the results, which indicate, that all three properties were successfully verified as true. Therefore the modeled vending machine behaves correctly in terms that it fulfills the specified requirements.

8.2 R/S Flipflop

8.2.1 Verification Properties and CTL Specs

In this section the flipflop model will be analyzed for three central properties. For testing purposes, an instance of a block called *sequence* was created, which nondeterministically generates a sequence of the valid input configurations, beginning with a reset. Notice that all specs are prefixed with AX, which allows the model to perform the initial reset, preventing false-negative verifications in the first macrostep. The complete model is located in Appendix C.1.

Flipflop can be set to High This logical implication describes the flipflop's immediate reaction to a set signal: *On all paths, globally: A falling edge on set activates q1.*

```
AX (AG (theSequence.s = 0 -> theFlipflop.q1 = 1))
```

```
AX (AX (AG (_DF_PC_.state != 0
    | (sequence_DF_INS_theSequence.s = 0
    -> flipflop_DF_SUB_nand_DF_SUB_not...
        _DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v = 1))))
```

Flipflop stores High until Reset This logical implication describes that when the flipflop's output is active, this state is stored until it receives a reset signal: *On all paths, globally: When the flipflop's output is active, it either remains active infinitely, or until the next falling edge on the reset input.*

```
AX (AX (AG (theFlipflop.q1 = 1
    -> A [(theFlipflop.q1 = 1) W (theSequence.r = 0)])))
```

Notice, that the AW operator needs to be transformed to an $\neg EU$ operator, as described in the previous chapter.

```
AX (AX (AG (_DF_PC_.state != 0
    | ((flipflop_DF_SUB_nand_DF_SUB_not...
        _DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v = 1
            -> !E [(_DF_PC_.state != 0
                | ((flipflop_DF_SUB_nand_DF_SUB_not...
                    _DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v = 1)
                    & !(sequence_DF_INS_theSequence.r = 0)))
                U (_DF_PC_.state = 0
                & (!(flipflop_DF_SUB_nand_DF_SUB_not...
                    _DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v = 1)
                    & !(sequence_DF_INS_theSequence.r = 0)))])))))
```

Reset sets the Flipflop to Low This logical implication describes that a reset signal leads to the flipflop's output being deactivated in the following macrostep. *On all paths, globally: A falling edge on reset leads to the deactivation of q1 in the following step, if no falling edge on set overrides the reset.*

```
AX (AX (AG (theSequence.r = 0
    -> AX (theSequence.s = 1 -> theFlipflop.q1 = 0))))
```

```
AX (AX (AG (_DF_PC_.state != 0
    | (sequence_DF_INS_theSequence.r = 0
        -> AX (A [_DF_PC_.state != 0 U sequence_DF_INS_theSequence.s = 1
            -> _DF_PC_.state = 0
                & flipflop_DF_SUB_nand_DF_SUB_not...
                    _DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v = 0]))
                        )))
```

8.2.2 Translation and NuSMV Analysis

The transformation of hierarchical-dataflow models to classical dataflow models was explained in detail in Chapter 6. Subjecting the hierarchical flipflop model to this transformation generates a non-hierarchical version, which is located in Appendix C.2. The translation of this model to SAML can now take place analogous to the translation of the vending machine. The model in Appendix C.3 is produced. The model-checking analysis is again performed by the VECS toolchain's NuSMV backend. The results of NuSMV's analysis indicate that all three properties were successfully verified for the flipflop model.

9 Conclusion

This paper has introduced a new language for dataflow-driven model development in the scope of the VECS framework. The design-goal was to make the language as intuitive as possible, without sacrificing too much expressiveness compared to SAML. For this reason the syntax was kept as minimalistic as possible, putting emphasis on a consistent user experience. Nevertheless, the base language encourages the user to write comprehensive models with little redundancy. The *block-instanciation-mechanism* allows users familiar with object oriented programming to apply their philosophy of structuring programs to the modeling process.

Chapter 6 identified an important class of problems, for which the basic dataflow language does not scale: Systems in which a comparatively small number of basic blocks are instantiated a vast number of times in recurrent patterns. The hierarchical-dataflow extension addresses this class of problems by allowing the creation of arbitrarily nested hierarchical blocks. Each recurrent pattern now only needs to be declared once. The hierarchical extension integrates neatly into the dataflow syntax and introduces no further keywords. This way users are given another means of structuring models without having to familiarize themselves with an entirely new syntax.

The translation algorithm proposed in Chapter 5 establishes the connection between the dataflow extension and the existing parts of the VECS framework. The synchronization-component approach, while being a pragmatic solution for pet-models, is most likely not practical when dealing with real-world applications. It increases the limitations of the well known state-space explosion problem[1], by multiplying the state-space with the number of instances in the model. This negative effect is particularly dramatic, when dealing with hierarchical models, since the resolution of the hierarchical blocks increases the number of instances exponentially. The obvious approaches at reducing the state-space implied by the dataflow model are: (1) Optimizing the translation algorithm and (2) Replacing the translation algorithm.

An idea for the first approach is to keep the range of the synchronization-component's state variable as small as possible. Instead using a simple topological sort as execution order, the actual partial ordering of the instance-wire-graph could be determined. This would allow the parallel activation of several instances instead of just a single instance per mircostep. Figure 8 shows an instance-wire-graph for which this optimization approach would reduce the state-space. The partial ordering for this network is $I_1 < I_2 \leq I_3 \leq I_4 < I_5$ for the instances 1 to 5. The instances 2, 3, 4 can be activated in the same microstep, because they are not interdependent. This reduces the range of the synchronization component by two. Applied to real world models this optimization approach would thus greatly reduce the state-space, because such systems seldomly feature only a single thread of execution.

A possible replacement for the translation algorithm could be to translate an entire dataflow model to a SAML component with a single state variable and a complex transitionsrelation. This component would then emulate the entire dataflow model. The

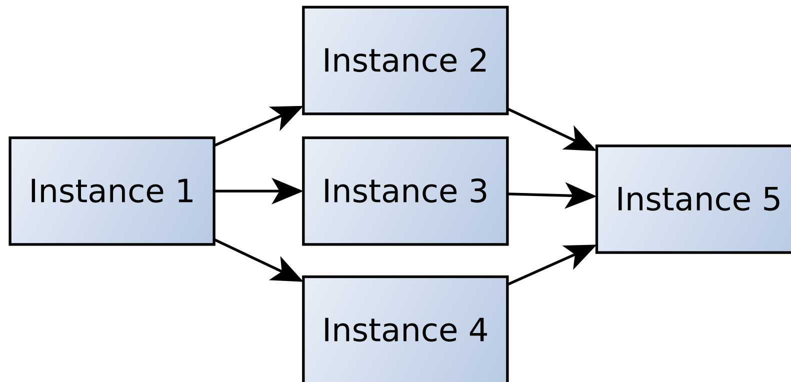

Figure 8: Instance Network benefiting from Optimized Translation

true challenge lies in finding a bijective encoding algorithm, which only respects the actually reachable states of the dataflow model.

Regardless of the translation algorithm, users should preferably attempt to create models which are slender in terms of state-space. Additional language features for facilitating the creation of such models can be conceived. For instance, the formulas known from SAML could be integrated into dataflow blocks, to allow the creation of *stateless-blocks*. These could be applied whenever a block's output-values can be calculated directly from its input-values, without relying on an internal state. Listing 25 demonstrates a possible syntax for stateless-blocks.

Listing 25: Stateless Block using a Formula

```
block adder
    in a, b;
    formula SUM := (a + b);
    out sum : SUM;
endblock
```

A hybrid-graphical editor for dataflow models would be of great additional value, because the circuit-like syntax of the instance-wire-graph is predestined for visualization. A textual mode could be used for the definition of atomic blocks, and a graphical mode for the creation of hierarchical blocks, instances and wires. It would allow a rapid method of model development, by visually creating the model's structure before delving into the detailed definition of each *block's* semantics.

Appendix A BNF Grammars

A.1 Dataflow Grammar

```
// External Declarations
<Root> ::= <Toplevel> | <Toplevel> <Root>
<Toplevel> ::= <Block> | <InstanceDeclaration> | <Wire>

// Block Syntax
<Block> ::= 'block' <BlockName> <Content> 'endblock'
<BlockName> ::= <Identifier>
<Content> ::= 'in' <Inputs> ';' <OutputDecls> <Updates>
    | <OutputDecls> <Updates>

<Inputs> ::= <Input> | <Input> ',' <Inputs>
<Input> ::= <Identifier>

<OutputDecls> ::= <OutputDecl> | <OutputDecl> <OutputDecls>
<OutputDecl> ::= 'out' <Output> ':' <OutputDeclRange> 'init' <Number> ';'
<Output> ::= <Identifier>
<OutputDeclRange> ::= '[' <Number> '..' <Number> ']'

// Block Updates
<Updates> ::= <Update> ';' | <Update> <Updates>
<Update> ::= <Condition> '->' <Assigns> ';'
<Condition> ::= <Disjunction>
<Assigns> ::= <NondetAssigns> | <ProbAssigns>

<NondetAssigns> ::= <NondetAssign> | <NondetAssign> '+' <NondetAssigns>
<NondetAssign> ::= 'choice' ':' <ProbAssigns>
        | 'choice' ':' '(' <ProbAssigns> ')'

<ProbAssigns> ::= <ProbAssign> | <ProbAssign> '+' <ProbAssigns>
<ProbAssign> ::= <Probability> ':' <NextStateAssigns>
        | '(' <Probability> <NextStateAssigns> ')'
        | <NextStateAssigns>

<NextStateAssigns> ::= <NextStateAssign>
                     | <NextStateAssign> '&' <NextStateAssigns>
<NextStateAssign> ::= '(' <Output> '\'' '=' <Addition> ')'

// Instance Declarations
<InstanceDeclaration> ::= 'instance' <Instances> ':' <Type> ';'
<Instances> ::= <Instance> | <Instance> ',' <Instances>
<Instance> ::= <Identifier>
<Type> ::= <Identifier>

// Wire Declarations
<Wire> ::= 'wire' <InstanceOutputQualifier> 'to' <InstanceInputQualifier> ';'

// Boolean Expressions
<Disjunction> ::= <Conjunction> | <Conjunction> '|' <Conjunction>
```

```
<Conjunction> ::= <Negation> | <Negation> '&' <Negation>
<Negation> ::= <PrimaryBoolExpression> | '!' <Negation>
<PrimaryBoolExpression> ::= <Comparison> | 'true' | 'false'
<Comparison> ::= <Addition> <RELATION> <Addition>

//Arithmetical Expressions
<Addition> ::= <Multiplication> | <Multiplication> <ADDOP> <Multiplication>
<Multiplication> ::= <UnaryMinus> | <UnaryMinus> <MULTOP> <UnaryMinus>
<UnaryMinus> ::= <PrimaryExpression> | '-' <UnaryMinus>
<PrimaryExpression> ::= '(' <Disjunction> ')' | <Number> | <Qualifier>

<Qualifier> ::= <InstanceOutputQualifier> | <InstanceInputQualifier>
<InstanceOutputQualifier> ::= <Instance> '.' <Output>
<InstanceInputQualifier> ::= <Instance> '.' <Input>

// Identifiers, Numbers, Sequences
<Identifier> ::= <Character> | <Character> <AlphaNumSeq>
<AlphaNumSeq> ::= <Character> | <DIGIT> | <Character><AlphaNumSeq>
                        | <DIGIT><AlphaNumSeq>
<Number> ::= <DIGIT> | <DIGIT> <Number>
<Character> ::= <LCHAR> | <UCHAR>

// Terminals
<DIGIT> ::= '0'|'1'|'2'|...|'9'
<LCHAR> ::= 'a'|'b'|...|'z'
<UCHAR> ::= 'A'|'B'|...|'Z'
<RELATION> ::= '=' | '!=' | '<=' | '>=' | '<' | '>'
<ADDOP> ::= '+' | '-'
<MULTOP> ::= '*' | '/'
```

A.2 Hierarchical Extension Grammar

```
// The grammar for the hierarchical-dataflow extension only shows
// newly created or modified rules.

// Block rule modified to allow hierarchical content
<Block> ::= 'block' <BlockName> <Content> 'endblock'
     | 'block' <BlockName> <HierarchicalContent> 'endblock'

// Rule defines contents of hierarchical blocks
<HierarchicalContent> ::= 'in' <Inputs> ';' <OutputDecls> <Root>
     | <OutputDecls> <Root>

// Modified wire rule to allow connecting block interface to
// subinstance interface
<Wire> ::= 'wire' <InstanceOutputQualifier> 'to' <InstanceInputQualifier> ';'
     | 'wire' <Input> 'to' <InstanceInputQualifier> ';'
     | 'wire' <InstanceOutputQualifier> 'to' <Output> ';'
```

Appendix B Vending Machine Model

B.1 Dataflow Model

```
block customer
        out coin : [0..3] init 0;
        out reset : [0..1] init 0;

        true -> choice : ((coin' = 0) & (reset' = 0))
                       + choice : ((coin' = 1) & (reset' = 0))
                       + choice : ((coin' = 2) & (reset' = 0))
                       + choice : ((coin' = 3) & (reset' = 0))
                       + choice : ((coin' = 0) & (reset' = 1));
endblock

block stagingArea
        in coin, reset1, reset2;
        out stagedTotal : [0..5] init 0;
        out returnMoney : [0..1] init 0;

        reset1 = 0 & reset2 = 0 & coin = 0
                -> (stagedTotal' = stagedTotal) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 1 & stagedTotal < 5
                -> (stagedTotal' = stagedTotal + 1) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 1 & stagedTotal >= 5
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 0 & reset2 = 0 & coin = 2 & stagedTotal < 4
                -> (stagedTotal' = stagedTotal + 2) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 2 & stagedTotal >= 4
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 0 & reset2 = 0 & coin = 3 & stagedTotal < 2
                -> (stagedTotal' = stagedTotal + 4) & (returnMoney' = 0);
        reset1 = 0 & reset2 = 0 & coin = 3 & stagedTotal >= 2
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 0 & reset2 = 1
                -> (stagedTotal' = 0) & (returnMoney' = 0);
        reset1 = 1 & reset2 = 0
                -> (stagedTotal' = 0) & (returnMoney' = 1);
        reset1 = 1 & reset2 = 1
                -> (stagedTotal' = 0) & (returnMoney' = 0);
endblock

block actuator
        in val;
        out giveDrink : [0..1] init 0;

        val < 5 -> (giveDrink' = 0);
        val >= 5 -> (giveDrink' = 1);
endblock


instance person : customer;
```

```
instance stage : stagingArea;
instance act : actuator;

wire person.coin to stage.coin;
wire person.reset to stage.reset1;
wire stage.stagedTotal to act.val;
wire act.giveDrink to stage.reset2 init;

NUSMVSPEC AG (((stage.stagedTotal < 5) & (act.giveDrink = 0)) | ((stage.stagedTotal = 5) & (

NUSMVSPEC AG (act.giveDrink = 1 -> (AX (stagingAreaa_DF_INS_stage.stagedTotal = 0)))

NUSMVSPEC AG (EF (actuator_DF_INS_act.giveDrink = 1))
```

B.2 SAML Model

```
component _DF_PC_
        state : [0..2] init 0;
        (state<2 ) -> (state' = (state + 1 ) );
        (state>=2 ) -> (state' = 0);
endcomponent
component customer_DF_INS_person
        formula _DF_ACTIVE_ := (_DF_PC_.state=0 ) ;
        coin : [0..3] init 0;
        reset : [0..1] init 0;
        (_DF_ACTIVE_ & true ) -> choice: ((coin' = 0) & (reset' = 0)) +
            choice: ((coin' = 1) & (reset' = 0)) + choice: ((coin' = 2) & (
            reset' = 0)) + choice: ((coin' = 3) & (reset' = 0)) + choice:
            ((coin' = 0) & (reset' = 1));
        !_DF_ACTIVE_ -> (coin' = coin) & (reset' = reset);
endcomponent
component stagingArea_DF_INS_stage
        formula _DF_ACTIVE_ := (_DF_PC_.state=1 ) ;
        stagedTotal : [0..5] init 0;
        returnMoney : [0..1] init 0;
        (_DF_ACTIVE_ & (((customer_DF_INS_person.reset=0 ) & (
            actuator_DF_INS_act.giveDrink=0 ) ) & (customer_DF_INS_person
            .coin=0 ) ) ) -> (stagedTotal' = stagedTotal) & (returnMoney
            ' = 0);
        (_DF_ACTIVE_ & ((((customer_DF_INS_person.reset=0 ) & (
            actuator_DF_INS_act.giveDrink=0 ) ) & (customer_DF_INS_person
            .coin=1 ) ) & (stagedTotal<5 ) ) ) -> (stagedTotal' = (
            stagedTotal + 1 ) ) & (returnMoney' = 0);
        (_DF_ACTIVE_ & ((((customer_DF_INS_person.reset=0 ) & (
            actuator_DF_INS_act.giveDrink=0 ) ) & (customer_DF_INS_person
            .coin=1 ) ) & (stagedTotal>=5 ) ) ) -> (stagedTotal' = 0)
            & (returnMoney' = 1);
        (_DF_ACTIVE_ & ((((customer_DF_INS_person.reset=0 ) & (
            actuator_DF_INS_act.giveDrink=0 ) ) & (customer_DF_INS_person
            .coin=2 ) ) & (stagedTotal<4 ) ) ) -> (stagedTotal' = (
            stagedTotal + 2 ) ) & (returnMoney' = 0);
        (_DF_ACTIVE_ & ((((customer_DF_INS_person.reset=0 ) & (
            actuator_DF_INS_act.giveDrink=0 ) ) & (customer_DF_INS_person
```

```
        .coin=2 )   )  & (stagedTotal>=4 )   )   ) -> (stagedTotal' = 0)
        & (returnMoney' = 1);
    (_DF_ACTIVE_ & ((((customer_DF_INS_person.reset=0 )  & (
        actuator_DF_INS_act.giveDrink=0 )   )  & (customer_DF_INS_person
        .coin=3 )   )  & (stagedTotal<2 )   )   ) -> (stagedTotal' = (
        stagedTotal + 4 ) ) & (returnMoney' = 0);
    (_DF_ACTIVE_ & ((((customer_DF_INS_person.reset=0 )  & (
        actuator_DF_INS_act.giveDrink=0 )   )  & (customer_DF_INS_person
        .coin=3 )   )  & (stagedTotal>=2 )   )   ) -> (stagedTotal' = 0)
        & (returnMoney' = 1);
    (_DF_ACTIVE_ & ((customer_DF_INS_person.reset=0 )  & (
        actuator_DF_INS_act.giveDrink=1 )   )   ) -> (stagedTotal' = 0)
        & (returnMoney' = 0);
    (_DF_ACTIVE_ & ((customer_DF_INS_person.reset=1 )  & (
        actuator_DF_INS_act.giveDrink=0 )   )   ) -> (stagedTotal' = 0)
        & (returnMoney' = 1);
    (_DF_ACTIVE_ & ((customer_DF_INS_person.reset=1 )  & (
        actuator_DF_INS_act.giveDrink=1 )   )   ) -> (stagedTotal' = 0)
        & (returnMoney' = 0);
    !_DF_ACTIVE_ -> (stagedTotal' = stagedTotal) & (returnMoney' =
        returnMoney);
endcomponent
component actuator_DF_INS_act
    formula _DF_ACTIVE_ := (_DF_PC_.state=2 ) ;
    giveDrink : [0..1] init 0;
    (_DF_ACTIVE_ & (stagingArea_DF_INS_stage.stagedTotal<5 )   ) -> (
        giveDrink' = 0);
    (_DF_ACTIVE_ & (stagingArea_DF_INS_stage.stagedTotal>=5 )   ) -> (
        giveDrink' = 1);
    !_DF_ACTIVE_ -> (giveDrink' = giveDrink);
endcomponent

NUSMVSPEC AX (AG (_DF_PC_.state != 0 | ((stagingArea_DF_INS_stage.
    stagedTotal < 5) & (actuator_DF_INS_act.giveDrink = 0)) | ((
    stagingArea_DF_INS_stagel.stagedTotal = 5) & (actuator_DF_INS_act.
    giveDrink = 1))))

NUSMVSPEC AX (AG (_DF_PC_.state != 0 | (actuator_DF_INS_act.giveDrink = 1
    -> (AX (A[(_DF_PC_.state != 0) U (_DF_PC_.state = 0 & (
    stagingAreaa_DF_INS_stage.stagedTotal = 0))])))))

NUSMVSPEC AX (AG (_DF_PC_.state != 0 | (EF (_DF_PC_.state = 0 &
    actuator_DF_INS_act.giveDrink = 1))))
```

Appendix C Flipflop Model

C.1 Hierarchical Dataflow Model

```
block flipflop
    in r,s;
    out q1 : [0..1] init 0;
    out q2 : [0..1] init 0;

    block nand
        in a, b;
        out v : [0..1] init 0;

        block and
            in a, b;
            out v : [0..1] init 0;

            (a > 0) & (b > 0) -> (v' = 1);
            (a <= 0) | (b <= 0) -> (v' = 0);
        endblock

        block not
            in a;
            out v : [0..1] init 0;

            (a <= 0) -> (v' = 1);
            (a > 0) -> (v' = 0);
        endblock

        instance theAnd : and;
        instance theNot : not;
        wire a to theAnd.a;
        wire b to theAnd.b;
        wire theAnd.v to theNot.a;
        wire theNot.v to v;
    endblock

    instance nand1, nand2 : nand;
    wire s to nand1.a;
    wire r to nand2.b;

    wire nand1.v to nand2.a;
    wire nand2.v to nand1.b init;
    wire nand1.v to q1;
    wire nand2.v to q2;
endblock

block sequence
    out r : [0..1] init 0;
    out s : [0..1] init 1;

    true -> choice: ((r' = 0) & (s'= 1))
```

```
        + choice: ((r' = 1) & (s'= 0))
        + choice: ((r' = 1) & (s'= 1));
endblock

instance theFlipFlop : flipflop;
instance theSequence : sequence;

wire theSequence.r to theFlipFlop.r;
wire theSequence.s to theFlipFlop.s;

NUSMVSPEC AX (AX (AG (theSequence.s = 0 -> theFlipflop.q1 = 1)))

NUSMVSPEC AX (AX (AG (theFlipflop.q1 = 1 -> A [(theFlipflop.q1 = 1) W (
    theSequence.r = 0)])))

NUSMVSPEC AX (AX (AG (theSequence.r = 0 -> AX (sequence_DF_INS_theSequence.
    s = 1 -> theFlipflop.q1 = 0))))
```

C.2 Dataflow Model

```
block flipflop_DF_SUB_nand_DF_SUB_and
        in a, b;
        out v : [0..1] init 0;

        ((a>0 ) & (b>0 ) ) -> (v' = 1);
        ((a<=0 ) | (b<=0 ) ) -> (v' = 0);
endblock

block flipflop_DF_SUB_nand_DF_SUB_not
        in a;
        out v : [0..1] init 0;

        (a<=0 ) -> (v' = 1);
        (a>0 ) -> (v' = 0);
endblock

block sequence
        out r : [0..1] init 0;
        out s : [0..1] init 1;

        true -> choice: ((r' = 0) & (s' = 1)) + choice: ((r' = 1) & (s' =
            0)) + choice: ((r' = 1) & (s' = 1));
endblock

instance theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd,
    theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd :
    flipflop_DF_SUB_nand_DF_SUB_and;
instance theFlipFlop_DF_SUB_nand1_DF_SUB_theNot,
    theFlipFlop_DF_SUB_nand2_DF_SUB_theNot :
    flipflop_DF_SUB_nand_DF_SUB_not;
instance theSequence : sequence;

wire theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd.v to
```

```
    theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.a;
wire theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd.v to
    theFlipFlop_DF_SUB_nand2_DF_SUB_theNot.a;
wire theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v to
    theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd.a;
wire theFlipFlop_DF_SUB_nand2_DF_SUB_theNot.v to
    theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd.b init;
wire theSequence.r to theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd.b;
wire theSequence.s to theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd.ablock
    flipflop_DF_SUB_nand_DF_SUB_and

NUSMVSPEC AX (AX (AG (theSequence.s = 0 ->
    theFlipFlop_DF_SUB_nand1_DF_SUB_theNot.v = 1)))

NUSMVSPEC AX (AX (AG (theFlipFlop_DF_SUB_nand1_DF_SUB_theNot = 1 -> A [(
    theFlipFlop_DF_SUB_nand1_DF_SUB_theNot) W (theSequence.r = 0)])))

NUSMVSPEC AX (AX (AG (theSequence.r = 0 -> AX (sequence_DF_INS_theSequence.
    s = 1 -> theFlipFlop_DF_SUB_nand1_DF_SUB_theNot = 0))))
```

C.3 SAML Model

```
component _DF_PC_
        state : [0..4] init 0;
        state < 4 -> state' = state + 1;
        state >= 4 -> state' = 0;
endcomponent

component
    flipflop_DF_SUB_nand_DF_SUB_and_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd

        formula _DF_ACTIVE_ := _DF_PC_.state = 1;
        v : [0..1] init 0;

        _DF_ACTIVE_ & (((sequence_DF_INS_theSequence.s>0 )  & (
            flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand2_DF_SUB_theNot
            .v>0 )   ) ) -> (v' = 1);
        _DF_ACTIVE_ & (((sequence_DF_INS_theSequence.s<=0 )   | (
            flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand2_DF_SUB_theNot
            .v<=0 )   ) ) -> (v' = 0);
        !_DF_ACTIVE_ -> (v' = v);
endcomponent

component
    flipflop_DF_SUB_nand_DF_SUB_and_DF_INS_theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd

        formula _DF_ACTIVE_ := _DF_PC_.state = 3;
        v : [0..1] init 0;

        _DF_ACTIVE_ & (((
            flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
            .v>0 )  & (sequence_DF_INS_theSequence.r>0 )   ) ) -> (v' = 1);
        _DF_ACTIVE_ & (((
```

```
            flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
            .v<=0 )  | (sequence_DF_INS_theSequence.r<=0 )  ) ) -> (v' = 0)
            ;
        !_DF_ACTIVE_ -> (v' = v);
endcomponent

component
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot

        formula _DF_ACTIVE_ := _DF_PC_.state = 2;
        v : [0..1] init 0;

        _DF_ACTIVE_ & ((
            flipflop_DF_SUB_nand_DF_SUB_and_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd
            .v<=0 ) ) -> (v' = 1);
        _DF_ACTIVE_ & ((
            flipflop_DF_SUB_nand_DF_SUB_and_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theAnd
            .v>0 ) ) -> (v' = 0);
        !_DF_ACTIVE_ -> (v' = v);
endcomponent

component
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand2_DF_SUB_theNot

        formula _DF_ACTIVE_ := _DF_PC_.state = 4;
        v : [0..1] init 0;

        _DF_ACTIVE_ & ((
            flipflop_DF_SUB_nand_DF_SUB_and_DF_INS_theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd
            .v<=0 ) ) -> (v' = 1);
        _DF_ACTIVE_ & ((
            flipflop_DF_SUB_nand_DF_SUB_and_DF_INS_theFlipFlop_DF_SUB_nand2_DF_SUB_theAnd
            .v>0 ) ) -> (v' = 0);
        !_DF_ACTIVE_ -> (v' = v);
endcomponent

component sequence_DF_INS_theSequence
        formula _DF_ACTIVE_ := _DF_PC_.state = 0;
        r : [0..1] init 0;
        s : [0..1] init 1;

        _DF_ACTIVE_ & (true) -> choice: ((r' = 0) & (s' = 1)) + choice: ((r
            ' = 1) & (s' = 0)) + choice: ((r' = 1) & (s' = 1));
        !_DF_ACTIVE_ -> (r' = r) & (s' = s);
endcomponent

NUSMVSPEC AX (AX (AG (_DF_PC_.state != 0 | (sequence_DF_INS_theSequence.s =
     0 ->
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
    .v = 1))))

NUSMVSPEC AX (AX (AG (_DF_PC_.state != 0 | ((
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
    .v = 1 -> !E [(_DF_PC_.state != 0 | ((
```

```
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
    .v = 1) & !(sequence_DF_INS_theSequence.r = 0))) U (_DF_PC_.state = 0 &
     (!(
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
    .v = 1) & !(sequence_DF_INS_theSequence.r = 0)))])))))

NUSVMSPEC AX (AX (AG (_DF_PC_.state != 0 | (sequence_DF_INS_theSequence.r =
     0 -> AX (A [_DF_PC_.state != 0 U sequence_DF_INS_theSequence.s = 1 ->
    _DF_PC_.state = 0 &
    flipflop_DF_SUB_nand_DF_SUB_not_DF_INS_theFlipFlop_DF_SUB_nand1_DF_SUB_theNot
    .v = 0])))))
```

Appendix D Implementation Notes

The (hierarchical-) dataflow language, transformation- and translation- algorithms were implemented as extension to the VECS project's SAML IDE. The source code can be found in the *vecs-backend* Eclipse plugin.

Saml.xtext The Xtext grammar definition of the SAML language now also contains the grammars for the dataflow language and hierarchical extension. The Xtext grammar is similar to the BNF grammar proposed in this paper, but contains additional information about references, scoping, etc.

DFTransformer.java and DFHierarchyTransformer.java These classes use VECS' translator architecture to implement the transformation and translation of (hierarchical-) dataflow models to saml models.

InstanceGraphUtil.java and WireGraphUtil.java These classes contain the algorithms for building and analyzing the instance-wire graph (cycle-detection, topological-sorting).

SamlJavaValidator.java This class was extended to make the SAML editor highlight all wires involved in cycles of the instance-wire-graph.

References

[1] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[2] Gérard Berry. Synchronous design and verification of critical embedded systems using scade and esterel. *Lecture Notes in Computer Science*, 4916:2–2, 2008.

[3] Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform. In *Computer Safety, Reliability, and Security*, pages 49–62. Springer, 2003.

[4] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.

[5] Jürgen Dassow. *Logik für Informatiker*. Springer DE, 2005.

[6] Matthias Güdemann. *Qualitative and quantitative formal model-based safety analysis*. PhD thesis, OvGU Magdeburg, 2011.

[7] Matthias Güdemann and Frank Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 132–141. IEEE, 2010.

[8] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[9] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[10] Gunter Saake and Kai-Uwe Sattler. *Algorithmen und Datenstrukturen*. dpunkt-Verlag, 4 edition, 2010.

[11] Andrew S Tanenbaum. *Modern operating systems*. Prentice hall Englewood Cliffs, 1992.

[12] Gerald Teschl and Susanne Teschl. *Mathematik für Informatiker: Band 1: Diskrete Mathematik und Lineare Algebra*, volume 1. Springer DE, 2010.

[13] Andreas Thums. *Formale Fehlerbaumanalyse*. PhD thesis, University of Augsburg, 2004.

[14] Gottfried Vossen and Kurt-Ulrich Witt. *Grundkurs Theoretische Informatik: Eine anwendungsbezogene Einführung. Für Studierende in allen Informatik-Studiengängen*. Springer DE, 5 edition, 2011.