Case Study: Software-engineering-friendly bi-transformation from AutoFOCUS to a model checker

Sudeep Kanav and Vincent Aravantinos

fortiss GmbH, Guerickestr. 25, 80805 Munich, Germany
{kanav, aravantinos}@fortiss.org

Abstract

We propose a case study for the Transformation Tool Contest (TTC) 2016 in this paper targeting the transformation of AutoFOCUS models to a model checker, and very importantly the reverse transformation from counterexample traces from model check to AutoFOCUS simulation traces. One key aspect to be addressed in the solution, which also makes this problem more interesting, is how "software-engineering"-friendly is the transformation. Hence the solution transformations will not be evaluated only on correctness, but also on the "resistance" to change (e.g., bug fix, change of requirement).

Keywords: model checking, bi-transformation, AutoFOCUS3

1 Introduction / Context

This case study targets the transformation of AutoFOCUS(AF3) models (components + state automata models) to a model checker (NuSMV/nuXmv). Most importantly, it targets as well the transformation from NuSMV/nuXmv *traces* to AF3 *simulations*. These two transformations do not constitute a bi-transformation because the transformation of traces to simulations is not a *reverse* transformation of the first transformation: NuSMV/nuXmv models are not NuSMV/nuXmv traces, and AF3 simulations are not AF3 models. However both transformations have a lot in common: AF3 ports are transformed (in the "first" transformation) into NuSMV/nuXmv variables, which are to be transformed back into ports in the "trace to simulation" transformation.

The transformations per se are not particularly difficult or different than others would be. **However the exciting challenge** in this case study is that we would like a "software-engineering friendly" solution: the transformation is part of a bigger project (AutoFOCUS) which often undergoes changes like bug fixes (e.g., the transformation is buggy and needs to be fixed), improvements (e.g., some model is better transformed into enumerations rather than integers), extensions (e.g., new types, new sorts of execution models are added and therefore need to be transformed as well). We therefore want not only a pair of correct transformations, but also a pair of transformations which are easily amenable to changes, re-engineering, refactoring. In particular, since both transformations share a lot, it is expected that if a change is required in both transformations, then only one place needs to be changed in the code/model describing the transformations.

For the evaluation of the solution, we provide of course a list of input models/expected models. These are made in such a way that they represent incremental steps in the *development of the transformation*: models of increasing complexity are considered, or different transformations are considered. This is typical of what happens when engineering a transformation: we start by considering simple models and then increase the expressiveness and we consider various ways of transforming the original models. The solution will therefore be evaluated to reflect this process: we do not expect the final (i.e., the most expressive) transformation only, but also the intermediate increments in order to assess how "small" the deltas are. The most "software-engineering-friendly" solution should provide the smallest deltas. Note that we do not require these increments to be the ones which are indeed accomplished during the case study: it is ok if an increment turns out to be later non-satisfactory and therefore needs to be re-worked. We are only interested in the transformation resulting finally of each increment.

We now provide more details about the context of the transformation.

1.1 AutoFOCUS3 ("AF3")

→ *AF3 is a model-based tool for systems engineering with formally defined execution semantics*

AF3 is an open source, seamlessly integrated model based development tool for distributed, reactive, embedded software systems. It is built on a system model based on the FOCUS theory [1] that allows to precisely describe a system, its interface, behavior, and decomposition in component systems on different levels of abstraction.

It covers the complete development process from requirements elicitation, modelling of software architecture, modelling of the hardware platform to code generation. In AF3 models are used for requirements, for the software architecture (SA), for the hardware platform and for relations between those different viewpoints: traces from requirements to the SA, refinements between SAs, and deployments of the SA to the platform. These models are projections of the underlying system model in AutoFOCUS. It uses different views for these aspects, which provide dedicated description techniques for different structural and behavioral aspects.

Software architecture and behavior of the system is specified using a component-based language with a formal semantics based on the FOCUS theory[1]. The components run in parallel according to a global, synchronous, and discrete time clock [1]. State automata, tables, or simple imperative code could be used to define the behavior of the components. Components interact with each other through typed input and output ports which are connected by fixed channels. Components can be simulated for testing purposes. Formal verification based on model checking can also be used to validate of the design.

1.2 The AF3 to NuSMV/nuXmv transformation

→ *AF3 integrates formal verification transparently*
→ *requires a **transformation** to a model checker*

Formal verification can help in finding design errors in an early stage of the system development. To do so AF3 makes use of the NuSMV (or its successor nuXmv [2]) *model checker* : AF3 components are transformed into NuSMV/nuXmv components and specifications are transformed into NuSMV/nuXmv temporal logic formulas. This is made possible because AF3 is based on FOCUS, which has formal semantics. The transformation of AF3 into NuSMV/nuXmv has to be performed transparently to the user in order to provide a user friendly workflow.

nuXmv is the successor of NuSMV/nuXmv model checker, but with respect to this case study there is no difference in the models. nuXmv has different license than NuSMV/nuXmv but both are can be downloaded.

1.3 The NuSMV/nuXmv to AF3 transformation

→ *need to **transform** NuSMV/nuXmv traces into AF3 simulations*
→ *this transformation and the previous one should share maximum information*

When a specification does not hold, the model checker returns a counter-example in the form of a trace. Such a trace can then be transformed into an AF3 simulation: this is essential for the user to get some comprehensive feedback regarding the reasons why the speicification is not satisfied.

It is to be noted, that this is *not* a bidirectional transformation, even though it may look like one: we are not interested in transforming the NuSMV/nuXmv input to the AF3 model, but in the transformation of a *trace* which, despite sharing a lot with the original transformation, is not a model. For software engineering considerations, it is however essential that both transformations share the maximum that they can share: if a similar reduction is done on both side then it should be expressed in the code only once to ensure that further changes are automatically taken into account on both sides.

2 The Two Transformations

In this section we provide more details on the transformations. We present the metamodels of AF3 and NuSMV/nuXmv, then the transformation and finally the variation points.

2.1 AF3 to NuSMV/nuXmv

2.1.1 AF3 Metamodel

→ *classical (but simplified) component+state machine metamodel*
→ *only two datatypes : boolean and integer*

In this subsection we present the metamodel of AF3:

1. we start with a concrete example of an AF3 model,
2. we then present a conceptual metamodel (i.e., restricted and simplified w.r.t. the real metamodel which a) is big, b) includes many concepts not considered in this case study and c) is distributed across several ecore models) describing the key elements of the metamodel used in AF3,
3. we finally show the real metamodel as used in AF3.

2.1.1.1 Concrete Example of a Model

As an example, we present a model of a component in AF3 (Figure 1) which counts the number of signals coming on the input port ***i*** without a break, and puts the count on the output port ***o***. It resets the count every time no signal is received. The notion of input not being present (or "no signal is received") is explicitly modeled in AF3 by giving it a special value "***NoVal***". This component is implemented as a state automaton with two states **init** and **s**.

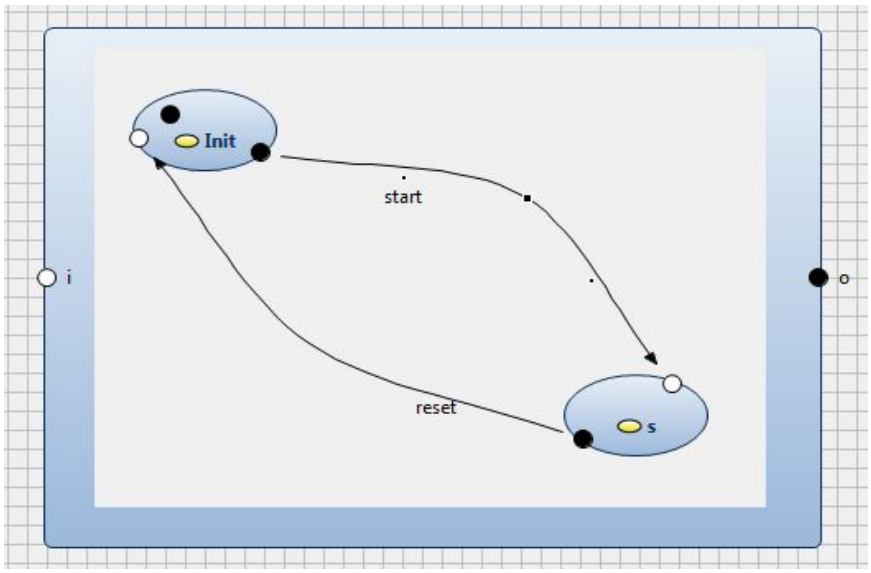

Figure 1.

In Figure 2, the transition table for this state automaton is given. The automaton uses an internal variable **v** to store the sum of the preceding inputs. It also lists the idle state actions (named _IDLE STATE ACTIONS_) which are executed for every clock tick in case no other transition is being executed which changes the state.

Source	Target	Name	Guard	Action
Init	s	start	i != NoVal	v = i
Init	Init	_IDLE STATE ACTIONS_		o=NoVal
s	Init	reset	i == NoVal	o=v
s	s	_IDLE STATE ACTIONS_		v=v+i; o=v

Figure 2.

2.1.1.2 Conceptual (simplified) Metamodel

We now explain the simplified meta model. This metamodel is *not to be used for the transformation*: we present it only for explanation purposes.

The metamodel is an ecore model comprising of three different sub-packages i) expression, ii) component, and iii) state. It also contains an interface with the name `INamedElement` which is implemented by the classes required to have string identifiers(or names).

The component package contains the following:

- `Component`: class which contains a state automaton and lists of input and output ports.
- `Port`: base class for input and output ports. A port has a name, type and an initial value.
- `InputPort`: class for input ports.
- `OutputPort`: class for output ports.

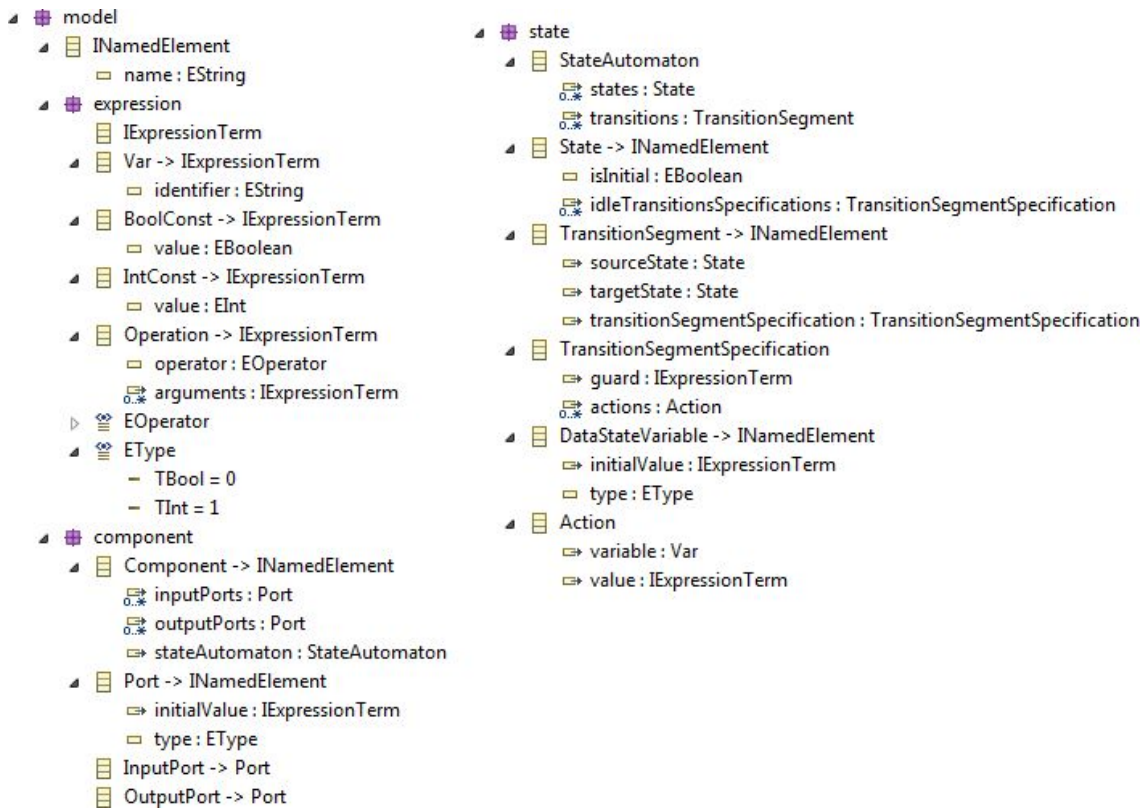

Figure 3

The state package contains the following:

- `StateAutomaton`: Class for state automata. It contains the collection of states and transitions.
- `State`: Class for states. States have a name, contain a boolean attribute `isInitial` specifying if the state is the initial state for automaton. It is to be noted that the metamodel does not enforces that the state automaton contains only one initial state, but it is assumed that it is the case. State also contains a list of the idle state transitions. Multiple idle state transitions are allowed as there can be different transitions depending on the guard.
- `TransitionSegment`: Class for transitions. Transitions are named and contain a reference to a source and a target state, as well as a transition segment "specification".
- `TransitionSegmentSpecification`: Class for specifications of transition segments. Contains a guard and a list of actions to be performed for the transition.
- `Action`: Class containing a variable and a value to be assigned to that variable. Actions are assignments of values to either an output port or a state automaton internal variable. Therefore the variable of the action is either the name of an output port or the name of an internal data state variable (see next item).
- `DataStateVariable`: Class for internal variables of the state automata. Contains a name, type and initial value. In the example of the previous section, "`v`" was such a variable.

The expression package contains the following classes/interfaces:

- `IExpressionTerm`: interface for all the expressions.
- `Var`: Class for variables. Simply a string typically referring to a port or a state variable.
- `BoolConst`: Class for boolean constants. Either true or false.
- `IntegerConst`: Class for integer constants. Any integer.
- `Operation`: Class for operations. It contains the operator and the arguments for the operator.
- `EOperator`: Enumeration of the available operators. Contains the basic operators listed in Figure 4.
- `EType`: Enumeration of types. We have only integer and boolean types in this simplified model.

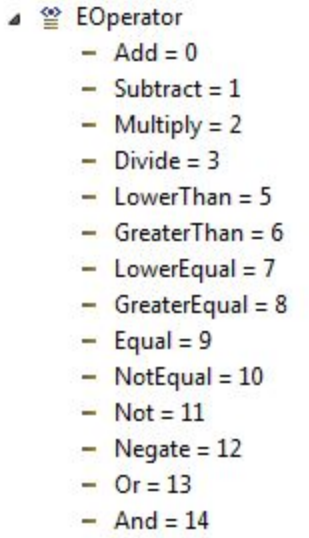

Figure 4

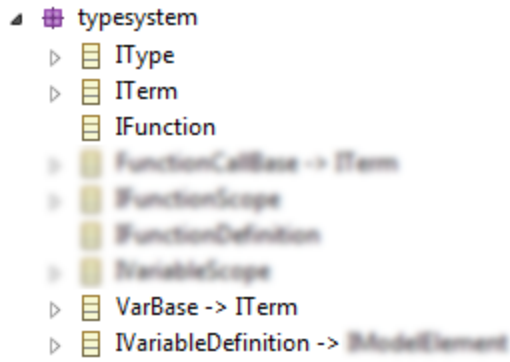

Figure 5

2.1.1.3 Real metamodel

→ *collection of metamodels (several ecore files)*
→ *very generic metamodels (heavy use of interfaces to factorize metamodel elements)*

As explained in introduction, AF3 covers many aspects of systems engineering and not just the above presented component architecture. Consequently, the metamodel is actually a collection of metamodels which are distributed in several ecore models. The metamodel is also designed to avoid redefinitions of similar elements and therefore makes heavy usage of interfaces to factorize definitions, like "`INamedElement`" in the previous simplified metamodel.

We now briefly present the metamodel elements in AF3 corresponding to the conceptual metamodel. The elements which are not relevant to this case study have been blurred out. *This presentation is not intended to be exhaustive: the actual metamodels are anyways of course provided with this case study.*

Figure 5 shows the metamodel for type system. The interface ITerm defines the abstract interface for every term, `IType` defines the abstract interface for type. It corresponds to "`EType`" in the conceptual metamodel. `IFunction` defines the abstract interface for functions. `VarBase` is an abstract class defining the named variables, and `IVariableDefinition` defines the interface for variable definitions.

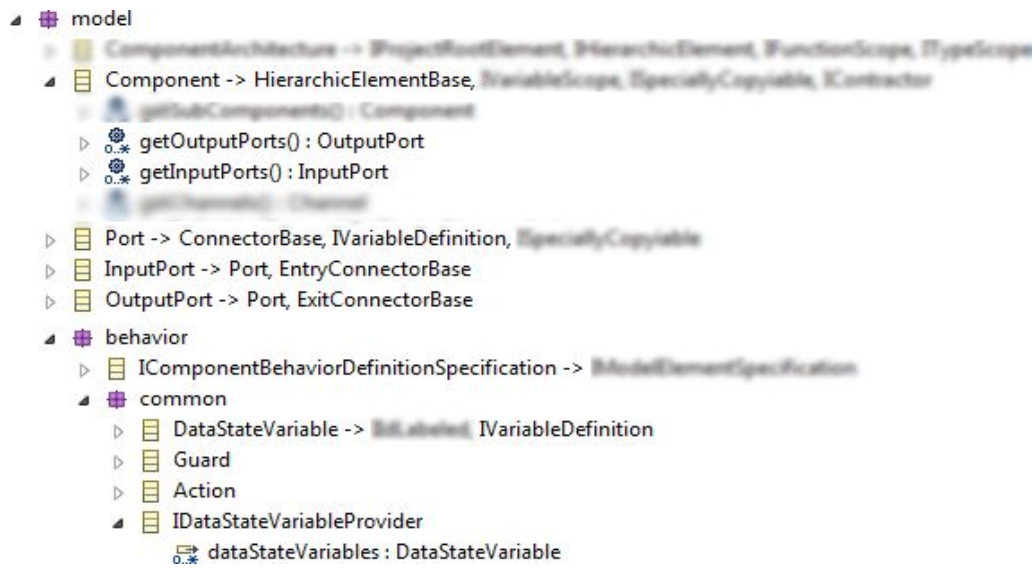

Figure 6

Figure 6 shows the component metamodel corresponding to the component package shown in Figure 3. *Observe that, contrarily to the conceptual metamodel,* `Component` *does not contain any attributes or references:* in particular, it does not contain any port or state automaton. This is compensated by the fact that `Component` inherits from "`HierachicElementBase`" (shown in Figure 7). `HierachicElementBase` is simply the canonical implementation of the `IHierarchicElement` interface, which does contains "`connectors`": this is the place where ports are actually "stored" in the metamodel; indeed `Port` inherits from `ConnectorBase`, which itself implements `IConnector`, and can therefore be added to the "`connectors`" field. *This sort of mechanism, even though counter-intuitive is very common in the metamodel of AF3 and is at the heart of the differences between the conceptual and real metamodel. The main reason for is is to facilitate the reuse of GUI elements among model elements.*

`Port` implements `IVariableDefinition` (defined in the typesystem). Similar to the conceptual metamodel input and output ports inherit from port. It can be seen that the ports are not contained in the `Component` class. They are actually present through the inheritance mechanism of AF3. We also developped facility functions `getOutPutPorts()` and `getInputPorts()` which allow to access these fields, as seen in the figure. The metamodel also contains the behavior of the component. It contains the interface `IComponentBehaviorDefinitionSpecification` which connects to the component implementation. It also contains the package common which defines data state variables, guard and actions which correspond the elements with same name in the state automaton package in the conceptual metamodel. It also defines the interface `IDataStateVariableProvider` which contains data state variables.

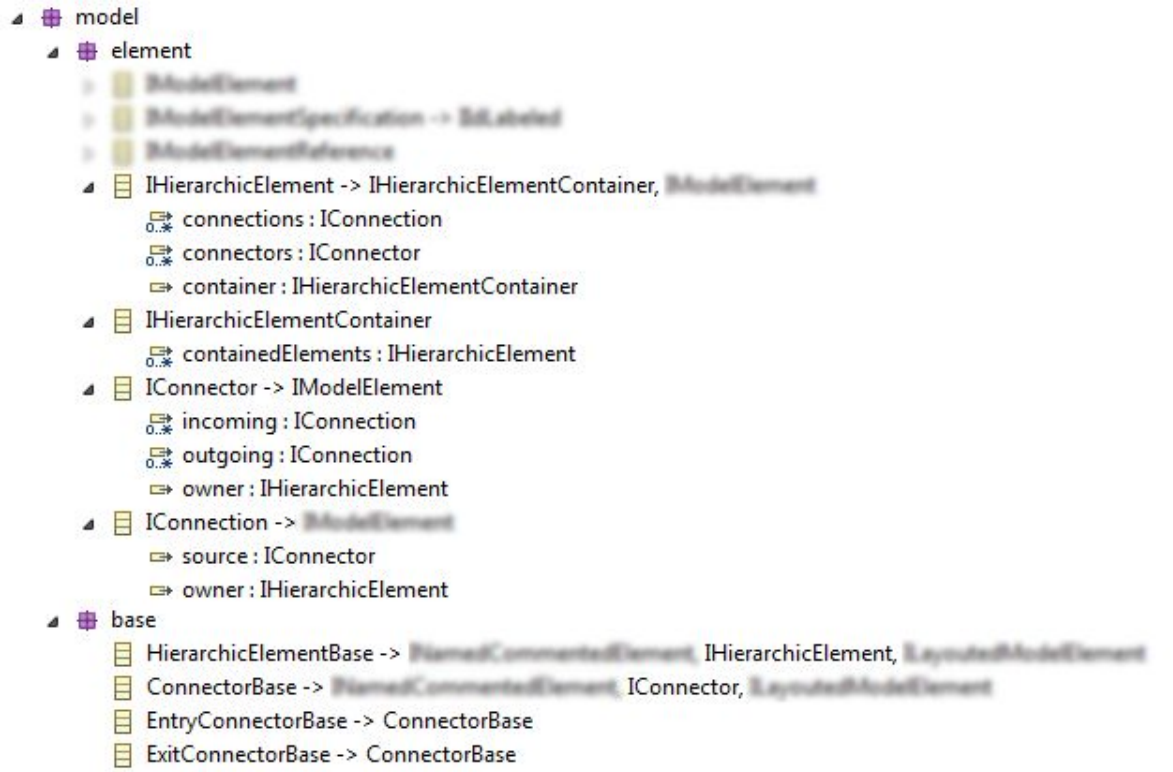

Figure 7

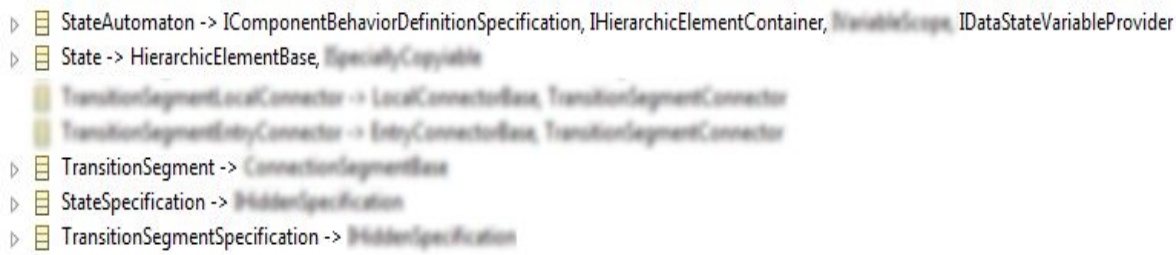

Figure 8

Figure 8 shows the metamodel for state automaton. The class `StateAutomaton` implements `IComponentBehaviorDefinitionSpecification`, `IHierarchicElementContainer` and `IDataStateVariableProvider` which have been described in the above paragraph. It also defines state, transition segment and transition segment specification corresponding to the elements with same name in the state automaton package in the conceptual metamodel. Additionally a class named `StateSpecification` is also defined which specifies if the state is initial and its idle actions.

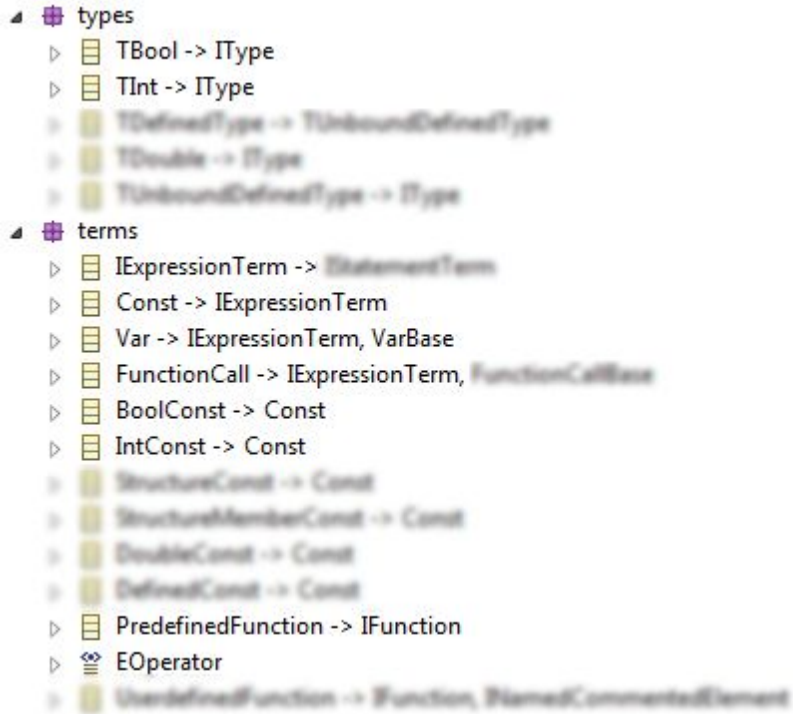

Figure 9

Figure 9 shows the metamodel for the expressions corresponding to the expression package in the conceptual metamodel. Types `TBool` and `TInt` were bundled in an enumeration in the conceptual metamodel, but here both implement the abstract interface `IType` (defined in the type system). Both boolean and integer constants inherit from the class `Const` which is an abstract class for the constants. Class `Var` additionally inherits from `VarBase` (defined in the type system). The class `Operation` in the conceptual metamodel is a simplification of `FunctionCall` and `PredefinedFunction`. `PredefinedFunction` implements `IFunction`(defined in the typesystem) and is used to model operations while `FunctionCall` models the instantiation of the operations.

2.1.2 NuSMV/nuXmv Metamodel

In this section we first present an example of NuSMV/nuXmv input file corresponding to the previous AF3 example model, then we present the metamodel corresponding to such files.

2.1.2.1 NuSMV/nuXmv input example

The NuSMV/nuXmv model corresponding to the example given above is presented in figure 10. It shows a parameterized module "`Component___6`" which defines few variables, assigns initial values to them and specifies the transitions of a finite state machine. It also specifies the bounds on variables `o` and `v` in terms of invariants.

Things to note:

- NuSMV/nuXmv doe not have the concept of `NoVal`. The current transformation deals with it by adding a boolean variable which indicates the presence of the port. It the example the module parameter `i_present` and variable `o_present` are these boolean variables.
- Word constants: The expressions of the format *0sd10_x* can be seen in the example at multiple places. *0sd10_x* denotes a signed (*s*) 10 bit word constant which is represented in decimal (*d*). The value x denotes the value of the constant.
- Values are assigned to the words after checking bounds.

```
MODULE Component___6(i, i_present)

VAR
             o : signed word[10];
             o_present : boolean;
             _current_state : { Init, s };
             v : signed word[10];

ASSIGN
init(o_present) := FALSE;
init(_current_state) := Init;
init(v) := (0sd10_0<(0sd10_0) ? -0sd10_1 : ((0sd10_0 > 0sd10_254) ? 0sd10_255 :
0sd10_0));

TRANS ((_current_state = Init) & i_present) &
       next(_current_state) = s &
       next(v) = ((i < 0sd10_0) ? -0sd10_1 : ((i > 0sd10_254) ? 0sd10_255 : i)) &
       next(o) = o & next(o_present) = FALSE
|((_current_state = Init) & !(i_present)) &
       next(_current_state) = Init & next(o_present) = FALSE &
       next(o) = o & next(v) = v
|((_current_state = s) & !(i_present)) &
       next(_current_state) = Init &
       next(o) = ((v < -0sd10_255) ? -0sd10_256 : ((v > 0sd10_254) ? 0sd10_255 :
v))&   next(o_present) = TRUE & next(v) = v
|((_current_state = s) & i_present) &   next(_current_state) = s   & next(v) = (((v
+ i) < 0sd10_0) ? -0sd10_1 : (((v + i) > 0sd10_254) ? 0sd10_255 : (v + i))) &
             next(o) = ((v < -0sd10_255) ? -0sd10_256 : ((v > 0sd10_254) ?
0sd10_255 : v)) & next(o_present) = TRUE
| (((!((_current_state = Init) & i_present)) & !((_current_state = Init) &
!(i_present)))) & !((_current_state = s) & !(i_present)))) & !((_current_state =
s) & i_present)) &
       next(o) = o & next(o_present) = FALSE &
       next(_current_state) = _current_state & next(v) = v;

INVAR ((o >= -0sd10_256) & (o <= 0sd10_255))
INVAR ((v >= -0sd10_1) & (v <= 0sd10_255))
```

Figure 10

We now give a brief description of the subset of the NuSMV/nuXmv language sufficient for our use case. For more details please refer to [3]:

- Types: The AF3 models in the scope of this case study only require boolean, word and enumeration types.
- Expressions: Expressions are typed in NuSMV/nuXmv. This case study requires following subset from the NuSMV/nuXmv expressions:
    - Constant expressions: boolean and integer constants
    - Basic expressions: a subset of the operations allowed in the NuSMV/nuXmv, e.g. addition, subtraction, negation, etc
    - Next expressions: used to express transition in the finite state machine relating variables in the current and next state.
- Finite state machine: Described in terms of state variables, input variables, frozen variables(which may have different values in different states), and transition relation.
- Module: The notion of modules can be divided into module declaration and module instantiation. A module declaration is an encapsulated collection of declarations, constraints and specifications. Modules are instantiated using **VAR** declarations. Each instance of a module refers to a different data structure. A module can contain instances of other modules, allowing a structural hierarchy to be built [3].
- VAR: Used to declare variables. Variables can be input, states and frozen variables.
- INIT constraint: It is used to specify the set of initial states with the help of a boolean expression.
- INVAR Constraint: It is used to specify the set of invariant states with the help of a boolean expression.

- ASSIGN: It is used for assignments of variables. In case right hand side evaluates to a set then the expression means that lhs is contained in rhs. It could be used to assign initial values, invariants and next values.

2.1.2.2 Ecore metamodel for NuSMV/nuXmv input

In this section we present the ecore metamodel for the NuSMV/nuXmv input.

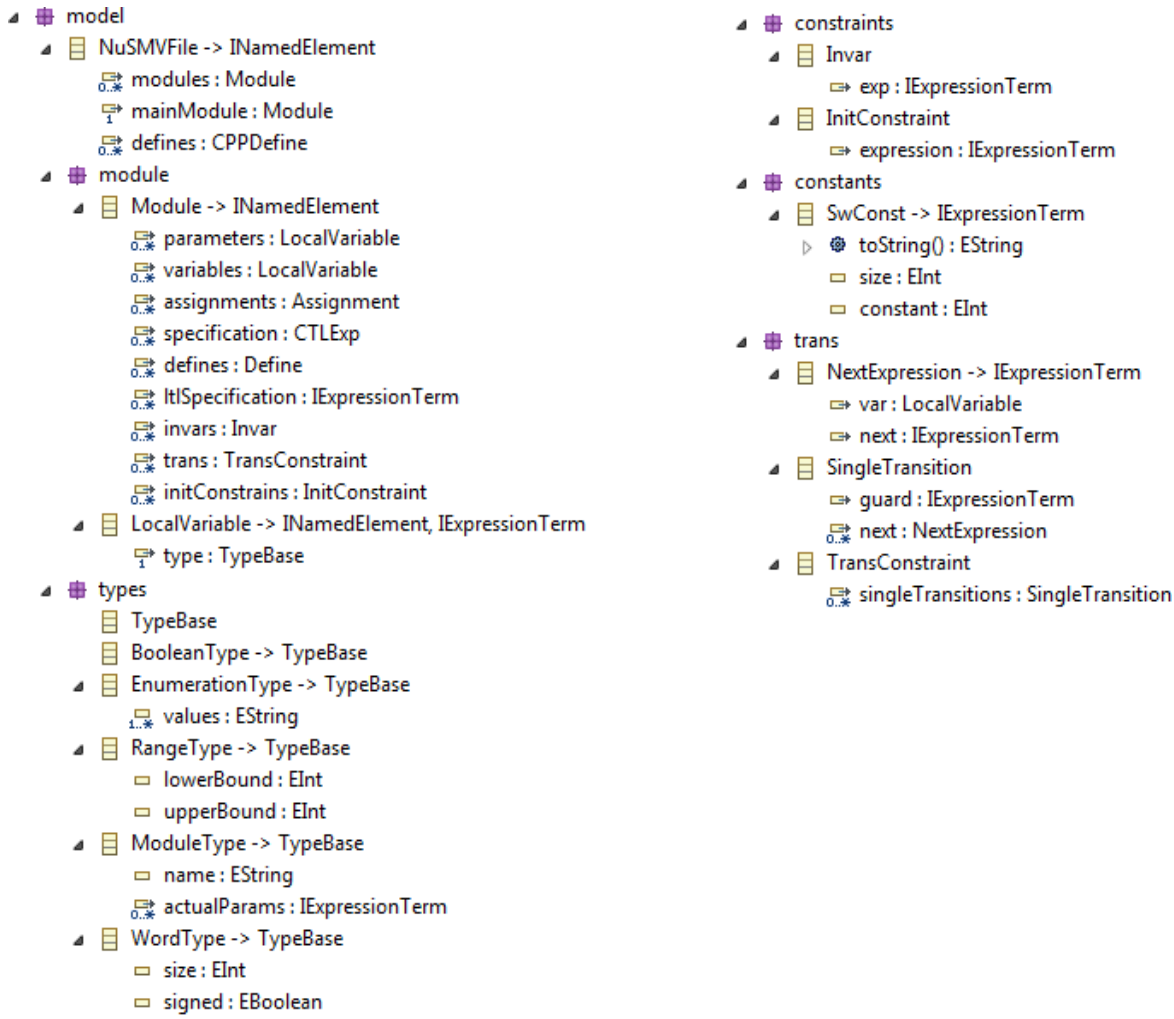

Figure 11

Figure 11 shows the metamodel NuSMV/nuXmv files. It contains a class NuSMVFile and packages module, types, operators, constraints, constants and trans. It contains few other packages as well but those are not required for this case study. Following is the description of some of the elements of this model:

- `NuSMVFile`: This class is finally transformed to text and given as an input to NuSMV/nuXmv. It contains the list of modules and a main module.
- `Module`: This class corresponds to the **MODULE** is NuSMV/nuXmv. It contains list of parameters, variables, assignments, transitions, invariants, etc.
- types package: this package contains types. For the case study only boolean, words and enumeration are important.
- constraints package:This package defines the classes for INVAR and INIT constraints mentioned above.
- constants package: This package defines the class for the word constant.
- trans package: this package contains following three classes:
    - `NextExpression`: contains the variable and the expression to be assigned to it in the next transition.
    - `SingleTransition`: contains the guard for the transition and a list of next expressions.
    - `TransConstraint`: consists of a list of single transitions.

2.1.3 Transformation

We now outline a few key points of the transformation:

- A component is transformed into a **MODULE** in NuSMV/nuXmv
- Input ports are passed as parameters, and for each port a boolean parameter is added which specifies if the port is present or not (indicating the presence of the signal in AF3).
- Output ports are defined as a variables in the module, and as in the case input ports, a boolean variable is added for each output port specifying if the port is present or not.
- Internal variables of the state automaton(called data state variables in AF3) are also defined as variables of the module.
- A variable indicating the current state of the automaton is also defined in the module.
- Variables are initialized based on the initial values in the component.
- Integer ports and data state variables are transformed to words. For this case study the size of the words can be used as 32.
- Appropriate bounds need to be added for the word types so that overflow is checked.
- Transition of the automata are transformed to TransConstraint.
- Each transition is transformed to SingleTransition, with the use of NextExpression for assignment of variables in the next "clock tick".
- Integer constants are transformed to SwConst.
- Main module also needs to be generated which simple instantiates the module(generated from component transformation).

There are few points to note though:

- The text file which could be used as NuSMV/nuXmv input **need not** be generated. It is **sufficient** if a model is generated conforming to the metamodel given in Figure 10.
- The transformation presented so far is actually not enough to get a fully usable NuSMV/nuXmv file (in particular, we are not focussing at all in this case study on the temporal logic specifications themselves, which should be normally provided to NuSMV/nuXmv).
- The translation of boolean expressions can be done up to logical equivalence: e.g., it does not matter whether *!(!flag)* is generated instead of simply *flag*.

2.2 NuSMV/nuXmv to AF3

So far, we described only one of the two mentioned transformations: the one transforming AF3 models to NuSMV/nuXmv traces. In this section we describe the "reverse" transformation turning NuSMV/nuXmv counterexamples into simulatable AF3 traces.

2.2.1 Example

Figure 12 shows the counterexample ouput by NuSMV/nuXmv in case of specification failure. Here `Component___8Var` refers to an instantiated module. Variable are in the form of C.x where C refers to the instantiated module and x refers to the variable or parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<counter-example>
        <node>
                <state id="1">
                        <value variable="Component___8Var.o___22">FALSE</value>
                        <value variable="Component___8Var._current_state">Init___15</value>
                </state>
        </node>
        <node>
                <state id="2">
                        <value variable="Component___8Var.o___22">TRUE</value>
                        <value variable="Component___8Var._current_state">Init___15</value>
                </state>
        </node>
</counter-example>
```

Figure 12

2.2.2 Metamodel

Once again, we do not expect the transformation to work at the text level. Instead we developed a metamodel corresponding to the XML format above. This metamodel is shown in the left part of Figure 13.

The **run** package describes the metamodel for the counterexample returned by NuSMV/nuXmv. It contains following classes:

- `NuSMVResult`: This class contains a list of counterexample states. Remaining fields **need to be ignored** for the transformation
- `CounterExampleEntry`: This is a pair of a strings where one string refers to a variable and second to its value.
- `CounterExampleState`: It is a list of counter example entries.

Figure 13

Figure 13 also shows the metamodel for the target model, on the right.

The **model** package describes the metamodel for the simulation trace. It contains:

- `Trace`: A trace contains a component (which was initially transformed) and a list of trace entries.
- `TraceEntry`: `Trace` entry contains the state of the automata and a list of value pairs.
- `ValuePair`: It is a pair of `IVariableDefinition` which refers to either port or data state variable and `IExpressionTerm` which refers to value.

2.2.3 Transformation

This transformation on the surface seems straightforward, but it is interesting for the following reasons:

- This transformation needs to be aware of the first transformation (AF3 to NuSMV/nuXmv) to generate the correct output. For this reason the component is included in the target model. Specifically, string parsing (of variable names in NuSMV/nuXmv counterexample) is **prohibited** to find the ports/variables in the components (refer to section 3.2 for more details).
- The output of this transformation enables the counterexample to be simulated in AF3. This point might not be of interest with respect to the transformation itself, but is an essential feature required in AF3.

Now we give mapping between the source and target meta models.

1. `NuSMVResult` → `Trace`
2. `CounterExampleState` → `TraceEntry`: One `CounterExampleEntry` in the `CounterExampleState` corresponds to the state of the automata, this needs to be transformed to State in the `TraceEntry`.
3. `CounterExampleEntry` → `ValuePair`

3 Evaluation

3.1 What is “software-engineering-friendly”?

As mentioned in introduction, the key evaluation criterion in this case study, is the “software-engineering-friendliness” of the solution. More precisely, consider Figure 14 which represents the *time* evolution of a transformation during its development:

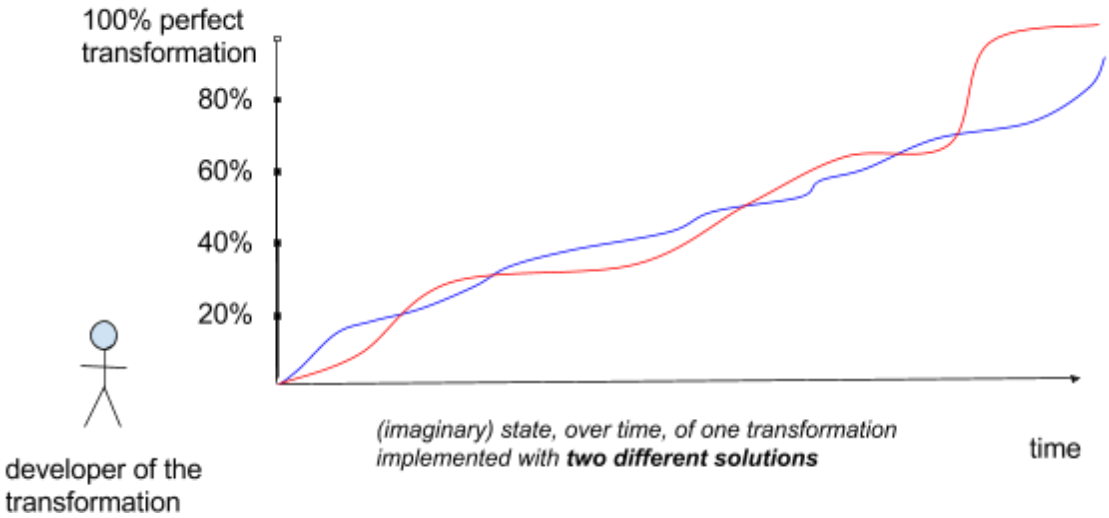

Figure 14

By "software-engineering-friendly" we mean a solution which minimizes the time to get to the 100% solution. In the figure above, the red transformation would probably be considered better. Actually, in our case, since we have two transformations, two transformations shall be considered at a time, see Figure 15:

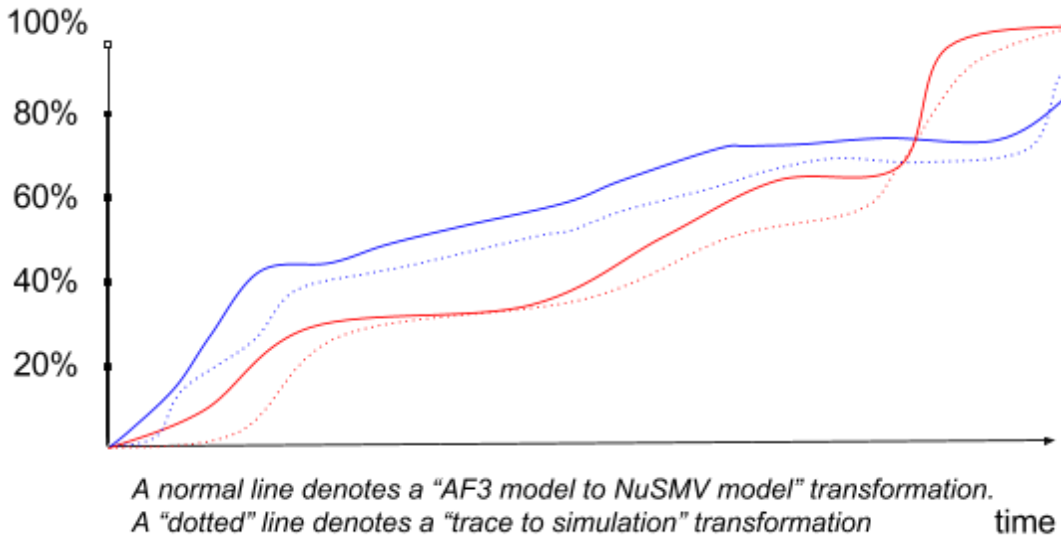

Figure 15

Of course, in practice, as the figure above shows, not because a solution is better at one point means that it is always the best. Furthermore the percentage achieved by the transformation is generally not measurable (the 100% is often even not known in advance). Finally, we do not know in advance the changes which will be required in the future, therefore a solution which is the best so far might not be the best for the future increments. Therefore we propose to evaluate the solutions by providing example models for various evolution points (an example model for 20%, one for 40%, etc.) which *both transformations should reach* and by considering the "deltas" between these intermediate transformations.

Every delta can be measured simply by the "diff" between one version of the pair of transformations to the next one. Since this is very language-dependent, this will be qualitatively evaluated by the reviewer. The underlying idea is of course that a change which impacts both transformations should make maximal reuse of what was developed for one transformation, or even if the common part is separately defined.

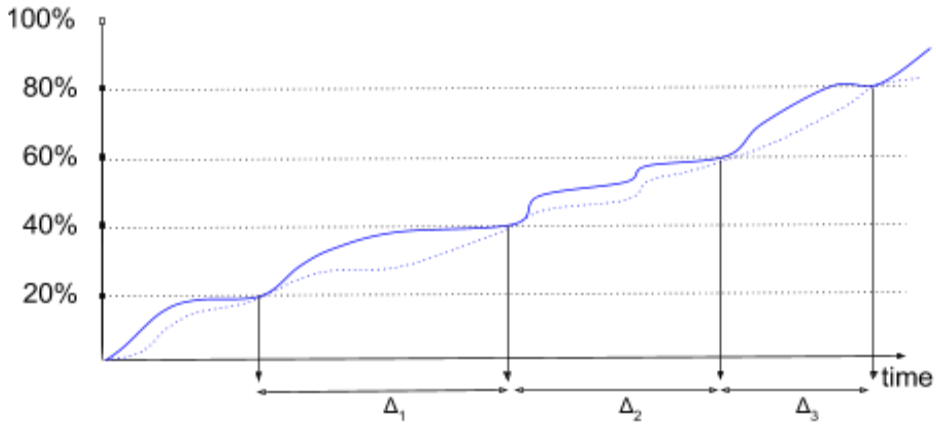

Figure 16

3.2 Concrete example of a non software-engineering-friendly pattern

In this section, we provide a concrete example of a pattern that we observed to be non "software-engineering-friendly" (and which our current implementation unfortunately relies on).

Consider the translation of AF3 ports to NuSMV/nuXmv "ports" (actually parameters of a module). As mentioned before, an essential difference between both is that AF3 ports all can contain a special value called "NoVal" (just like all objects in Java can be null for instance), but NuSMV/nuXmv does not have such a notion. This is dealt with by transforming every AF3 port into **two** NuSMV/nuXmv parameters: one which is intended to contain the value carried by the message, and one which is a boolean value indicating whether there is indeed a message or not (in which case the value carried by the message is actually meaningless). In the current transformation, the first parameter gets the same name as the original port, and the second one gets the same name appended with "_PRESENT" (e.g., for a port "p": "p_PRESENT"). This is then used during the reverse transformation: if a string terminates with "_PRESENT" it is considered to be the counterpart of a port. This is not an acceptable solution: first if the string changes in the future, both transformations have to be changed; second, this is not a perfect method: some names might actually terminate with "_PRESENT" just because the AF3 user decided to name their port like this; how is it possible then to distinguish this case?

Unfortunately, the current transformation implemented in AF3 follows this approach, which entails many practical problems. Ideally, the transformation should be able to detect what is the originating element, independently of its shape, but by re-using aspects of the first transformation.

This situation is recurrent:

- even though not present in the models of this use case, AF3 datatypes allow structures (or "records" in C) which NuSMV/nuXmv does not. It is therefore necessary to "flatten" them when transforming to NuSMV/nuXmv. For instance a port "p" whose type is a structure with two members "x" and "y" would be flattened into "p_x" and "p_y" (and "p_PRESENT" due to the previous transformation). The reverse transformation then makes use of the "_x" suffix to infer that it comes from the flattening of a structure, but this is actually not robust.
- even though not present in the models of this use case, AF3 allows hierarchic components whose subcomponents need to be flattened, using again similar assumptions.

3.3 Systematic Evaluation

Concretely, we provide the following increments:

1. input component containing a simple state automaton with only one boolean output port
2. same input component but the output of the transformation should encode the state as an integer instead of an enumeration
3. input component containing in addition a boolean input port
4. same input component but the output of the transformation should encode the booleans as integers
5. the ports can receive the special "NoVal" value which encodes the absence of message
6. input component containing an integer output port (in addition to the previous boolean output port)
7. input component containing an integer input port (in addition to the previous boolean input port)
8. state automaton of the input component makes use of data state variables

Each increment contains the following:

- "one-way" transformation: original AF3 Model

- "one-way" transformation: expected NuSMV/nuXmv Model
- "reverse" transformation: original NuSMV/nuXmv trace
- "reverse" transformation: expected AF3 Simulation trace

For every increment, a transformation shall be provided which extends the previous one. For every increment, the transformation is graded as follows:

- 2 points if the one-way transformation is correct, 1 point if it is generally working but has some bugs
- 2 points if the *reverse* transformation is correct, 1 point if it is generally working but has some bugs

In addition, for every increment but the first one, a grade between 0 and 3 points is given for the "software-engineering friendliness" of the transformation w.r.t. the previous increment: i.e., the reviewer asses if the delta is small enough.

- 0 = both increments are completely separate
- 1 = superficial reuse exists between both transformation increments
- 2 = good reuse exists between both transformation increments
- 3 = the reviewer does not see any possibly better reuse between both transformation increments

The last increment (with "NoVal") receives double weight for this latter note.

Note: just like in reality, it might be that some of the increments do not offer any possibility for reuse.

The scores are summarized in Table 1.

Increment	Correctness of the one-way transformation	Correctness of the reverse transformation	Software-engineering-friendliness
1	2 points	2 points	n.a.
2	2 points	2 points	3 points
3	2 points	2 points	3 points
4	2 points	2 points	3 points
5	2 points	2 points	3 points
6	2 points	2 points	3 points
7	2 points	2 points	3 points
8	2 points	2 points	3 points - weight: 2
Total: 59 points			

Table 1

Repository URL

The repository is hosted at https://github.com/skanav/ttc2016
This repository contains:

- Models to be used for systematic evaluation as described in section 3.2
- Address of AF3 repository which is public
- Conceptual metamodel

References

[1] Broy, Manfred, and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[2] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., et al. (2014). The nuXmv symbolic model checker. *Computer Aided Verification*. Springer International Publishing.

[3] nuXmv user manual, https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf