Modular transformation from AF3 to nuXmv

Sudeep Kanav, Vincent Aravantinos
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany
Email: {last_name}@fortiss.org

***Abstract*—AutoFOCUS3 (AF3) [1] supports formal verification of its models using the nuXmv [2] model checker. This requires a model transformation from AF3 to nuXmv models. In this paper we present this behavior transformation. It is a two way transformation between a high-level and a low-level model involving intricate cases typical of behavior transformations whose solutions can therefore be beneficial to the community.**

I. Introduction

Model driven Engineering (MDE) allows us to model the system: its requirements, behavior, target platform, and generate code out of this model which can be deployed on the system. The behavior model of the system can be used for early verification, thus detecting bugs at an early stage of development, thereby saving time and money.

One way to perform verification is using formal methods, which are useful when we need high assurance, but are hard to use. They are however difficult to integrate with MDE tools because it requires a behavior transformation from a high-level MDE model to a low-level model of the formal analysis tool.

AutoFOCUS3 (AF3) [1], is an MDE tool which supports formal verification of its models. We use the nuXmv [2] (successor of NuSMV [3]) model checker for formal analyses. We transform AF3 models to nuXmv models, run model checking on them, interpret the result, and simulate the model checker trace, if available.

In this paper we present the details of this model transformation. We believe this is useful to the community because it points out pitfalls that one typically encounters while developing a bi-directional transformation from a high-level to a low-level model and shows how to circumvent them. Most importantly, we designed the transformation to be modular, extensible and to maximize reuse: the reuse aspect is particularly put to practice with the reverse transformation.

This work deals with 1) transformation between a high level model and a low level model, 2) two way transformation: from AF3 to nuXmv and vice versa, 3) transformation of a high level model which can model both events and messages. Note finally that we have applied this solution to an industrial use case.

This transformation is a complete rebuild of the one informally presented in [4] covering a larger fragment, fixing important limitations of the previous transformation and designed in a modular manner.

The paper is structured as follows: Section II presents AF3 and nuXmv, Section III describes our transformation, Section IV describes a use case, Section V considers the related work, and Section VI concludes this paper.

II. Background

In this section we give an overview of AF3 [1] and provide a brief introduction to the concepts needed to understand the paper. We then give an introduction to nuXmv [2], and lastly we list down the challenges which make this solution useful and non trivial.

A. AutoFOCUS3 introduction

AutoFOCUS3 [1] is a model-based development tool for embedded systems, which provides support for most development phases: requirements engineering, software architecture, hardware architecture, as well as mapping between the artifacts of these various phases. C and Java code can be generated from the developed model after the architecture and behavior are fully defined.

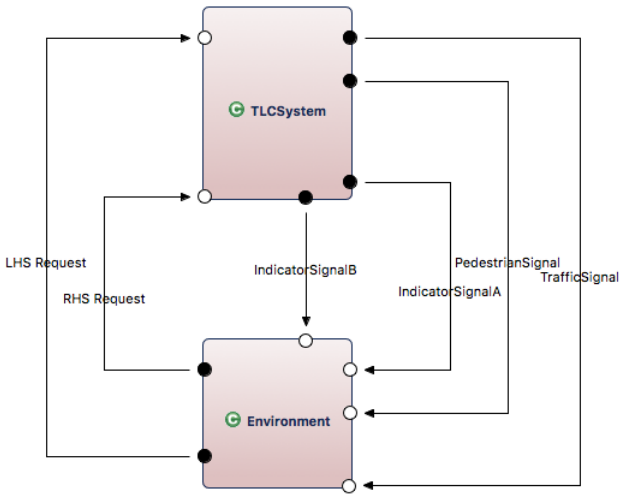

Fig. 1. Component architecture in AutoFOCUS3

This paper concerns the *software architecture*, whose following characteristics are essential for formal verification:

- *Component-based*: Software architecture in AF3 is a hierarchy of components communicating with each other through typed *channels* (Fig. 1) which are connected

to ports. Ports, which can be input or output, form the interface of the component. Each port carries values of a defined type (no objects: only booleans, integers, doubles, enumerations, or combinations thereof). Channels in AF3 are statically fixed.
Atomic components, if implemented, contain a behavior specification modeled by a state machine or code expressed in a very simple language.[1]

- *Formal execution semantics*: The semantics of components is formally defined according to the FOCUS model of computation [5], i.e., a global clock is assumed and all components are (in a nutshell) synchronously executed. One can then simulate their component architecture in AF3 according to these semantics.

B. nuXmv

nuXmv [2] is a symbolic model checker. We now describe the features which we use for the formal verification of an AF3 component.

In nuXmv, a system is modeled as a finite state machine (FSM) described in terms of variables and constraints. Variables, which can be state variables or input variables, can have different values in different *states*. Constraints are used to describe 1) transition relations: how a state machine evolves during execution, 2) initial state, 3) invariants. It also supports hierarchical descriptions.

Three types of specifications are supported: LTL, CTL, and INVAR (invariants). The definition of the FSM, specifications, and declarations (input variables, state variables, constants, and define) are encapsulated in a module declaration.

A module can be then instantiated inside another module. Every nuXmv program must contain a *main* module. The *main* module does not have any parameters and is the entry point of the nuXmv model.

C. A non-trivial transformation

The transformation from AF3 to nuXmv models is not trivial as it transforms a model at a high level of abstraction to one at a lower level of abstraction. We now describe some AF3-specific notions which make the transformation challenging.

NoVal: AF3 allows to model situations where a channel is not carrying a value. This is required when modeling events, e.g., a user pressing the brake in a car. We could also see it as a *null* value. This situation is modeled using a special constant called *NoVal* which belongs to every *port* type and symbolizes the absence of an event in the current tick. This is problematic because a *boolean* type port in AF3 can then have three values: $true, false$, and $NoVal$, whereas in the model checker a boolean value can be only $true$ or $false$. This is analogously true for ports of other types.

Causality: AF3 allows to specify the causality of a component as *strong* or *weak*. In a strongly causal components output ports are updated in the next clock tick, whereas in a weakly causal component output ports are updated in the same tick.

The strongly causal components can be seen as Moore machines: the output of the current tick is only dependent on the state and not on the input, input can only influence the output of the next tick (we can also see it as input only changing the state, which in turn affects the output). Weakly causal components can be interpreted as Mealy machines: output is dependent on both the state and the input.

As nuXmv does not have the notion of strong and weak causality this needs to be additionally handled.

User defined functions: AF3 also supports the use of functions defined by the user in a *Data Dictionary*. These functions are stateless and use the same simple language which can be used to define a component's behavior. These are not part of the transformed component's tree, but are contained in a separate subtree. Note that we do not describe this transformation in the paper.

III. Transformation description

We decompose the transformation in small step transformations. These transformations are then executed sequentially: the output of the first transformation is the input of the next. We denote the sequential composition by ";": applying $T_1; T_2$ on I means that first T_1 is applied on I, and then T_2 is applied on its output.

This decomposition allows the small step transformations (or blocks) to be reused in different contexts, e.g., for the reverse transformation (which was not possible in [4]). The transformations are implemented in Java.

As a running example, we consider a component C, with input port ip which is a boolean array of size 2, and a boolean output port op which is true if both values are equal in the input array. The behavior of the component is specified using a code specification:

```
if(ip[0]==ip[1])
  { op := true;}
```

It is a strongly causal component, which means that the output values are updated on the next tick.

At the top level, transformation of an AF3 model to a nuXmv model $\mathcal{T}$ is a chain of 2 transformations 1) normalization of the model, 2) transformation to nuXmv:

$$\mathcal{T} = \mathcal{T}_{Norm}; \mathcal{T}_{nuXmv} \quad (1)$$

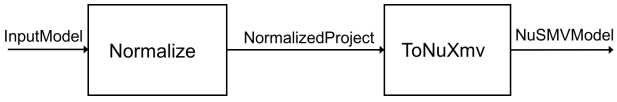

Fig. 2. Transformation from AF3 to nuXmv model

Fig 2 shows this chain in a pictorial form. From now on we describe chains using only the ";" operator.

[1]Note that behavior can also be expressed using tables, but this feature is deprecated at the moment.

A. Transformation normalizing the project

This transformation takes an AF3 model and transforms it into a normalized AF3 model, which is easier to transform to a nuXmv model. It is a chain of 5 transformations:

$$\mathcal{T}_{Normalize} = \mathcal{T}_{Names}; \mathcal{T}_{CodeSpec}; \mathcal{T}_{NoVal} ; \mathcal{T}_{ProductTypes}; \mathcal{T}_{Causality} \quad (2)$$

Note that (1) and (2) will be reused in Section III-C.

1) Name transformation: $\mathcal{T}_{Name}$ changes the names of the artifacts by appending their *internal id*, and removing white spaces and special characters. The references to these elements are also transformed. The example is then transformed to:

```
if(ip_40[0]==ip_40[1])
  { op_24 := true;}
```

2) Code specification to state automaton: $\mathcal{T}_{CodeSpec}$ turns a behavior of an atomic component, in case it is specified as a code specification, into a state automaton.

A code specification in AF3 is stateless, so it is transformed to an automaton with a single state, where each *if* (and *else*) block is converted to a transition. $\mathcal{T}_{CodeSpec}$ is again a chain of 2 transformations:

$$\mathcal{T}_{CodeSpec} = \mathcal{T}_{NormalizeCodeSpec}; \mathcal{T}_{ToAutomaton}$$

$\mathcal{T}_{NormalizeCodeSpec}$ outputs a code specification which is behaviorally equivalent to the input and in a normalized form from which we can do a canonical transformation to a state automaton, and $\mathcal{T}_{ToAutomaton}$ performs this canonical transformation.

Applying $\mathcal{T}_{NormalizeCodeSpec}$ on the above code specification results in the following code specification:

```
if(ip_40[0]==ip_40[1]){
  op_24 := true;
  return;}
else{
  return;}
```

Note the addition of the empty *else* block modeled to obtain a canonical transformation to an automaton.

Fig. 3 shows the result of applying $\mathcal{T}_{ToAutomaton}$ to the above code specification. Note that the transition for the else block (lower half in the figure) has no action. The semantics of this automaton and the initial code specification are equivalent.

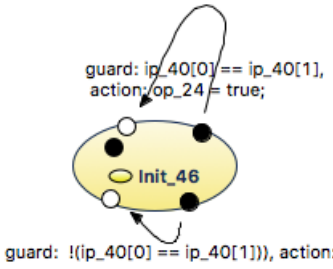

Fig. 3. Result of applying $\mathcal{T}_{ToAutomaton}$

The output of this transformation satisfies the property: *All the components are specified using a state automaton.* In case the input already satisfies this property the transformation behaves as the identity transformation.

3) NoVal resolution: As described in Section II-C, AF3 ports can have one more value: *NoVal*, than the corresponding type in the model checker.

To take this discrepancy into account, for each port p we add a boolean port $p\_PRESENT$ whose value denotes whether the port is carrying a value or not. This also requires that we change the behavior of the state automaton. This transformation is also a chain of 2 transformations:

$$\mathcal{T}_{NoVal} = \mathcal{T}_{UpdateTransitions}; \mathcal{T}_{ResolveNoVal}$$

$T_{UpdateTransitions}$: If a port is not explicitly assigned a value in a transition it should carry *NoVal*. This transformation makes this step explicit, i.e., we transform the transitions to explicitly assign *NoVal* to ports which are not assigned a value.

$T_{ResolveNoVal}$: This transformation adds the *PRESENT* port and updates the transitions such that for every port p which is assigned a value, $p\_PRESENT$ is assigned *true*, otherwise $p\_PRESENT$ is assigned *false* (in this case p is assigned a default value of the type). The guards of the transitions are also transformed accordingly. This is done to ensure that after this transformation no *NoVal* can flow through the model.

The absence of a value is interpreted using the value of the *PRESENT* port.

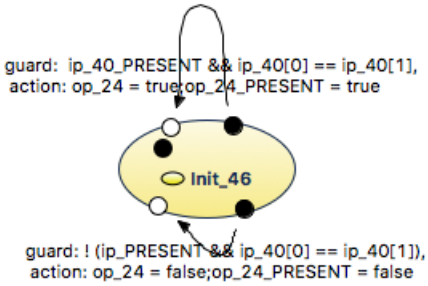

Fig. 4. Result of applying $\mathcal{T}_{NoVal}$

Fig. 4 shows the result of applying $\mathcal{T}_{NoVal}$ on the state automaton in Fig. 3.

The output of this transformation satisfies the property: *Every channel carries a value.*

4) Product types to simple types: This transformation converts arrays and structures to simple types and correspondingly transforms the variable definitions and expressions.

This transformation is a chain of two transformations : 1) T_{Array}: transforms arrays to structures, 2) $T_{Flatten}$: flattening of structures and their expressions.

$$\mathcal{T}_{ProductTypes} = \mathcal{T}_{Array}; \mathcal{T}_{Flatten}$$

T_{Array} is again a chain of two transformations:

1) $\mathcal{T}_{ToStructType}$: transforms array types to structure types by creating a structure member for each element of the array. This transformation also changes the type of the

array variables to the corresponding structure type. It is to be noted that after this transformation the model could be inconsistent, as the expressions involving the ports would still be array expressions but the port type has changed to the *Structure* type.

2) $\mathcal{T}_{ToStructExpression}$: transforms array expressions to structure expressions. It converts an array access to a structure access, and also transforms the array constants to structure constants.

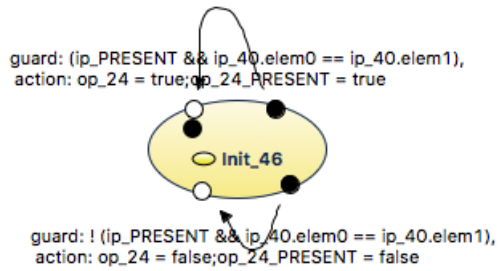

Fig. 5. Result of applying $\mathcal{T}_{Array}$

Fig. 5 shows the result of applying T_{Array} on the automaton in Fig. 4.

The output of this transformation satisfies the following property: *The model does not contain arrays.*

$\mathcal{T}_{Flatten}$ flattens the structures, i.e., 1) creates a variable definition for each member of a structure variable definition, 2) converts the expressions involving structure variable to semantically equivalent flattened expressions. It is defined as a chain of three transformations:

1) $\mathcal{T}_{FlattenExpression}$: flattens the structure expressions, e.g., an expression $x == y$, will be converted to a conjunction of a list of equality expressions of the form $x.field_i == y.field_i$.
2) $\mathcal{T}_{ToNonStructExpressions}$: replaces the flattened expression with member access to the corresponding variable created for the structure member. The model is inconsistent after this transformation.
3) $\mathcal{T}_{flattenVariableDefinitions}$: flattens a variable definition, i.e., creates a variable for each member of a structure variable.

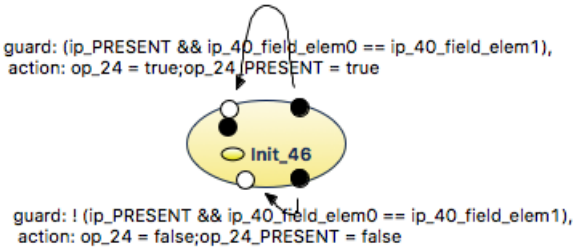

Fig. 6. Result of applying $\mathcal{T}_{Flatten}$

Fig. 6 shows the result of applying $\mathcal{T}_{Flatten}$ on the automaton in Fig. 5.

The output of this transformation satisfies the property: *The model does not contain structures.*

Note that the decomposition in modular transformations enables the handling of nested arrays and structures in an easy manner, a feature which was not supported in [4].

5) Causality transformation: As described in subsection II-C, AF3 supports two kinds of causalities: strong and weak. We convert the strongly causal components to weakly causal components and add a delay, which can be seen as transforming a Moore machine to a Mealy machine. This allows us to reuse the transformation of a weakly causal atomic component to the model checker.

The approach is as follows: 1) add a state variable for every output port, 2) assign each of these state variables the value which was assigned to the corresponding output port, and 3) assign the values of these state variables to output ports.

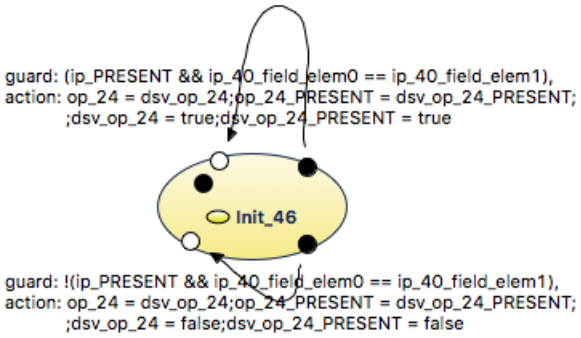

Fig. 7. Result of applying $\mathcal{T}_{Causality}$

Fig. 7 shows the result of applying $\mathcal{T}_{Causality}$ on the automaton in Fig. 6.

After the execution of this transformation all the components are weakly causal.

Once this chain of transformations is executed our AF3 model is considered normalized. It has the following properties:

1) All variable, component, transition, function, etc. names are unique.
2) All atomic components are specified as state automata.
3) All ports carry a value.
4) The model contains only simple types, i.e., no arrays or structures.
5) All components are weakly causal.

The normalized component can be transformed to a nuXmv module in an easier way. Note that it is still not a canonical transformation.

B. Transformation to NuXmv

The second top level transformation $\mathcal{T}_{nuXmv}$ transforms a normalized AF3 model to a nuXmv model. The first step generates a nuXmv model, and the second one is a model-to-text transformation which is straightforward and therefore not discussed in this paper.

AutoFOCUS3	NuXmv
Component	Module
InputPort	Module parameter
OutputPort	Define (a named expression)
Transition	State variable
State	*Not transformed explicitly*
Guard	Define (a named expression)
Action	1) transition ($TRANS$) constraint for state variable 2) case statement for output ports

TABLE I
MAPPING OF AF3 AND NUXMV ELEMENTS

Table I shows the mapping of AF3 elements to the nuXmv elements. Note that the transitions are transformed to state variables, and not the states.

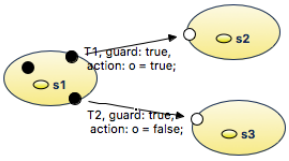

Fig. 8. Example state automaton

This transformation seems counterintuitive, but it was designed this way as the intuitive transformation is erroneous in the case of non-deterministic automata. As an example, consider a component with a boolean output port o and behavior expressed as the state automaton given in Fig. 8. There is a non-determinism between $T1$ and $T2$. Intuitively, this automaton would be transformed to:

```
DEFINE o := case
   (s = s1) & TRUE: TRUE; -- T1
   (s = s1) & TRUE: FALSE; -- T2
esac;

TRANS (s = s1 & TRUE) -> next(s) = s2;
TRANS (s = s1 & TRUE) -> next(s) = s3;
```

Here, although the next state of the automaton might be $s3$ (i.e., $T2$ is executed), o will be always assigned $TRUE$ (corresponding to the case $T1$) as it is the first satisfying case. This kind of description leads to outputs being updated incorrectly in case of non-determinism, which precisely was a bug in [4].

Modeling the chosen transition explicitly instead of inferring it from the change in *state* enables us to solve this issue. Instead of modeling the automaton as evolution of states, we model it as a sequence of transitions. Our transformation specifies the automaton in Fig. 8 as:

```
DEFINE o := case
    (t = T1) : TRUE; -- T1
    (t = T2) : FALSE; -- T2
esac;

TRANS t in {dummy_val}
   -> next(t) in {T1, T2};

INVAR (t = T1) -> true -- guard of T1
INVAR (t = T2) -> true -- guard of T2
```

Here, outputs are updated correctly depending on which transition is taken. Note that the transition is initialized with $dummy\_val$; it is necessary as there is no initial transition of a state automaton.

We now briefly describe the technicalities of this transformation. Let us consider a state automaton with states $\mathcal{S}$, transitions $\mathcal{T}$, where each transition consists of a guard and a set of actions $\{T \in \mathcal{T} = (T_{guard}, T_{actions})\}$, each action is of the form $v := e$. $S_{in} \subseteq \mathcal{T}$, and $S_{out} \subseteq \mathcal{T}$ denote the set of incoming and outgoing transitions for the state $S \in \mathcal{S}$. t is a nuXmv variable representing the transition chosen for execution. Now we describe the module specification.

We generate the following sets of transition constraints:

$$\{TRANS\ t \in S_{in} \rightarrow next(t) \in S_{out}\ | S \in \mathcal{S}\}$$

$$\{TRANS\ t = T \rightarrow execute(T_{actions}) |\ T : \mathcal{T}\}$$

where

$$execute(t_{actions}) \equiv \{next(v) := e \mid (v := e) \in T_{actions}\}$$

We generate the following set of invariants:

$$\{INVAR\ t = T \rightarrow T_{guard} \mid T : \mathcal{T}\}$$

An output port *op* transformed to a *define* *op*′ is assigned a value given by a case expression with cases

$$\{t = T \rightarrow T_{actions}[op'].value\}$$

where $T_{actions}[op'].value$ represents the value assigned to *op*′ in the transition actions for the transition T.

We also initialize the automaton with *INIT* constraint, with initial values of the state variables.

C. Counterexample transformation

The user needs to observe a trace returned by the model checker in AF3. A counterexample trace is a sequence of trace steps at the nuXmv level: each trace step assigns a value to every variable of the analysed nuXmv module. For usability, we need however to express this trace back in a sequence at the AF3 level: we should assign values to *AF3 ports*. Consequently, we should transform nuXmv variables back to AF3 variables. Intuitively, we should *reuse* the forward transformation to ensure that both transformations are always coherent. Both transformations go however in opposite directions, so it is not possible to use the same transformation trivially.

We actually need on one hand to transform AF3 *ports* into nuXmv variables (to reuse the transformation), and on the other hand to transform nuXmv *valuations* into AF3 ones (to lift the results back at the user level). To do so, we introduce a *symbolic* operator $[e]$ denoting the evaluation of an expression e in (a given step of) the trace. The *concrete* evaluation of this

operator is trivial for nuXmv variables, e.g., evaluating $[ip_{40}]$ is as simple as reading its value in the trace. Our objective is however to find out the concrete evaluation of this operator for *AF3 ports*, e.g., $[ip]$. We therefore need to express the evaluation of an AF3 port in terms of the (trivial) evaluation of a nuXmv variable. This can be achieved simply by applying the following subsequence of the original transformation to this epxression:

$$\mathcal{T}_{CE} = \mathcal{T}_{Name}; \mathcal{T}_{NoVal}; \mathcal{T}_{ProductTypes}$$

Table II shows how this expression for the input port ip of our running example is transformed by $\mathcal{T}_{CE}$.

Transformation	Result
-Input-	$[ip]$
$\mathcal{T}_{Name}$	$[ip\_40]$
$\mathcal{T}_{NoVal}$	$[ip\_40\_PRESENT]$? $[ip\_40]$: $NoVal$;
$\mathcal{T}_{ProductTypes}$	$[ip\_40\_PRESENT]$? $[[ip\_40\_elem0], [ip\_40\_elem1]]$: $NoVal$;

TABLE II
INTERMEDIATE RESULTS OF THE COUNTEREXAMPLE TRANSFORMATION

Note that not all transformations are necessary here, e.g., the transformation $\mathcal{T}_{CodeSpec}$ (Section III-A2) is irrelevant for the counterexample interpretations as this transformation deals only with representation of the component's behavior. We therefore do not use it in our counterexample transformation. However, the transformation $\mathcal{T}_{NoVal}$ (Section III-A3) changes the behavior of the component and affects how the port values are interpreted, and therefore is a part of the composition chain.

D. Specifications

We support LTL formulas in the form of verification patterns, and also LTL contracts in the form of assume-guarantee expressions. Here as well we reuse a part of the chain $\mathcal{T}$ (eqn. 1, 2) for transforming specifications.

$$\mathcal{T}_{Spec} = \mathcal{T}_{Names}; \mathcal{T}_{NoVal}; \mathcal{T}_{ProductTypes}; \mathcal{T}_{nuXmv}$$

As an example, a specification

$$Always\,(op \neq NoVal \;\&\&\; op)$$

is transformed to

$$G(op\_24\_PRESENT \;\&\; op\_24)$$

IV. USE CASE

As a case study we successfully ran various formal verification checks: state reachability, non-determinism, port bound violation checks, on a model of a pick and place unit (PPU)[2] [6] presented in Fig. 9. A PPU picks a work piece from one place and places it at another place, as shown in Fig. 9:

1) *Stack*: Input storage of work pieces.
2) *Ramp*: Output storage for work pieces.

[2]www.ppu-demonstrator.org

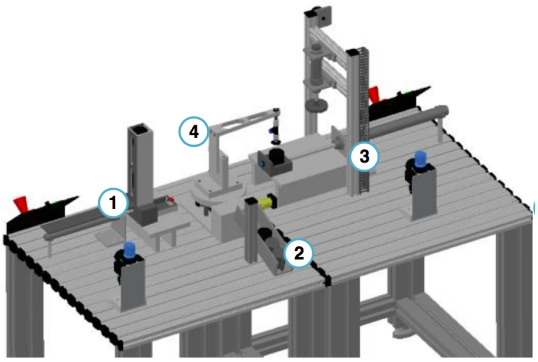

Fig. 9. Components of the PPU [6]

3) *Stamp*: Used to stamp work pieces.
4) *Crane*: Used for picking and placing work pieces between above 3 stations.

Work pieces can be metallic or black plastic. The PPU picks up the work piece from the stack and *processes* it. The crane picks up a work piece from the stack. If the work piece is black plastic it is transported to the ramp, if it is metallic then it is transported to the stamp: where it is stamped, and then the stamped piece is transported to the ramp by the crane.

The PPU model in AF3 is a big model which uses a wide range of features provided by AF3: both strong and weak causalities, product types, user defined functions, code specifications, state automatons. We checked state reachability, non-determinism and ports' bound violation on this model. We found that in particular *stamp* automaton was non-deterministic.

V. RELATED WORK

The closest related work to our approach is the older implementation of formal verification support for AutoFOCUS3 [4]. Our solution is modular, of a better quality (fixed known bugs), supporting more features: product types, user defined functions, support for different causalities, and better support for counterexample simulation.

mbeddr [7] also supports formal verification using CBMC[3] on the generated C code, however, it is not a high-level to low-level transformation, as the input to CBMC is C code.

Simulink Design Verifier[4], Scade Suite[5], Dezine[6] are professional tools which support formal analyses requiring a behavioral transformation but their details are not public for comparison.

The work [8] describes a two step transformation from Scade to SMT for the purpose of verification. The first step transforms a Scade model to an intermediary language (called LAMA) program, and second step transforms a LAMA program to SMT. This transformation is not modular and does

[3]http://www.cprover.org/cbmc/
[4]https://www.mathworks.com/
[5]http://www.esterel-technologies.com/
[6]http://www.verum.com/

not supports reverse transformation (note in addition that this is a master thesis written in German).

A transformation from a DSL to Event-B [9] is described in [10]. This transformation targets a DSL, whereas AF3 is more general and has a wider area of application. Secondly, this transformation does not support reverse transformation, instead the DSL program is simulated in a simulator for Event-B.

The GEMOC[7] initiative aims at the conceptual globalization of modeling languages. They target the automated processing of heterogeneous modeling languages, and provide framework for coordinated execution. Even though dealing with execution semantics at different levels of abstraction, the approach used to solve the problem is not based on model transformation and therefore not relevant for our work.

VI. Conclusion

We have implemented the transformation of AutoFOCUS3 models to nuXmv models. We have used this transformation to apply formal verification on model of a pick and place unit.

This complex transformation provides new insights on what is required by a model transformation such that it can be reused with ease. In fact, systematizing model transformation reuse is one of our future work which would be useful for the community. Secondly, we plan work on reasoning about the model transformations: find a set of properties we can verify, and find (or define if needed) a language suitable to express such properties.

Acknowledgments. This work builds on top of the work done by Daniel Ratiu who implemented the original transformation of [4].

References

[1] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz, "AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems," in *ACES-MB&WUCOR@ MoDELS*, 2015, pp. 19–26. [Online]. Available: http://ceur-ws.org/Vol-1508/paper4.pdf

[2] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *CAV*, 2014, pp. 334–342. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_22

[3] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: A new symbolic model verifier," in *CAV*, 1999, pp. 495–499. [Online]. Available: http://dx.doi.org/10.1007/3-540-48683-6_44

[4] A. Campetelli, F. Hölzl, and P. Neubeck, "User-friendly model checking integration in model-based development," in *CAINE*, 2011.

[5] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer New York, 2001. [Online]. Available: https://books.google.de/books?id=A13SBNBSUIsC

[6] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann, "Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit," Technische Universität München, Tech. Rep., 2014.

[7] M. Voelter, D. Ratiu, B. Schätz, and B. Kolb, "mbeddr: an extensible c-based programming language and IDE for embedded systems," in *SPLASH*, 2012, pp. 121–140. [Online]. Available: http://doi.acm.org/10.1145/2384716.2384767

[8] H. Basold, "Transformation von scade-modellen zur smt-basierten verifikation," *CoRR*, vol. abs/1403.2752, 2014. [Online]. Available: http://arxiv.org/abs/1403.2752

[9] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[10] U. Tikhonova, M. Manders, M. Brand, S. Andova, and T. Verhoeff, "Applying model transformation and Event-B for specifying an industrial DSL," pp. 41–50. [Online]. Available: http://ceur-ws.org/Vol-1069/07-paper.pdf

[7] http://gemoc.org