Converting Extended Finite-State Machines from Supremica to CIF

Downloaded from: https://research.chalmers.se, 2026-09-09 02:07 UTC

Citation for the original published paper (version of record):
Fabian, M., Reniers, M. (2026). Converting Extended Finite-State Machines from Supremica to CIF. 2026 18th International Workshop on Discrete Event Systems Wodes 2026: 43-48. http://dx.doi.org/10.1109/WODES69290.2026.11598540

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Converting Extended Finite-State Machines from Supremica to CIF

Martin Fabian, Michel Reniers

***Abstract*— SUPREMICA and ESCET are two tools that both support modeling discrete event systems by extended finite-state machines, which are finite-state automata extended with bounded discrete variables and guards and actions over those variables. Both tools implement efficient algorithms for verification and synthesis of supervisors according to the Ramadge and Wonham supervisory control theory. But the tools are not identical, and they excel in different areas of functionality, which makes it useful to be able to convert models between them. One distinction between them is that SUPREMICA uses a graphical interface, while ESCET uses a textual representation given in the CIF language. Conversion from CIF to SUPREMICA is readily available from within ESCET, and this paper describes the conversion from SUPREMICA to CIF implemented as a script within SUPREMICA. This conversion is not as straightforward as would first be thought, due to multiple idiosyncrasies of the two formats of extended finite-state machine representation.**

I. INTRODUCTION

Discrete event systems (DES) [2] is a useful formalism for modeling systems that exhibit a discrete state-transition behavior. This includes many man-made systems, such as, infrastructural systems [19], manufacturing systems [22], automotive control systems [12], and even many types of computer programs, like smart contracts [17].

A specific type of DES are Extended Finite-State Machines (EFSMs) [3, 23], which are finite-state machines (FSM) [2] extended with bounded discrete variables and guards and actions over those variables. Though EFSMs have the same expressivity as FSMs, the inclusion of variables with arbitrary (but bounded and discrete) domains lets EFSMs model systems very compactly.

The Supervisory Control Theory (SCT) [18] is a formal approach to generate correct-by-construction controllers for DES. Given a DES modeling a *plant* to be controlled, and a DES modeling a *specification* of the desired behavior, a controller, called a *supervisor*, can be automatically *synthesized* that guarantees that the controlled system of plant and supervisor exhibits the realizable parts of the desired behavior. During its 30+ years of existence [24], the SCT has shown a big promise but has not yet delivered industrially on a large scale, in part probably due to lack of useful tools.

Though there exist many implementations and libraries for supervisor synthesis [4, 5, 7, 8, 11, 21][1], two tools stand out due to their documented industrial use, ESCET [6, 9, 10] and SUPREMICA [14, 20]. At the core, these two tools have the same functionality, modeling EFSMs and synthesizing supervisors, but they implement different approaches. In addition, there is functionality in one tool not available in the other. For instance, ESCET can generate PLC code from supervisors, which is not available in SUPREMICA. On the other hand, SUPREMICA implements abstraction-based compositional synthesis algorithms [15] not available in ESCET. SUPREMICA uses a graphical user-interface for laying out EFSMs, while ESCET uses a textual format for EFSM modeling. Each tool has benchmark libraries that would be useful to have available in the other tool.

This paper presents an implementation to convert from SUPREMICA EFSM models to the ESCET format called CIF. This allows SUPREMICA users to benefit from functionality exclusive to ESCET, and so opens up a wider palette of tools to engineers and others wanting to reap the benefits of the SCT. Emphasis has been placed on a syntactical conversion in the form of pure textual replacements. However, this has not been possible in all cases, mainly when the domains of variables affect the conversion. Then, the conversion necessarily involves some semantics.

SUPREMICA is a tool that implements an informal graphical "language" for modeling EFSMs. The syntactic and semantic rules that govern SUPREMICA's language are enforced by the implementation of the tool, but these rules are not written down in any formal way (other than the Java code that constitutes the implementation). Most parts of SUPREMICA's language follow standard syntax and semantics, but there exist corner cases with not immediately obvious behavior.

On the other hand, CIF is a declarative modeling language to describe EFSM systems, with formally defined syntax and grammar. CIF also supports modeling timed, and hybrid systems as collections of synchronizing automata, but the functionality outside of pure EFSMs is not considered in this work. Similar to SUPREMICA, the semantics of CIF are embedded within its implementation in a tool (ESCET), and not formalized (in any other way).

II. BACKGROUND

Extended Finite-State Machines (EFSMs) [3, 23] are ordinary finite-state machines [2], extended with bounded discrete variables $V = \langle v_1, v_2, \ldots v_n \rangle$ each with the domain D_{v_i}, and logical expressions, called *guards*, and arithmetic operations, *actions*, over those variables. A variable v_i has an initial value (possibly non-deterministically) chosen from the set $v_i^0 \subseteq D_{v_i}$ and a set of *marked* values $v_i^\omega \subseteq D_{v_i}$.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

M. Fabian is with the Dept. of Electrical Eng., Chalmers University of Tech., Göteborg, Sweden; M. Reniers is with the Dept. of Mechanical Eng., Eindhoven University of Tech., Netherlands

[1] https://ieeecss.org/tc/discrete-event-systems/resources

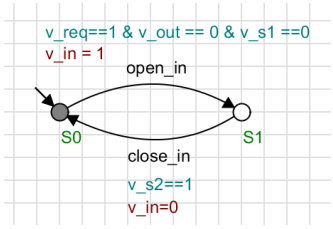

Fig. 1. The `InletValve` model of SUPREMICA's `Dosing Tank – EFA` module.

Define an EFSM as $E = \langle \Sigma_E, L_E, L_E^0, L_E^\omega, \rightarrow \rangle$, with Σ_E the finite set of event labels, the *alphabet*; L_E the finite set of *locations*, with $L_E^0 \subseteq L_E$ the set of initial, and $L_E^\omega \subseteq L_E$ the set of marked locations; and $\rightarrow$ the *transition relation*, where a transition $\langle l_m, \sigma, g, a, l_n \rangle$ denotes that the EFSM can transit from location $l_m \in L_E$ to location $l_n \in L_E$ when the guard g evaluates to *true* and the event $\sigma \in \Sigma_E$ occurs, and then the action a is performed.

A *state* of an EFSM is given by the location together with the values of the variables. Thus, a *marked state* is one where the location and all variable values are marked. The *language* of an EFSM E, $\mathcal{L}(E) \subseteq \Sigma_E^*$, is the set of all possible finite sequences of events, and the *marked language*, $\mathcal{L}_m(E) \subseteq \mathcal{L}(E)$, is the set of all finite sequences of events reaching marked states.

EFSMs interact through *synchronous composition* [23], where shared events occur simultaneously. The guards on the transitions labeled by the events are conjuncted, and actions are merged.

Both SUPREMICA and CIF support EFSMs and operations on them with similar, but not identical, syntax and semantics.

A. SUPREMICA *EFSMs*

SUPREMICA uses a graph based format to define EFSMs, where the user adds nodes, events and transitions, and edits guards and actions in a graphical user interface (GUI). As an example, the `InletValve` EFSM of SUPREMICA's `Dosing Tank – EFA`[2] model is shown in Fig. 1. The graph nodes are the locations, the edges represent the transitions, and the events, guards, and actions are laid out in connection to the transitions. In Fig. 1, `v_s2==1` is the guard for the transition from `S1` to `S0`, while `v_in=0` is the action, and `close_in` is the event. The `S0` location is both initial, as denoted by the small arrow pointing to it, and marked, denoted by the node being filled.

Variables can be *primed* in guards to refer to the *next-state value*, such as `varX'`. Most times, expressions with primed variables can be replaced by actions, except when this construct is used to express a non-deterministic assignment. A transition can have a guard like `varX == 0 & varX' >= 2`, which expresses that the transition is enabled when `varX` has the value 0, and when the transition fires the next-state value of `varX` will be some domain value larger or equal to 2. Such a guard would not be true, and thus disable the transition, if the next-state value would be outside of the variable's domain, which can happen when arithmetic is performed.

SUPREMICA uses the symbols `&`, `|`, `!` for the respective logical connectives *and*, *or*, and *not*; also, SUPREMICA uses `==` for comparison and `=` for assignment.

Enumerations, henceforth called *enums*, are defined in SUPREMICA by defining the variable domain to be a set of distinct values, like `[low,mid,high]`, and the initialization predicate to assign the variable one (or more) of those values. From then on SUPREMICA prevents assigning the variable any value not in the defined domain, and arithmetic is not allowed with that variable.

B. CIF EFSMs

CIF uses a text based format for defining EFSMs, locations, events, etc. An EFSM is defined by a type, `plant`, `requirement`, `supervisor`, or the generic `automaton`. Fig. 2 shows the definition of a plant automaton named `Sensor1` in CIF. In this textual representation, the logical connectives are `and`, `or`, and `not`. CIF uses `=` for comparison and `:=` for assignment.

In CIF, enums are user-defined distinct types, defined like `enum TrafficColor = RED, ORANGE, GREEN;` after which variables of this type can be defined, such as `disc TrafficColor light = RED;`, which defines a variable `light` of type `TrafficColor` with the initial value `RED`. After such a definition, CIF prevents assigning the variable any value outside of the defined domain, and arithmetic is not allowed with that variable, just as in SUPREMICA. However, different from SUPREMICA, CIF also does not allow comparison of the `TrafficColor` variable with any variable of another enum type, even though the value `RED` may exist with that other enum type.

III. SUPREMICA TO CIF CONVERSION

The SUPREMICA to CIF conversion is implemented as a Lua [13] script in a specific version of SUPREMICA, called LUPREMICA. This version is downloadable as a public branch from the SUPREMICA web site [20]. The script, named `Supremica2CIF.lua`, makes extensive use of Lua's built-in functions for textual pattern matching and replacement, `gmatch`, `gsub`, `find`, etc.

A. Converting Variables

In SUPREMICA, variables are first-class objects, just like EFSMs; in fact a valid SUPREMICA model can consist of only variables. This is not the case in CIF, where each variable has an *owner*, an EFSM that defines the variable. This owner-EFSM is the only EFSM that can assign the variable, other EFSMs can only read the value of the variable, and must then refer to the variable with its name prefixed by the owner's name, see for instance the variable `v_out` on line 7 in Fig. 2; this variable is owned by `OutletValve` and hence must be prefixed by that name when it is read by `Sensor1`.

[2]available in SUPREMICA: `Examples > Manufacturing Systems Examples > Dosing Tank – EFA`

```
1  plant Sensor1:
2    disc int[0..1] v_s1 in any;
3    initial v_s1=0;
4    marked v_s1=0;
5
6  location S1:
7    edge  s1_off when (OutletValve.v_out=1  and
         Sensor2.v_s2=0) do v_s1 := 0 goto S0 ;
8  location S0:
9    initial;
10   marked;
11   edge  s1_on when (InletValve.v_in = 1) do v_s1 := 1
         goto S1 ;
12 end // Sensor1
```

Fig. 2. The `Sensor1` EFSM of SUPREMICA's `DosingTank-EFA` model converted to CIF.

```
1  plant EFA:
2    disc int[0..3] varX in any;
3    initial varX = 0  or  1 = varX;
4    marked 3 = varX;
5  location S1:
6    marked;
7    edge  e2 goto S0 ;
8  location S0:
9    initial;
10   marked;
11   edge  e1 when (0 <= varX + 1 and varX + 1 <= 3) do varX
         := varX + 1 goto S1 ;
12 end // EFA
```

Fig. 3. Protective guards added to the CIF model, see parentheses on line 11, to guarantee that the new value of `varX` is within its domain.

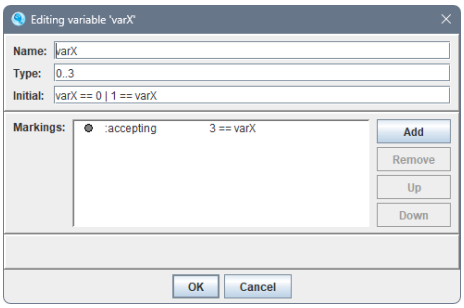

Fig. 4. A SUPREMICA variable named `varX`, with domain 0..3, non-deterministic initial value 0 or 1, and marked value 3.

The conversion script reads the elements (variables and EFSMs) from SUPREMICA in arbitrary order, and must therefore do some extra book-keeping in relation to the variables and their owners. The first EFSM encountered that assigns a variable will be considered to own that variable, and if another EFSM is later encountered that also assigns the variable, the conversion will stop with an error message saying that two EFSMs cannot assign the same variable. A suggestion to get around this is to synchronize all EFSMs that assign the same variable, so that there is only a single one, but this is left at the discretion of the user.

Though a module with only variables and no EFSMs is valid in SUPREMICA, it would not be valid in CIF, since there would be no owners of the variables. A single-location EFSM could be generated, but since such modules are of limited use, the conversion script throws an exception instead.

B. Converting Assignments

For variables with numerical domains, like `varX` shown in Fig. 4, SUPREMICA allows several arithmetic operators that are not available in CIF, $+=$, $-=$, $*=$, and $/=$. These are shorthand for assigning the variable a new value from its previous value plus (or minus or multiplied or divided) by what comes after the operator. So for `varX += varY`, this is equivalent to `varX = varX + varY`, and all such constructs are converted to their equivalent forms, which are accepted by CIF. This conversion can and is done purely syntactically.

Furthermore, SUPREMICA has implicit guards on transitions with assignment or increment or decrement actions like `varX = varY`, or `varX -= 1`, so that if the resulting assignment of `varX` would be outside of `varX`'s domain, the transition is disabled. This is not handled the same way by ESCET; the assignment would be allowed but in some following operations on the EFSMs ESCET may report an error. Thus, when a variable is assigned in whatever way, if necessary (and sometimes when not), the conversion adds explicit *protective guards*, as shown in Fig. 3, line 11. These guards are conjuncted with any explicit guards of the transition to guarantee that the value of the variable always remains within its domain.

Fig. 3 also shows how the domain, the non-deterministic initialization, and the marking of values of a variable, `varX` shown in Fig. 4, are converted to CIF, see lines 2–4.

SUPREMICA does not implement variable *types*; a variable is defined by its name, domain, and initialization predicate, see Fig. 4. In contrast, variables are strongly typed in CIF, so the conversion must decide on a proper type for each variable. This is done based on the type of the variable's domain.

C. Converting Enums

As noted above, CIF considers enum types distinct, while SUPREMICA is much more relaxed, and allows to compare two enums of different types, simply checking if the identifiers are equal when compared as strings. This is not allowed by CIF, where only enums of the same type can be compared. To get around this, all SUPREMICA enums are put in one big `Enums` type, being careful to remove duplicates. Additionally, guards are added on each assignment of an enum so that it is not assigned values outside the domain it had in SUPREMICA.

Consider two SUPREMICA variables, `varX` and `varY`, with the respective domains [`low`, `mid`, `high`] and [`left`, `mid`, `right`]. When converting to CIF, the `Enums` type will have the domain [`low`, `left`, `mid`, `right`, `high`], and whenever `varX` is assigned a value, a guard checks that this value is either `low` or `mid` or `high` (and similarly for `varY`). This allows for `varX` and `varY` to be compared in CIF.

A special case of SUPREMICA enums are variables that have domains [`true`, `false`]. These are by the conversion script considered to be Booleans in CIF.

D. Converting Booleans

SUPREMICA does not have proper Boolean variables. Instead, users can define their own Boolean variables in

multiple ways, 0–1-variables, or enums with a [`true`, `false`] domain, or many other ways. For variables with numerical domains, any arithmetic operation is valid in SUPREMICA. However in CIF, which has a proper Boolean type, arithmetic is not allowed on Booleans.

So, two-valued numerical domain variables cannot arbitrarily be converted to Boolean in CIF. Instead all variables with numerical domains are directly converted into CIF variables with the same domains. Similarly, arbitrary two-valued domain enums are converted as ordinary enums, as described in Section III-C.

The only case where SUPREMICA variables are converted to Booleans in CIF, are for enums with the domain [`true`, `false`] (or [`false`, `true`]). The conversion code checks for this particular type of domain, and specifically converts these variables into Booleans.

E. Sanitizing Identifiers

In SUPREMICA, identifiers such as event labels, EFSM names, variable names, and enum labels can include one or more colons. For instance, `var:X:Y:Z` is a valid identifier in SUPREMICA, as is `e::::1`. This is an unfortunate historical accident, and of course, CIF does not allow this. So, all such identifiers must be *sanitized*, that is, turned into identifiers accepted by CIF. Unique sanitized output for unique un-sanitized input must be guaranteed, that is, it must be made sure that a *sane* identifier, like `varX`, does not clash with some to-be sanitized identifier, like `var:X`. For this, a double-directed map is implemented in the conversion script with both the sanitized and un-sanitized identifiers as keys. This allows to quickly check if a new sanitized identifier clashes with some other already sanitized identifier.

Sanitizing the identifiers is done by replacing colons with underscores (_). However, this is not done as a one-for-one replacement, instead, it is done in a way that tries to minimize the number of underscores, sequences of colons are mapped to possibly shorter sequences of underscores. If a sanitized identifier clashes with some other already sanitized identifier, the number of underscores is increased by one, and the result is checked again.

F. Namespaces

In addition to SUPREMICA allowing colons in identifier names, there is also an issue with *namespaces*. CIF has only a single namespace, so automata names, location names, variable names, and event labels all have to be distinct; there cannot be an automaton with the same name as an event, for instance. SUPREMICA, on the other hand, implements multiple namespaces (to some extent); event labels are in their own namespace, and so an event can be named the same as an automaton or a variable, as in Fig. 5. In such case, the conversion must differentiate between the automaton, the event, and the location, all named `Same` in SUPREMICA. Currently, an exception is raised and the user must sort this out, but ideally this should be automated.

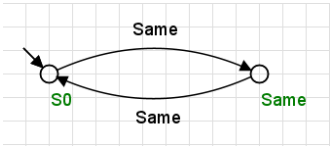

Fig. 5. SUPREMICA EFSM named `Same`, with a location and an event with the same name.

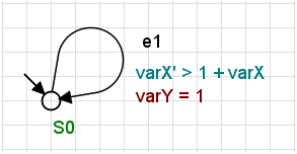

Fig. 6. Simple EFSM with a guard referring to the next-state value of `varX`.

G. Converting Predicates

Apart from guards, also variable initializations and variable markings are expressed by predicates in SUPREMICA, as shown in Fig. 4.

In general, predicates can be straightforwardly converted from SUPREMICA to CIF, by simple textual replacement of the logical operators `&`, `|` and `!`, by `and`, `or` and `not`, respectively, and comparison == by =. The other comparison operators <=, != etc, use the same symbols in SUPREMICA and CIF. This is a straightforward syntactic conversion.

However, for guard predicates involving *primed* variables, that is, referring to the next-state values, which is not directly available in CIF, a bit more care has to be taken.

H. Primed Variables in Guards

As mentioned above, in SUPREMICA guards can refer to the next-state value of a variable by putting a prime (′) on the variable name, like `varX′ > 1 + varX`. This predicate is *true* when the next-state value of `varX` is larger than 1 plus the current value of `varX`, and *false* otherwise. Thus, the transition is disabled if the next-state value of `varX` is *not* larger than 1 plus the current value of `varX`, which happens if 1 plus `varX`'s current value is outside of `varX`'s domain.

The main use of this notation is to have non-deterministic assignments of variables.

Except for knowing the domains of the variables, the conversion can be done purely syntactically. For a transition $\langle l_j, \sigma, g, a, l_k \rangle$ in SUPREMICA, let the guard g have $n \in \mathbb{N}$ number of primed variables v_i $(i = 1, \ldots, n)$, each with the domain D_{v_i}. Let $g(\langle \hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n \rangle)$ denote g with the respective primed variable v_i replaced by the value $\hat{v}_i \in D_{v_i}$. Then, for each $\langle \hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n \rangle \in D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$, generate a transition $\langle l_j, \sigma, g', a', l_k \rangle$ in CIF with the guard $g' = g(\langle \hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n \rangle)$ and the action $a' = \{a, v_1 := \hat{v}_1, v_2 := \hat{v}_2, \ldots, v_n := \hat{v}_n\}$.

For example, the SUPREMICA model in Fig 6, with the domain of `varX` [0..2], is converted to the CIF model of Fig 7. This conversion leads to $|D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}|$ non-deterministic transitions in CIF for such a transition in SUPREMICA with a primed guard.

```
1  controllable e1;
2
3  plant EFA:
4    disc int[0..2] varX in any;
5    initial varX = 0;
6
7    disc int[0..2] varY in any;
8    initial varY = 0;
9
10 location S0:
11   initial;
12   marked;
13   edge e1 when 0 > 1 + varX do varY := 1, varX := 0 goto
         S0;
14   edge e1 when 1 > 1 + varX do varY := 1, varX := 1 goto
         S0;
15   edge e1 when 2 > 1 + varX do varY := 1, varX := 2 goto
         S0;
16 end // EFA
```

Fig. 7. Conversion of the next-state value guard `varX' > 1 + varX`. As the domain of `varX` is non-negative, the guard on line 13 is always false.

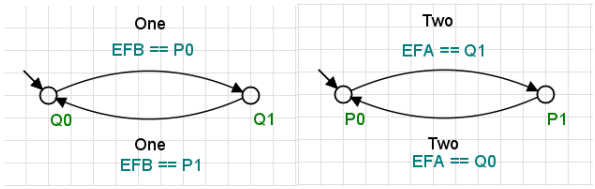

Fig. 8. Two EFSM, `EFA` to the left and `EFB` to the right, using automaton variables to sequence the events as `One Two One Two...`

Conversion of primed variables can result in a variable being assigned twice on the same transition, once as part of converting the guard, and once as part of the ordinary action. This is not allowed in CIF, even if the two assignments agree. In that case, an exception is thrown.

I. Automaton Variables

If in SUPREMICA the *Automaton Variables Compiler* is enabled (`Configure > Options... > GUI > Compiler`), it is possible to check in guards if an EFSM is currently in a specific location. This is illustrated in Fig. 8, where the transitions of the left EFSM, named `EFA`, are guarded by the right EFSM, named `EFB`, being in certain locations. Similarly, the transitions of `EFB` are guarded by `EFA` being in its respective location. For instance, `EFA`'s transition from `Q0` to `Q1` can only occur when `EFB` is in its `P0` location.

Also CIF can refer to the locations of EFSMs, though with a slightly different syntax; `EFA.Q0` refers to the `Q0` location of `EFA`, and these expressions are logical propositions just as the similar comparison predicates in SUPREMICA. The two EFSMs of Fig. 8 are converted into the CIF code in Fig. 9.

This is a purely syntactic conversion, with the only issue that the conversion code must check each such equality expression to see if the elements on either side of the == are the name of an EFSM and the label of a location in that EFSM, and if so convert to CIF syntax. However, due to the relaxed handling of namespaces in SUPREMICA, these conversions should only be made if the *Automaton Variables Compiler* is enabled.

IV. On Correctness

Neither SUPREMICA nor CIF have their full formal semantics written down, the semantics are embedded in the implementation of the respective tools (ESCET for CIF). This makes it hard to formally assess correctness of the conversion. However, EFSMs can be *unfolded* [1] into ordinary FSMs, and both SUPREMICA and ESCET can do this. Correctness of the conversion from SUPREMICA to CIF can then be assessed for specific (not so big) examples by comparing the state spaces of the unfolded EFSM systems from the respective tools. Agreeing on the number of states, events, and transitions of the respective unfolded models is an indication that the original and converted models are equivalent, but no guarantee. If the unfolded models are isomorphic though, then it is clear.

```
1  controllable One, Two;
2
3  plant EFA:
4
5  location Q1:
6      marked;
7      edge  One when (EFB.P1) goto Q0 ;
8  location Q0:
9      initial;
10     marked;
11     edge  One when (EFB.P0) goto Q1 ;
12 end // EFA
13
14 plant EFB:
15
16 location P0:
17     initial;
18     marked;
19     edge  Two when (EFA.Q1) goto P1 ;
20 location P1:
21     marked;
22     edge  Two when (EFA.Q0) goto P0 ;
23 end // EFB
```

Fig. 9. Conversion to CIF of the EFSMs in Fig. 8.

In Supremica, the *partial unfolding* of [16] is performed, when switching to the `Analyzer` tab. This results in a system of FSMs, one for each variable, and one for each EFSM. The reason not to do the (monolithic) unfolding of [1] is to preserve the compositional structure of the EFSM system for the benefit of SUPREMICA's abstraction-based compositional verification and synthesis algorithms. Selecting all FSMs and synchronizing them results in the monolithic model.

In ESCET, unfolding is done with the `Explore untimed state space` command. This command generates an FSM in the CIF format, which can then be examined for equivalence with the original SUPREMICA model. For the `Dosing Tank – EFA` example, both tools generate monolithic FSMs with 16 states, 10 events and 30 transitions. Manual inspection of the two graphs shows that they are isomorphic.

V. Conclusions

This paper describes the conversion of EFSM models from SUPREMICA to CIF. This conversion, together with reciprocal conversion from CIF to SUPREMICA available in ESCET, allows to take advantage of functionality in one tool not available in the other. This extends the palette of functionality available to the SCT community and engineers in general.

Formally assessing correctness of the conversion in general is hard, if not outright impossible. However, as the conversion is mainly syntactic it is unlikely that the conversion results in an incorrect CIF model. In addition, comparing the monolithic state spaces of multiple models and their conversion gives some confidence in the correctness of the conversion.

Future work includes minimizing the number of nondeterministic transitions generated for guards with primed variables. The truth values of many of those guards can be determined during conversion with just a slight use of semantics, and transitions with always false guards can then be removed. Lua's ability to compile and execute *chunks* comes in handy here.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 026202649X. URL: https://mitpress.mit.edu/9780262026499/principles-of-model-checking/.

[2] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Cham, 2021. ISBN: 978-3-030-72272-2. DOI: 10.1007/978-3-030-72274-6.

[3] Kwang Ting Cheng and A. S. Krishnakumar. "Automatic functional test generation using the extended finite state machine model". In: *Proceedings of the 30th ACM/IEEE Design Automation Conference*. 1993, pp. 86–91. DOI: 10.1145/157485.164585.

[4] Ashfaq Farooqui, Fredrik Hagebring, and Martin Fabian. "MIDES: A Tool for Supervisor Synthesis via Active Learning". In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. 2021, pp. 792–797. DOI: 10.1109/CASE49439.2021.9551435.

[5] Lei Feng and W.M. Wonham. "TCT: A Computation Tool for Supervisory Control Synthesis". In: *2006 8th International Workshop on Discrete Event Systems*. 2006, pp. 388–389. DOI: 10.1109/WODES.2006.382399.

[6] W. J. Fokkink, M. A. Goorden, D. Hendriks, D. A. van Beek, A. T. Hofkamp, F. F. H. Reijnen, L. F. P. Etman, L. Moormann, J. M. van de Mortel-Fronczak, M. A. Reniers, J. E. Rooda, L. J. van der Sanden, R. R. H. Schiffelers, S. B. Thuijsman, J. J. Verbakel, and J. A. Vogel. "Eclipse ESCET™: The Eclipse Supervisory Control Engineering Toolkit". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 44–52. DOI: 10.1007/978-3-031-30820-8_6.

[7] Benoît Fraikin, Marc Frappier, and Richard St-Denis. "Supervisory control theory with Alloy". In: *Science of Computer Programming* 94 (2014). Abstract State Machines, Alloy, B, VDM, and Z, pp. 217–237. DOI: 10.1016/j.scico.2014.04.016.

[8] Florian Göbe, Thomas Timmermanns, Oliver Ney, and Stefan Kowalewski. "Synthesis Tool for Automation Controller Supervision". In: *2016 13th International Workshop on Discrete Event Systems (WODES)*. 2016, pp. 424–431. DOI: 10.1109/WODES.2016.7497883.

[9] Dennis Hendriks, Michel Reniers, Wan Fokkink, and Wytse Oortwijn. *Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0*. 2025. arXiv: 2511.04370 [eess.SY].

[10] Eclipse ESCET – Home. https://eclipse.dev/escet/. [Online; Accessed: 2026-04-07.

[11] Claudius Jordan, Canlong Ma, and Julien Provost. "An educational toolbox on supervisory control theory using MATLAB Simulink stateflow: From Theory to practice in one week". In: *2017 IEEE Global Engineering Education Conference (EDUCON)*. 2017, pp. 632–639. DOI: 10.1109/EDUCON.2017.7942912.

[12] Tim Korssen, Victor Dolk, Joanna van de Mortel-Fronczak, Michel Reniers, and Maurice Heemels. "Systematic Model-Based Design and Implementation of Supervisors for Advanced Driver Assistance Systems". In: *IEEE Transactions on Intelligent Transportation Systems* 19.2 (2018), pp. 533–544. DOI: 10.1109/TITS.2017.2776354.

[13] *Lua Reference Manuals*. URL: https://www.lua.org/manual/.

[14] Robi Malik, Knut Åkesson, Hugo Flordal, and Martin Fabian. "Supremica – An Efficient Tool for Large-Scale Discrete Event Systems". In: *IFAC-PapersOnLine* 50.1 (2017). 20th IFAC World Congress, pp. 5794–5799. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.427.

[15] Robi Malik, Sahar Mohajerani, and Martin Fabian. "A survey on compositional algorithms for verification and synthesis in supervisory control". In: *Discrete Event Dynamic Systems* 33.3 (Sept. 2023), pp. 279–340. DOI: 10.1007/s10626-023-00378-8.

[16] Sahar Mohajerani, Robi Malik, and Martin Fabian. *Partial unfolding for compositional nonblocking verification of extended finite-state machines*. Tech. rep. Chalmers University of Technology, Göteborg, Sweden, 2013. URL: https://research.chalmers.se/publication/172205.

[17] Nishant Parekh, Wolfgang Ahrendt, and Martin Fabian. "Automatic Conversion of Smart Contracts for Non-Blocking Verification". In: *IFAC-PapersOnLine* 58.1 (2024). 17th IFAC Workshop on Discrete Event Systems WODES 2024, pp. 282–287. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2024.07.048.

[18] P.J.G. Ramadge and W.M. Wonham. "The control of discrete event systems". In: *Proceedings of the IEEE* 77.1 (1989), pp. 81–98. DOI: 10.1109/5.21072.

[19] F.F.H. Reijnen, J.M. van de Mortel-Fronczak, M.A. Reniers, and J.E. Rooda. "Design of a Supervisor Platform for Movable Bridges". In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. 2020, pp. 1300–1306. DOI: 10.1109/CASE48305.2020.9216894.

[20] Supremica – Public Repo. https://tinyurl.com/Supremica. [Online; Accessed: 2025-03-31].

[21] L. Ricker, S. Lafortune, and S. Genc. "DESUMA: A Tool Integrating GIDDES and UMDES". In: *2006 8th International Workshop on Discrete Event Systems*. 2006, pp. 392–393. DOI: 10.1109/WODES.2006.382402.

[22] Bram van der Sanden, Michel Reniers, Marc Geilen, Twan Basten, Johan Jacobs, Jeroen Voeten, and Ramon Schiffelers. "Modular model-based supervisory controller design for wafer logistics in lithography machines". In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015, pp. 416–425. DOI: 10.1109/MODELS.2015.7338273.

[23] Markus Sköldstam, Knut Åkesson, and Martin Fabian. "Modeling of discrete event systems using finite automata with variables". In: *2007 46th IEEE Conference on Decision and Control*. 2007, pp. 3387–3392. DOI: 10.1109/CDC.2007.4434894.

[24] W.M. Wonham, Kai Cai, and Karen Rudie. "Supervisory control of discrete-event systems: A brief history". In: *Annual Reviews in Control* 45 (2018), pp. 250–256. DOI: 10.1016/j.arcontrol.2018.03.002.