UltraDES - A Library for Modeling, Analysis and Control of Discrete Event Systems ⋆

Lucas V. R. Alves * **Lucas R. R. Martins** ** **Patrícia N. Pena** ***

* *COLTEC - Universidade Federal de Minas Gerais Belo Horizonte, MG, Brazil (e-mail: lucasvra@ufmg.br).*
** *Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil (e-mail: lucasrangelrm@gmail.com)*
*** *Department of Electronics Engineering - Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil, (e-mail: ppena@ufmg.br)*

Abstract: In this paper a library of functions and data structures for analysis and control of Discrete Event Systems based in the .NET Framework is proposed. The main objective is to create an environment for the implementation of algorithms for Discrete Event Systems, as well as the integration of these algorithms and codes in the fields of IT (Information Technology) and AT (Automation Technology). The data structure, and the functions implemented so far are presented. The performance of the current version of the library is evaluated.

Keywords: Discrete Event Systems, Supervisory Control Theory, Software Package

1. INTRODUCTION

UltraDES is an object oriented library composed of data structures and algorithms for the modeling, analysis and control of discrete event systems (DES). The library was developed in `C#` language, a very popular language in the Information Technology area (IT) and is based on the *.NET Framework* as its execution platform. UltraDES can be used in any language that supports *.NET Framework*, including the *.NET* versions of the languages C++ and Python, Visual C++ and IronPython respectively.

One great advantage of using the *.NET Framework* in the industrial field is that there is an OPC protocol, an industrial communication pattern, named OPC .NET 4.0, developed to support programs developed in the *.NET* platform. Allowing the use of *UltraDES* in real industrial applications

The *Supervisory Control Theory* (SCT), proposed by Ramadge and Wonham (1989), is a framework for the modeling and control of discrete event systems, based on language and automata theory (Hopcroft et al., 2001). The system to be controlled in named *plant* and the controller agent is named *supervisor*. The role of the supervisor is to restrict the behavior of the plant to a sublanguage that respects a desired language for the closed loop system. The action of the supervisor is on the disablement of a subset of the events in response to the observation of events in the plant.

There is a number of softwares developed for the study of DES: TCT (Feng and Wonham, 2006), Supremica (Åkesson et al., 2006), DESUMA (Ricker et al., 2006), libFAUDES (Moor et al., 2008), DESLAB (Clavijo et al., 2012), among others. Some of the softwares are not open source (TCT and Supremica) what prevents the implementation of customized solutions and new algorithms. The ones that are open code are developed in a specific language (C++ for libFAUDES and Python for DESLAB).

This paper presents a new software, *UltraDES*, which has implementations of the most common algorithms and data in Discrete Event Systems. The *UltraDES* was developed having in mind usability and expandability, with a low learning curve and allowing new algorithms to be implemented without much effort.

The paper is structured as follows. Section 2 presents the preliminary concepts of Discrete Event Systems (DES) and Supervisory Control Theory of DES. The following section describes the main classes and methods implemented in *UltraDES*. Section 4 presents the code for the implementation of a simple example and Section 5 presents the performance tests where *UltraDES* is compared to two other softwares. The paper ends with Section 6, where the results are summarized and the current and future work is mentioned.

2. PRELIMINARIES

UltraDES was developed under the framework of TCS (Ramadge and Wonham, 1989). Under this paradigm, the logical behavior of a DES is modeled by strings of events obtained from an alphabet Σ. The *Kleene closure* Σ^* is the set of all strings over Σ, including the empty string ε. Consider sequences s, v and t over Σ^*. The concatenation of s with v forms the string $t = sv$, and we can say that s is *prefix* of t, denoted by $s \leq t$. Any subset $L \subseteq \Sigma^*$ is

⋆ This work was supported by Capes - Brazil, CNPq and FAPEMIG.

named a *language*. The *prefix closure* $\overline{L}$ of L is the set of all prefixes of strings of L.

From the Kleene Theorem, regular languages are recognized by automata. A nondeterministic finite state automaton (NFA) is defined by a quintuple $G = (\Sigma, Q, \rightarrow, Q^\circ, Q_m)$, where Σ is an alphabet, Q is the finite set of states, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q^\circ \subseteq Q$ is the set of initial states and $Q_m \subseteq Q$ is the set of marked states. An automaton is called a deterministic finite automaton (DFA) if $|Q^\circ| = 1$ and for each $q \in Q$ and $\sigma \in \Sigma$ there is at most one state $q' \in Q$ such that $q \xrightarrow{\sigma} q'$. An automaton G implements two languages, the generated language $\mathcal{L}(G)$ and the marked language $\mathcal{L}_m(G)$. The generated language represents all sequences that can be executed in the automaton from the initial state and the *marked language* $\mathcal{L}_m(G) \subseteq \mathcal{L}(G)$ is composed of the set of strings that reach marked states.

Given an automaton G, a state $q \in Q$ is *acessible* in G if $q^\circ \xrightarrow{s} q$ such that $q^\circ \in Q^\circ$ and $s \in \Sigma^*$; a state $q \in Q$ is *coacessible* if $q \xrightarrow{s} q'$ with $q \in Q$, $q' \in Q_m$ and $s \in \Sigma^*$. An automaton is said to be accessible if all states are acessible. An automaton is said to be coacessible if all states are coacessible. The acessible component of G, $Ac(G)$, is obtained from G by eliminating nonacessible states and associated transitions. The coacessible component of G, $CoAc(G)$, is obtained by eliminating noncoacessible states and associated transitions. An automaton G is trim if it is acessible and coacessible, namely $trim(G) = CoAc(Ac(G))$. An automaton G is *nonblocking* if $\overline{\mathcal{L}_m(G)} = \mathcal{L}(G)$.

A DES can be obtained by the parallel composition of subsystems, namely $G = \|_{i=1}^{n} G_i$, where G_i for each $i = 1, \ldots, n$, represents the model of each subsytem. A *global specification* E is another automaton that implements the restrictions that are to be applied to the open loop system. E can be obtained by the parallel composition of a set of E_j, $j = 1, \ldots, m$, such that $E = \|_{j=1}^{m} E_j$.

The *supervisory control problem* consists in finding a supervisor that restricts the behavior of a DES to respect a global specification. An important aspect of the modeling process is to establish a partition of the events of the system into *controllable* and *uncontrollable*. The controllable events are the events that can be disabled by the controller (commands, typically) and the uncontrollable events are the events that cannot be disabled (responses of the system). The supervisor action over the plant is to inhibit the occurrence of controllable events aiming to retain the system within the legal behavior K (that is modeled by the composition of E and G).

The generated and marked language of the closed loop system are $\mathcal{L}(S/G)$ and $\mathcal{L}_m(S/G) \subseteq \mathcal{L}_m(G)$, respectively. Let G be a plant and E a specification, the necessary and suficient condition for the existence of a nonblocking supervisor S for G such that $\mathcal{L}_m(S/G) = \mathcal{L}(G) \parallel E = K$, is that K be *controllable* with relation to $\mathcal{L}(G)$ and Σ_{nc}, namely, $\overline{K}\Sigma_{nc} \cap \mathcal{L}(G) \subseteq \overline{K}$. If K is not controllable, then a supervisor is obtained by obtaining the supremal controllable sublanguage of K, denoted by $Sup\mathcal{C}(K, G)$. This language always exist.

In order to manipulate, analize and compose automata, design supervisors and so on, it is very useful if there is a software to do so. Also, new algorithms developed can be implemented. The next sections introduce the main aspects of *UltraDES*.

3. DATA STRUCTURE

The library *UltraDES* is composed by many classes that represent automata, their components (states, events and transitions) and regular expressions. Also, there are many auxiliary classes.

3.1 States

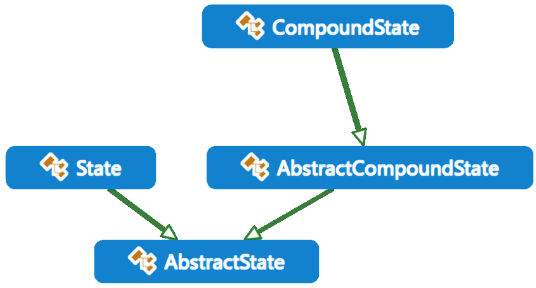


Fig. 1. Relation among classes that represent states.

The main class that represents a state is named *AbstractState*. It is abstract, which means that it is not possible to instantiate an object from it, but it defines basic characteristics that a state must have, such as alias (*alias*) and marking (*Marking.Marked* or *Marking.Unmarked*). Since the union of two states is very common in operations with automata, another abstract class is created, derived from *AbstractState*, named *AbstractCompoundState* that has, other then the characteristics already defined in *AbstractState*, a pointer to the original states that generated the compound state. *UltraDES* uses both *AbstractState* and *AbstractCompoundState* as states.

Since it is not possible to create objects from the classes *AbstractState* and *AbstractCompoundState*, classes derived from these two primitive classes were created and named, respectively *State* and *CompoundState*.

```
var  s1 = new State("s1",Marking.Marked)
var  s2 = new State("s2",Marking.UnMarked)
```

3.2 Events and Regular Expressions

In the library, an event is defined by the abstract class *AbstractEvent*, that establishes its basic characteristics such as (*alias*) and controllability (*Controllability.Controllable* or *Controllability.Uncontrollable*).

In a similar way of what was done with the state, classes that implement *AbstractEvent* are defined. A general event is defined by the class and there are two special events defined as *singleton* classes, *Epsilon* (ϵ) and *Empty* ($\varnothing$). The decision to implement ϵ as an event when it is well know to be a string is related to the implementation of the regular expression, where events are considered as the basic strings for the iterative definition of a regular expression.

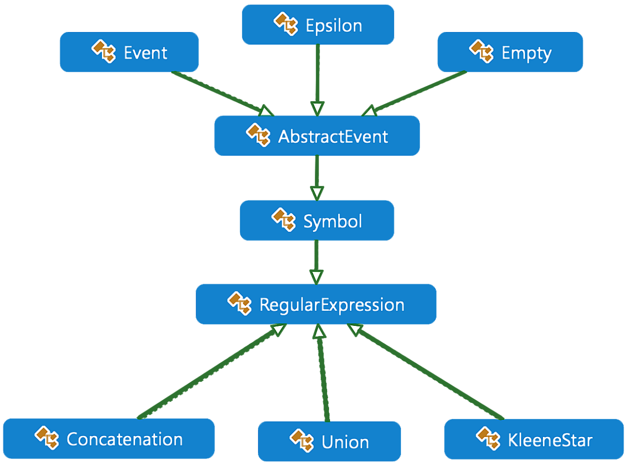


Fig. 2. Relation among the classes that represent events and regular expressions.

```
var e1 = new Event("e1", Controllability.Controllable);
var e2 = new Event("e2", Controllability.Uncontrollable);
```

Another abstract class that was defined in *UltraDES* is the *RegularExpression* class that represents a regular expression. A regular expression is defined by operations over regular expressions or symbols. For this reason, the operations over regular expressions are also defined as classes. The union of two regular expressions is represented by the class *Union*, the concatenation of two regular expressions is represented by the class *Concatenation*, the Kleene star is given by *KleeneStar* and a symbol is represented by the abstract class *Symbol*. The class *Symbol* is the base class of *AbstractEvent*, such that any event is also a symbol.

3.3 Transitions

Transitions among states are defined by means of a *Transition* class, that contains the origin state (*Origin*), the destination state (*Destination*) and the event that labels the transition (*Trigger*).

```
var t = new Transition(s1,e1,s2);
```

3.4 Auxiliary

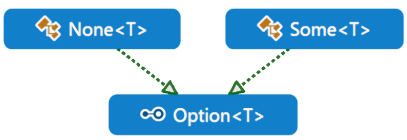


Fig. 3. Relationship between the *Option* interface and the *Some* and *None* classes used in the *UltraDES*

The main auxiliary data structure defined in *UltraDES* is the interface *Option*. This interface has two implementations, *Some*, a class that saves data of a specific type and *None*, a class that represents the absence of data. This structure is used in the transition function when, from an origin state an event transitions to another state, an object *Some* returns the destination state. If such event does not implicate a transition to another state, an object *None* is returned.

3.5 Deterministic Finite Automaton

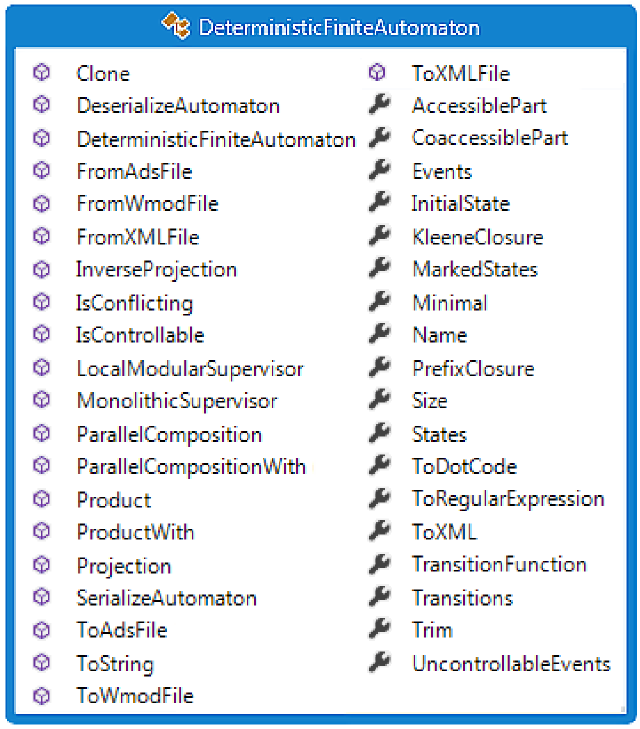


Fig. 4. Methods and Properties defined in the *DeterministicFiniteAutomaton* class

A class *DeterministicFiniteAutomaton* represents a deterministic finite automaton and is defined by a list of transitions (*Transition*), an initial state (*AbstractState*) and a name. The internal structure of the *DeterministicFiniteAutomaton* class is defined using the abstract classes *AbstractState*, *AbstractCompundState* and *AbstractEvent*, such that any class that derives from them will work in the same way, without modifications to the implemented algorithms.

```
var G = new DeterministicFiniteAutomaton(new[]{
    new Transition(s1, e1, s2),
    new Transition(s2, e2, s1)}, s1, "G");
```

Properties of a DeterministicFiniteAutomaton

States
: Returns a list with the states (*AbstractState*) of the automaton.

MarkedStates
: Returns a list with all marked (*AbstractState*) of the automaton.

InitialState
: Returns the initial state (*AbstractState*) of the automaton.

Events
: Returns a list with all events (*AbstractEvent*) of the automaton.

Name
Returns the name of the automaton. This name is modified indicating the operations that were performed over it.

Transitions
Returns a list with all transitions (*Transition*) of the automaton.

TransitionFunction
Returns the transition function of the automaton. The input of the function is an origin state and an event and the output is *None*, if there is no destination state or *Some* with the destination state.

Main Operations over DeterministicFiniteAutomaton

ParallelCompositionWith
When applied over $G1$ and with parameter $G2$, returns an automaton $G3 = G1||G2$.

```
var G3 = G1.ParallelCompositionWith(G2);
```

ProductWith
When applied over $G1$ and with parameter $G2$, returns an automaton $G3 = G1 \times G2$.

```
var G3 = G1.ProductWith(G2);
```

AccessiblePart
When applied over an automaton G, returns the accessible part of G, $Ac(G)$.

```
var G1 = G.AccessiblePart;
```

CoaccessiblePart
When applied over an automaton G, returns the coaccessible part of G, $CoAc(G)$.

```
var G1 = G.CoaccessiblePart;
```

Trim
When applied over an automaton G, returns the trim automaton of G, $Trim(G)$.

```
var G1 = G.Trim;
```

MonolithicSupervisor
The method's input is a list of plants, a list of specifications and a boolean value, *true* (default) or *false*, indicating if the supervisor should be nonblocking. The output is a monolithic supervisor that implements the supremal controllable - in relation to the composition of all plants - sublanguage contained in a desired language (K) obtained by composing the plants with the list of specifications).

```
var S = DeterministicFiniteAutomaton.MonolithicSupervisor
        (new[]{M1, M2},new[]{E});
```

LocalModularSupervisor (Queiroz and Cury, 2000)
The method's input is a list of plants, a list of specifications (can also be a list of supervisors to be checked for conflict). The output is a list of local modular supervisors. If the closed loop system is conflicting, an exception is created to indicate the error.

```
var S = DeterministicFiniteAutomaton
        .LocalModularSupervisor(new[]{M1, M2, M3}
        ,new[]{E1, E2});
```

Input and Output Methods

ToXMLFile and FromXMLFile
The method *ToXMLFile* saves information of the automaton in a XML file and the method *FromXMLFile* generates an automaton from a XML file.

ToAdsFile and FromAdsFile
The method *ToAdsFile* saves information from the automaton in an ADS file, to be used with software TCT. The information regarding states and transitions are all lost. The method *FromAdsFile* reads an ADS file and generates and automaton.

ToWmodFile and FromWmodFile
The method *ToWmodFile* saves information from the plants and the specifications in a WMod file, to be used with software Supremica. The method *FromWmodFile* reads a WMod file and generates a plant list and a specification list.

SerializeAutomaton and DeserializeAutomaton
The method *SerializeAutomaton* generates a binary file containing information of the automaton and *DeserializeAutomaton* reads a binary file and generates an automaton.

ToDotCode
The method *ToDotCode* returns a text (type *string*) that contains the representation of the automaton in DOT format, that can be visualized with software Graphviz.

3.6 Supervisor synthesis algorithm

UltraDES uses a modified version of the algorithm present in the literature to compute a monolithic supervisor. In the original algorithm it is necessary to build the automaton K to find the supervisor S. The automaton K is the limiting factor in the solution of the problems since, in the majority of problems, it has much more states and transitions than the supervisor S. In the version implemented by *UltraDES* K is not calculated directly which allows to solve bigger problems.

Instead of generating K, followed by the loop: identify bad states, remove them, repeat until it converges, our algorithm performs only one composition with all automata (plants and specifications). During the composition, when generating the states of the resulting automaton, checks are performed to detect the bad states and thus prevent such states from being included in the solution. Then, all states that would be uniquely accessed from those bad states are not calculated. Finally, the algorithm performs the removal of blocking states to obtain a trim automaton. The removal of the blocking states can generate bad states and if this occur they are removed and the algorithm returns to the step of removing blocking states.

To reduce the memory usage, *UltraDES* stores all states and transitions of the original automata (used in the parallel composition described above) not by using a '*State Object*' but by using a sequence of bits to represent a state of the supervisor. A sequence of bits informs which states from the original automata need be composed to result in a state of the automaton. The states are obtained virtually only.

The transitions of the supervisor are also not saved and they are created only if necessary. To compute the transitions of a state, *UltraDES* checks if there is a destination state for each event in the states of the original automata, using rules of the parallel composition. If so, the presence of such states in the sequence of bits is checked and the transition is created if that is the case.

4. CASE STUDY

In order to illustrate the use of *UltraDES* we present a typical DES problem, the extended small factory (Fig. 5), composed of three machines (M_i with $i \in \{1, 2, 3\}$, Fig.6(a)) and two specifications (E_j with $j \in \{1, 2\}$, Fig.6(b)) that implement the restrictions over the unitary buffers.

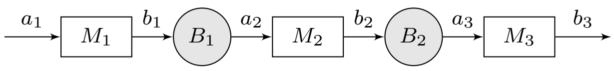
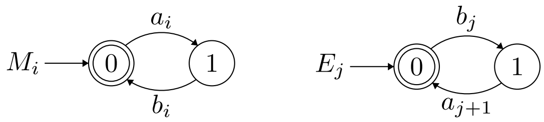


Fig. 5. Extended small factory



(a) Plants M_i, $i = \{1, 2, 3\}$

(b) Plants E_j, $j = \{1, 2\}$

Fig. 6. Modeling plants and specifications for the extended small factory.

We include the list of commands to generate the automata and design the monolithic and local modular supervisory control.

```
var s = Enumerable.Range(0, 2).Select(
    k => new State(k.ToString(),
        k == 0
            ? Marking.Marked
            : Marking.Unmarked))
        .ToArray();

var ev_a = Enumerable.Range(1, 3)
    .Select(k => new Event("a" + k,
        Controllability.Controllable)).ToArray();

var ev_b = Enumerable.Range(1, 3)
    .Select(k => new Event("b" + k,
        Controllability.Controllable)).ToArray();

var M1 = new DeterministicFiniteAutomaton(
    new[]{new Transition(s[0], ev_a[0], s[1]),
        new Transition(s[1], ev_b[0], s[0])}, s[0], "M1" );

var M2 = new DeterministicFiniteAutomaton(
    new[]{new Transition(s[0], ev_a[1], s[1]),
        new Transition(s[1], ev_b[1], s[0])}, s[0], "M2" );

var M3 = new DeterministicFiniteAutomaton(
    new[]{new Transition(s[0], ev_a[2], s[1]),
        new Transition(s[1], ev_b[2], s[0])}, s[0], "M3" );

var E1 = new DeterministicFiniteAutomaton(
    new[]{new Transition(s[0], ev_b[0], s[1]),
        new Transition(s[1], ev_a[1], s[0])}, s[0], "E1" );

var E2 = new DeterministicFiniteAutomaton(
    new[]{new Transition(s[0], ev_b[1], s[1]),
        new Transition(s[1], ev_a[2], s[0])}, s[0], "E2" );
```

```
var sup = DeterministicFiniteAutomaton
            .MonolithicSupervisor(new[]{ M1, M2, M3 },
                new[]{ E1, E2 } );

var sups = DeterministicFiniteAutomaton
            .LocalModularSupervisor(new[]{ M1, M2, M3 },
                new[]{ E1, E2 } );
```

5. PERFORMANCE TESTS

In order to show how *UltraDES* performs, four different problems in the literature were chosen and the classical approach of the SCT is applied, namely, the monolithic supervisor is designed. Two established academic softwares; TCT (Feng and Wonham, 2006), version 20160701 (release date: July 2016), and Supremica (Åkesson et al., 2006), version 201412081211 (release date: December 2014), were used to compare with *UltraDES*' results.

The first example is the *Cluster Tool* that models a semiconductor manufacturing system, introduced by Su et al. (2012). This is an interesting problem because it can be expanded by adding clusters, increasing the complexity of the problem to be solved. *UltraDES* gave results up to the size of 7 clusters. The other softwares did not give monolithic supervisors for this size of problem.

Two other examples were taken from the Supremica's examples library, the *Robot Assembly Cell* (Losito, 1999) and the *Automated Guided Vehicles* (Moody and Antsaklis, 1998). For the AGV plant, a specification 'ZoneX' that introduces a new zone at the input station was included, such as in the Supremica library. Also, the Flexible Manufacturing System (FMS) (Queiroz and Cury, 2000) is used.

The automata that model the FMS and Cluster Tool were initially implemented in *UltraDES* and later converted to TCT and Supremica. The remaining automata were converted from Supremica to *UltraDES* and then converted to TCT. All the experiments were ran in the same computer, a personal computer, with operating system Windows 7 64-bits, i5 and 6GB of RAM.

From Table 1 it can be noticed that *UltraDES* performs faster, typically, and it computes solution to bigger problems than the other two softwares experimented. For TCT, it is not possible to compare the computation time of the operations since the duration of each operation is given in seconds (rounded). That justifies the 0s in Table 1. Supremica performed faster for one of the examples and it was able to solve the cluster tool example up to 5 clusters. It should be mentioned that we were not able to obtain a monolithic supervisor for the cluster tool example with more than seven clusters, using *UltraDES*.

Table 2 shows the peak memory usage of *UltraDES*, Supremica and TCT. The data structure of *UltraDES* was able to store the supervisors using just a few megabytes. It used 43 times less memory than Supremica in the *Cluster Tools (5)* example.

It is important to mention that Supremica has an user interface which uses at least 160 MB, what may justify the large amount of memory for the small examples.

Plant	States	Transitions	UltraDES	TCT	Supremica
Cluster Tools (2)	45	74	0.04 s	0 s	0.03 s
Cluster Tools (3)	419	972	0.05 s	0 s	0.07 s
Cluster Tools (4)	4,184	12,630	0.21 s	233 s	0.89 s
Cluster Tools (5)	42,964	160,092	3.08 s	Does not compute	28.19 s
Cluster Tools (6)	447,998	1,988,053	55.98 s	Does not compute	Does not compute
Cluster Tools (7)	4,721,862	24,327,158	1,214.00 s	Does not compute	Does not compute
Robot Assembly Cell	4,675	20,752	0.08 s	5 s	0.13 s
FMS	45,504	200,124	1.18 s	Does not compute	7.15 s
Automated Guided Vehicles	11,489,280	68,667,392	358.40 s	Does not compute	Does not compute

Table 1. Execution time of monolithic supervisor design for nine examples.

Plant	UltraDES	TCT	Supremica
Cluster Tools (2)	13.8 MB	3.69 MB	167 MB
Cluster Tools (3)	14.4 MB	40.4 MB	172 MB
Cluster Tools (4)	18.3 MB	4.50 GB	248 MB
Cluster Tools (5)	37.5 MB	-	1.60 GB
Cluster Tools (6)	211 MB	-	-
Cluster Tools (7)	2.42 GB	-	-
Robot Assembly Cell	15.4 MB	3.74 GB	170 MB
FMS	29.1 MB	-	764 MB
Automated Guided Vehicles	1.50 GB	-	-

Table 2. Peak memory usage.

6. CONCLUSION

In this paper, a library named *UltraDES* is presented for the modeling, analysis and control of discrete event systems.

The library was developed to be literal, keeping the full names of the functionality. Moreover, the interface does not depend on the implementation, such that the creation of new functionality or the change of internal structures do not prevent the application to work with previous versions of *UltraDES*.

UltraDES has shown to solve bigger problems (Cluster Tool with up to 7 clusters) than the other softwares experimented, designing supervisors with more that eleven million states. Also, *UltraDES* provides results faster in most examples, than the other two softwares.

The current version of *UltraDES* can be downloaded at `https://github.com/lacsed/ultrades`. New contributions that may come are of interest. Currently, a tool to draw automata and also generate code for latex using the package tikz are being developed. Also, other algorithms of the literature are being implemented, such as the OP-verifier (Pena et al., 2014), supervisor reduction techniques and so on.

REFERENCES

Clavijo, L. B., Basilio, J. C., Carvalho, L. K., 2012. DESLAB: A Scientific Computing Program for Analysis and Synthesis of Discrete-Event Systems. In: Proceedings of the 11th International Workshop on Discrete Event Systems, WODES'12. Guadalajara, Mexico, pp. 349–355.

Feng, L., Wonham, W., Jul 2006. TCT: A Computation Tool for Supervisory Control Synthesis. In: Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06. Ann Arbor, MI, USA, pp. 388–389.

Hopcroft, J., Motwani, R., Ullman, J., 2001. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley.

Losito, M., 1999. An Architecture for Flexible Manufacturing Systems Applied to an Assembly Cell. Master's thesis, Politecnico di Milano, Italy.

Moody, J., Antsaklis, P. J., 1998. Supervisory Control of Discrete Event Systems Using Petri Nets. Vol. 8. Springer US.

Moor, T., Schmidt, K., Perk, S., 2008. libFAUDES - An Open Source C++ Library for Discrete Event Systems. In: Proceedings of the 9th International Workshop on Discrete Event Systems, WODES'08. Göteborg, Sweden, pp. 125–130.

Pena, P. N., Bravo, H. J., Da Cunha, A. E. C., Malik, R., Lafortune, S., Cury, J. E. R., 2014. Verification of the Observer Property in Discrete Event Systems. IEEE Transactions on Automatic Control 59 (8), 2176–2181.

Queiroz, M. H. D., Cury, J. E. R., 2000. Modular supervisory control of large scale discrete event systems. In: Proceedings of the 5th Workshop on Discrete Event Systems, WODES'00. Kluwer Academic, pp. 103–110.

Ramadge, P., Wonham, W., Jan. 1989. The Control of Discrete Event Systems. Proceedings of the IEEE 77 (1), 81–98.

Ricker, L., Lafortune, S., Genc, S., 2006. DESUMA: A Tool Integrating GIDDES and UMDES. In: Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06. Ann Arbor, MI, USA, pp. 392–393.

Su, R., van Schuppen, J., Rooda, J., 2012. The synthesis of time optimal supervisors by using heaps-of-pieces. IEEE Transactions on Automatic Control 57 (1), 105–118.

Åkesson, K., Fabian, M., Flordal, H., Malik, R., Jul 2006. Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. In: Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06. Ann Arbor, MI, USA, pp. 384–385.