Context Modelling and Partial-Order Reduction: Application to SDL Industrial Embedded Systems

Xavier Dumas*, Frédéric Boniol†, Philippe Dhaussy‡, Eric Bonnafous*
*CSSI, 3, rue du Prof. P. Vellas. 31300 Toulouse, name.surname@c-s.fr
†ONERA, 2, av. E. Belin. 31000 Toulouse, frederic.boniol@cert.fr
‡ENSIETA, 2, rue F. Verny. 29806 Brest, philippe.dhaussy@ensieta.fr

***Abstract*—In this paper we introduce a method combining system environment description and partial-order reduction for the verification of SDL systems. The aim of this work is to identify a new independence relation in case of open reactive systems: a system communicating asynchronously with its surrounding environment (context). The environment contrains the system to some interesting configurations so that to reducce its behaviour. We show how to characterize the context as a set of traces which can explode when the environment is composed by several concurrent actors. We then propose a solution for its reduction gathering the traces into equivalent Mazurkiewicz classes so that to make such approach scalable. We apply this method on two avionics case study.**

I. Introduction

The conception of a reactive system is a very difficult and arduous task requiring a lot of expertise from the engineers. In most cases they are classified as critical so that a little error could potentially lead to disastrous situations. As a consequence the validation process is a crucial step and must be reinforced all along the development cycle to go through the certification process. Unfortunately simulations and tests are becoming less accurate as systems are getting bigger and difficult to handle. System asynchronism and non determinism make the validation process even harder. Over this last twenty five years, formal methods have been investigated by countless researchers to improve and automate the verification of complex systems. The model checking method have known a big surge but quickly encountered a new disturbing phenomenon: the state space explosion.

A. The addressed problem: formal verification

Several model checker have been developed to help the verification of concurrent asynchronous systems. Especially, the SPIN model-checker [18] based on the formal language promela has been created by G. Holzman. The tool allows the verification of LTL properties encoded in "never claim" formalism and further converted into buchi automatas. Other techniques have been investigated in order to improve the performances of SPIN. For instance the state compression method or partial-order reduction contributed to further alleviate the combinatorial explosion [11]. In [19] the partial-order algorithm based on a depth first search (DFS) have been adapted to the bread first search (BFS) algorithm in the SPIN model-checker to exploit interesting properties inherent of the BFS. However the obtained reduction remains approximatively the same than in DFS algorithm.

Methods using symbolic model-checking based on BDD (Binary Decision Diagram [4]) have encountered a big success. The idea of the BDD is that all programs are composed by a set of bytes and thus manipulating and simplifying binary operations is very easy. Nevertheless the drawback of this method is pointed out with the necessity to transform the program into a boolean circuit possibly leading to irretrievable number of variables.

To deal with numerous variables and their possible ranges, work on abstraction methods [5] and abstract interpretation [6] have been investigated. Nevertheless, those two methods are not fully automatic and needs expertise from the user.

Other techniques emerged in the 90's taking advantage of the structure of the system rather than the global state space. Partial-order methods [9] aim at eliminating on the fly equivalent sequences of transitions in the global state space without modifying the falsity of the property under verification. Particularly in [11], [12] a method for computing on the fly the subset of transitions to be explored in each global state visited is described. This set is called persistent sets and contains the reduced set of transitions to be executed.

B. Verification by context modelling

One way to circumvent the problem of the combinatorial explosion resides in restricting the system behaviour with a specific surrounding environment describing the different configurations in which one want to observe the system. The idea behind context description is that properties are often related to specific use cases of the system corresponding to well-defined operational phases such as initialization, reconfiguration, degraded modes... so that it is not necessary for a given property to take into account all possible environment behaviour but only a subpart leading the system in critical and pertinent configurations. The context description thus allows a first limitation of the explored space search for the verification of a property, and hence a first reduction of the combinatorial explosion. In this article we consider the context approach for the verification but we also explore a second reduction step to alleviate the combinatorial explosion by exploiting symmetries in the system, in the context and in the properties.

C. Contribution of the paper

The context modelling allows to reduce the combinatorial explosion by restricting the system behaviour only to relevant part for the verification of a given property. However, in case the context is composed by several concurrent actors communicating asynchronously with the system, the number of scenarios to explore could potentially explose. For instance, the interleaving of n actors emitting a single event to the system leads to $n!$ possible scenarios.

In this paper we study an approach based on the reduction of the number of scenarios generated by the context relying on partial-order methods. The aim is to factorize a set of equivalent scenarios into a single one using the monoid structure of Mazurkiewicz traces [15]. To this purpose we define an independence relation between context events (intuitively, two events are independent if they can commute in the context without modifying the truth value of the property under verification).

In the sequel the document is built as follows. Section II introduces the formal basis of the independence relation: the monoid structure of Mazurkiewicz traces. Section III presents the SDL specification language used to model systems, and the CDL language to model their context. Section IV proposes the independence relation between events, from which we deduce an equivalence relation between set of scenarios (section V). Finally, section VI exposes experimental results obtained on two industrial avionics systems.

II. Partial-order semantics

A. Trace monoid

A trace monoid or free partially commutative monoid is a very simple mathematical algebraic structure often used in computer science and language theory. They are very well suited to describe and to model asynchronism in concurrent systems communicating by discrete events. In a monoid of trace, some events can be commuted when they are executed in parallel. The monoid representation is often used to describe computer systems where one can't cancell neither go back to a previous state of the program execution.

The monoid of traces has been generalized for the description of concurrent program by the theory of traces of Mazurkiewicz. Most partial order methods rely on the Mazurkiewicz trace semantics aiming at describing the system as a set of traces and gathering them into equivalent classes. Only one representant of each class is necessary for the verification of the entire system.

a) Definition of a trace monoid: Let Sig^* the free monoid containing all strings with the empty word ε. Let $I \subseteq Sig \times Sig$ a symmetric and irreflexive relation called the independence relation inducing a binary relation $\sim$ between pairwised events of Sig such that two words are equivalent: $u \sim_I v$ if and only if there exist $x, y \in Sig^*$ and $(a, b) \in Sig^2$ such that $u = xaby$ and $v = xbay$. $\mathbb{M}(Sig, I)$ is called the free partially commutative monoid (because not all events can commute for instance if they are produced by the same sequential process). It symbolizes the set of traces composed by pairwised independent events. The equivalent class of a sequence w is denoted $[w]$ and is called Mazurkiewicz trace.

B. Mazurkiewicz traces.

The idea behind Mazurkiewicz traces is that giving an independence relation, two traces are equivalent according to this relation if one trace can be obtained by successive transpositions of neigthboring letters from the alphabet.

b) Example: Let us consider the three following discrete events A, B and C, the independence relation and the corresponding scenario ABC. We assume that A and B are dependent with $I = \{(A, C); (B, C)\}$. Thus the equivalent class of ABC is defined by: $[ABC] = \{ABC, CAB, ACB\}$ which correponds to the set of traces such that A is always before B . Therefore only one representant of this class can be kept. However, the scenario BAC belongs to another class.

The aim of this article is to apply the Mazurkiewicz trace theory on systems modelled in SDL and their surrounding context modelled in CDL. The objective is to reduce the number of scenarios to explore for verifying a given property.

III. The modelling framework: SDL + CDL + observers

As described in the introduction we consider in this article the verification of a reactive asynchronous system closed by its environment. The model under verification is composed by the three following parts: (1) the model of the system described in a subset of the SDL formal language; (2) the model of the environment, described using the CDL language [13]; and (3) the properties to verify encoded by an observer, i.e, an automaton which produces a *reject* event whenever the property becomes false.

A. SDL for system modelling

SDL (Specification Description Language) is a formal hierarchical language standardized by the ITU-T organism firstly dedicated to the specification of protocol communication systems and further extended for the modelling of reactive real time systems with discrete events. SDL is a graphical language. It can be decomposed in four complementary views: a hierarchical view, a communication view, a behavioural view, and a data view. We only consider in the following the behavioural view. The semantics of SDL relies on an extended state machine with asynchronous passing value message. The reader not familiar with SDL can refer to [1]. In this article, we only focus on a subset of SDL (mono-process system, without timers and assignments).

In the following, we consider that a model SDL is a transition system $\mathcal{S} = \langle \Sigma, s_o, T, Sig_i, Sig_o, Sv \rangle$ where:

- Σ is a set of states, and $s_o \in \Sigma$ is the initial state;
- Sig_i and Sig_o are respectively the sets of input and output events; we suppose $Sig_i \cup Sig_o = \emptyset$;
- $T \subseteq \Sigma \times Sig_i \times Sig_o^* \times \Sigma$ is the set of transitions; a transition is a tuple (s_1, a, σ, s_2) where s_1 is the transition source state, s_2 is the destination state, a is the input event

activating the transition (the transition is fired if and only if the system is in the state s_1 and if a is received), and σ is the ordered sequence of output events produced when the transition is fired. (Sig_o^* is the set of finite words of output events);
- $Sv : \Sigma \rightarrow 2^{Sig_i}$ is a function associating each state $s \in \Sigma$ with a set (potentially empty) of input events saved (i.e., memorized) if they are received when the system is in the state s.

Notice that if an event e is received when the system is in a state s which does not wait for e (there is no transition guarded by e from s), then if $e \notin Sv(s)$, e is lost (i.e., it is not memorized in the input buffer). However, if $e \in Sv(s)$, it is memorized in the current buffer and it will be consumed (or saved again) in the next reached state.

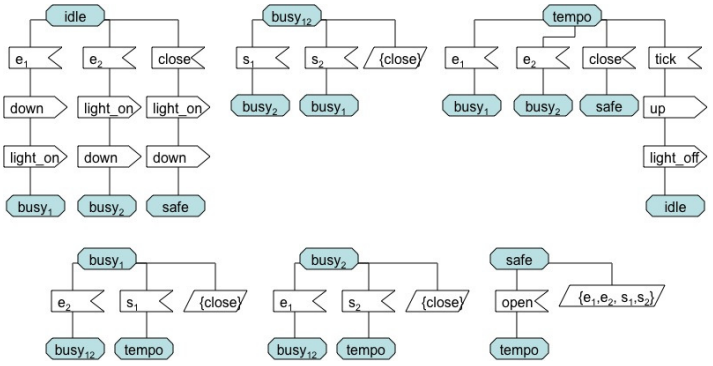


Fig. 1. Example of an SDL process: control of a rail-crossing

For instance, let us consider the system figure 1. This model represents an SDL process commanding the closing and the opening of a rail-crossing barriers (events *down* and *up*), and the ignition and extinction of the corresponding traffic lights (events *light_on* and *light_off*). Following the previous notations, this system is defined by $\mathcal{S} = \langle \Sigma, idle, T, Sig_i, Sig_o, Sv \rangle$ with

- $\Sigma = \{idle, busy_1, busy_2, busy_{12}, safe, tempo\}$;
- T is the set of 14 transitions represented figure 1;
- $Sig_i = \{e_1, s_1, e_2, s_2, close, open, tick\}$ are the input events of the model: e_i (resp. s_i) is sent by the train when it enters (resp. leaves) the level crossing section i with $i = 1, 2$; *close* is a closure order sent by a human operator (for emergency closure for example); *open* is a manual opening order sent after a *close* order;
- $Sig_o = \{down, up, light\_on, light\_off\}$;
- Sv is the save function (memorization) associating the empty set with *idle* and *tempo* states, associating $\{close\}$ with $busy_1$, $busy_2$, and $busy_{12}$, and associating $\{e_1, e_2, s_1, s_2\}$ with *safe* (graphically, Sv is defined by a beveled parallelogram associated; the absence of parallelogram implies $Sv(s) = \emptyset$).

For instance, the first transition from *idle* to $busy_1$ is guarded by the e_1 and produces the sequence *down*.*light_on*. Notice that if s_1 is received when the system is in *idle*, s_1 is lost (it is not saved). However, if e_1 is received in *safe*, then it is memorized until the arrival in *tempo* (after reception of *open*) and then consumed (in *tempo*). At last, to illustrate the technical point developed in this article, let us consider the scenarios e_1e_2 and e_2e_1 from state *idle*. From an observational point of view, those two scenarios have (almost) the same impact in the system. They both lead to the state $busy_{12}$ and produce the same outputs (*down* and *light_on*) but in a different order. However, if this order is irrelevant for the environment and for the property to verify, those two scenarios will be considered as equivalent.

B. CDL for context modelling

Let us now consider the closure of the system by its environment. This environment, called context, is described in CDL (Context Description Language) [13], [14]. A context is a finite process C producing and receiving events to and from the SDL system. Formally, a context is described by:

$$C ::= M \mid C_1.C_2 \mid C_1 + C_2 \mid C_1 \| C_2$$
$$M ::= \mathbf{0} \mid a!; M \mid a?; M$$

where $a!$ (resp. $a?$) is an event sent to the system (resp. waited by the context from the system). A context is either a Message Sequance Chart (MSC) M, either a sequential composition ($C_1.C_2$), either a non deterministic choice ($C_1 + C_2$), or a parallel composition ($C_1 \| C_2$). A MSC is either the empty MSC which doesn't do anything ($\mathbf{0}$), or sequence of emissions and receptions.

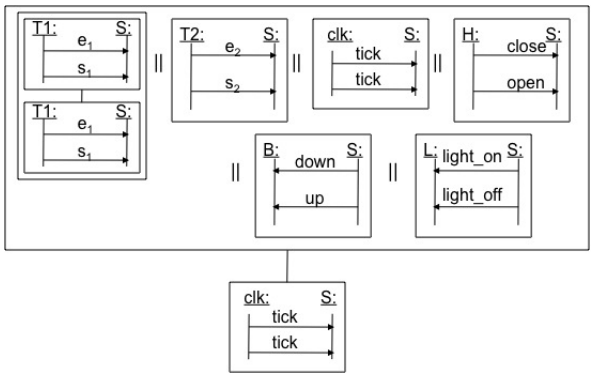


Fig. 2. Example of a CDL context

For instance, let us consider the context C of the level crossing system figure 1 graphically given figure 2. This context is formally defined by $C = (((T_1.T_1) \| T_2 \| clk \| H \| B \| L).(tick!.\mathbf{0}))$ with $T_1 = e_1!; s_1!; \mathbf{0}$, $T_2 = e_2!; s_2!; \mathbf{0}$, $clk = tick!; tick!; \mathbf{0}$ and $H = close!; open!; \mathbf{0}$, $B = down?; up?; \mathbf{0}$ and $L = light\_on?; light\_off?; \mathbf{0}$. This context is composed in parallel by (a) two successive trains on rail 1 (T_1 and T_1), (b) a train T_2 on the rail 2, (c) a human operator (H) sending a closure order *close* and then an opening order *open*, (d) a clock clapping 2 *tick*, (e) a barrier waiting for *down* and then *up* (the barrier is initially open), and (f) a traffic light waiting *light_on* and then *light_off* (the barrier is initially

off). At the end, one tick is sent to the system. Notice that this context is restricting the behaviour of the system allowing only one train simultaneouly on each rail. The user interested to verify the the level crossing model with respect to context involving only trains on rail 1, could further restrict C by removing the T_2 occurrence. In this case, the verification only focuses on the system figure 1 without exploring transitions beginning by e_2 and s_2. Such a restriction leads to reduction of the state explosion problem.

From a semantical point of view, a context is a parallel process communicating asynchronously with the system, memorizing its input events (from the system) in an ordered input buffer. The behaviour of a context C is writen by the relation $(C, B) \xrightarrow{a} (C', B')$ to mean that the context C associated with the buffer B (a FIFO queue of input events emitted by the system to its context) "makes" a (which can be a sending or a receiving event, or an empty event called $null_\sigma$ if C doesn't evolve) and then becomes the new context C' with a new input buffer B'. CDL follows the classical asynchronous semantics of process algebra like CCS. For instance, if an actor (a parallel sub-context like T_2 figure 2) can produce an event, then all the context can emit this event. If several actors can produce several events, then these events are sequentialy produced in a arbitrary order (according to the interleaving semantics of asynchronous parallelism). In the same way, if the context received an event a (stored in the input buffer) and if there is an actor A waiting for a, then a is consumed (removed from the input buffer) by A and A continues its behaviour. If several actors are waiting for a, only one of them consumes it (the choice is non deterministic). Notice that such an asynchronous semantics of parallelism (based on interleaving) leads to an explosion of possible scenarios for a given context. For instance, the context figure 2 specifies more than 37,800 different scenarios. However, it is obvious that a great number of these scenarios have the same impact on the system and the property to verify. In that sense, they are equivalent. The idea developed in this article is to reduce the context by removing equivalent scenarios.

C. Composition context + system

The model under verification is then composed of the system specified in SDL and its context specified in CDL. The composition between the system and its context is, here again, an asynchronous parallel composition. The system and its context have their own input buffer, and they communicate by producing and consuning asynchronous events. The global behaviour of a system $\mathcal{S}$ and a context C is written by the relation $< (C, B_1) \mid (s, \mathcal{S}, B_2) > \xrightarrow[\sigma]{a} < (C', B_1') \mid (s', \mathcal{S}, B_2') >$ where B_1 is the input buffer of the context, B_2 is the input buffer of the system, s is the current state (at the beginning of the behaviour) of the system. The system and the context evolves then in parallel: C sends a to $\mathcal{S}$ and becomes the new context C'; a is stored in the input buffer of the system which becomes B_2'. The system reacts (to a previous event stored in its buffer) by producing an output event sequences σ to the context, and arrives in a another internal state s'. The sequence σ is stored (in that order) in the input buffer of the context and will be consumed in the next steps. The main rules defining the composition between the system and its context can be informally summarized by:

- If $\mathcal{S}$ can react to the event stored in the head of its input buffer, the corresponding transition is fired, and the sequence σ produced by this transition is sent to the context. $\mathcal{S}$ then reaches the destination state of the transition, and σ is put at the end of the buffer of C.
- If C can emit a, it evolves and a is queued in the buffer of $\mathcal{S}$.
- At last, if C can consume the event stored in the head of its input buffer, it does it, and it reaches its next step.

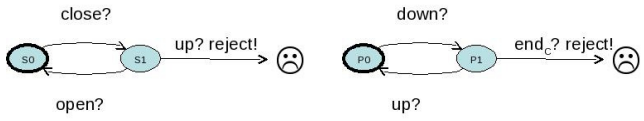
Let us note $[\![C|\mathcal{S}]\!]$ the set of *runs* (i.e., traces) of the system $\mathcal{S}$ closed by the context C from the initial state of $\mathcal{S}$.

Let us notice that contexts are defined as the composition of finite MSC. The scenarios of the context are then necessarily finite. Consequently *runs* of $[\![C|\mathcal{S}]\!]$ are also finite. In the following, we consider that all runs are ended by the terminal event end_C. Observing this terminal event allows to "test" bounded liveness properties.

D. The observers

The third part of the formalization relies on the expression of the properties to verify. In this article, we only consider reachability properties, expressed by observers. For the sake of simplicity, observers are specified as SDL automata which *observe* events exchanged by the system and its surrounding context C (and thus events occurring in *runs* of $[\![C|\mathcal{S}]\!]$) and which produce a *reject* event whenever the property is false.

As example let us consider again the system figure 1 and its context figure 2. Let us consider the two following properties: after a *close* order, the system can not produce any occurrence of *up* before receiving *open* (P_1), and at the end of the context, barriers are opened (P_2). Those two properties are formalized by the observers O_1 and O_2 figures 3(a) and 3(b).



(a) Observer O_1 (property P_1) (b) Observer O_2 (property P_2)

Fig. 3. Examples of observers

From a formal point of view, an observer is a SDL automaton $\mathcal{O} = \langle \Sigma_o, init_o, T_o, Sig, \{reject\}, Sv_o \rangle$ which can produce a single output event called *reject*. One says that $\mathcal{S}$ closed by C satisfies $\mathcal{O}$, denoted $C|\mathcal{S} \models \mathcal{O}$, if and only if there is no run from $[\![C|\mathcal{S}]\!]$ which, when composed with $\mathcal{O}$, leads to the *reject* event.

IV. An independence relation between pairwised events

A. Overview

As mentioned in the introduction, the main drawback in formal verification is the combinatorial explosion induced when generating the global state space. A solution exposed in [13], [14], and summarized in section III-B, consists to circumvent this explosion by closing the system under verification by its operational context. Even though this method reduces the global state space, it is still not scalable for real industrial systems where contexts are themselves complex. Hence a complementary approach is necessary.

As described in introduction, the aim of this article is to study a reduction method of the global state space relying on the exploitation of symmetries of the model and its context. Those symmetries result of independence relation between events emitted from the context to the system. Let us consider again the system depicted figure 1 together with its context figure 2, and the two properties 3(a) and 3(b). This system presents some symmetries. For instance let us consider the two scenarios e_1e_2 and e_2e_1. In every state of the system, the execution of those two scenarios leads to the same state. As exemple, from *idle*, receptions of e_1e_2 and e_2e_1 lead both to the same state $busy_{12}$. Note however that these two reactions produce the same output events (*down* and *light_on*) but in a different order. Nevertheless these events are consumed by the environment by two different and concurrent MSC. Moreover *light_on* doesn't modify the truth or falsity of the observers $\mathcal{O}_1$ and $\mathcal{O}_2$. Hence the two sequences $down \cdot light\_on$ and $light\_on \cdot down$ can be considered as equivalent with respect to the system, the context, and the observers. As a consequence we can say that e_1 and e_2 commute (we'll say that they are independent). Then it is not necessary to "test" both e_1e_2 and e_2e_1 but only one of them.

Let us notice that to prove that e_1 and e_2 commute, it is not necessary to explore the global automaton obtained by composing the system, its context and its observers, but it only requires a static analysis the SDL system, the context, and the observers. These models are designed by engineers, hence one can expect this exploration won't lead to a combinatorial explosion. This is the main interest of the proposed approach.

B. The independence relation

We'll say that two events a_1 and a_2 produced by the context are independent[1] if and only if:

- they are independent for the SDL system, meaning that they satisfy the "diamond property" (input scenarios a_1a_2 and a_2a_1 lead to the same state);
- these events and the output sequences produced by the system in response to a_1a_2 and to a_2a_1 are independent with respect to the observer;
- and these events and the output sequences produced by the system in response to a_1a_2 and to a_2a_1 are independent with respect to the context.

In the sequel we consider a system $\mathcal{S} = \langle \Sigma, init, T, Sig_i, Sig_o, Sv \rangle$, an observer $\mathcal{O} = \langle \Sigma_o, init_o, T_o, Sig, \{reject\}, Sv_o \rangle$, and a context C. We denote by $tgt_{\mathcal{S}}(a_1.a_2, s)$ (respectively $tgt_{\mathcal{O}}(a_1.a_2, s)$) the set of $\mathcal{S}$ states (respectively $\mathcal{S}$ states) reached after executing the scenario $a_1.a_2$ from state s. We also denote by $|w|$ the length of a word w and by $|w|_a$ the occurence of a in the word w. The first independence (with respect to the system) is denoted $I_{\mathcal{S}}$. The second one (with respect to the observer) is denoted by $I_{\mathcal{O}}$. The last one (with respect to the context) is denoted I_C.

1) System independence: ($I_{\mathcal{S}}$): Two events from the context are independent whenever their consumption by the system lead to the same set of states. For instance in the picture 1, one can observe that e_1 and e_2 are globally independent because their consumption order is irrelevant in all states. In the sequel we denote by σ_1 the scenario $e_1.e_2$ and by σ_2 the scenario $e_2.e_1$:

state	$tgt_{\mathcal{S}}(\sigma_1, state)$	$tgt_{\mathcal{S}}(\sigma_2, state)$	$I_{\mathcal{S}}(e_1, e_2, s)$
idle	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
$busy_{12}$	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
tempo	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
$busy_1$	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
$busy_2$	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
safe	$\{safe\}$	$\{safe\}$	*true*

In all states s, $tgt_{\mathcal{S}}(\sigma_1, s) = tgt_{\mathcal{S}}(\sigma_2, s)$. e_1 et e_2 are then independent with respect to $\mathcal{S}$.

Let us now consider events e_1 and $close$. there exists a state (*idle*) such that the set of reached states is different: $tgt_{\mathcal{S}}(e_1.close, idle) = busy_1$ and $tgt_{\mathcal{S}}(close.e_1, idle) = safe$. They are then dependent (they do not commute).

At the end, we obtain the following set of pairwise independent events $I_{\mathcal{S}} = \{(e1, e2), (e1, open), (e2, open), (e1, s2), (e2, s1), (s1, s2), (s1, open), (s2, open)\}$.

The method computing $I_{\mathcal{S}}$ is given by algorithm 1.

2) Observer independence: ($I_{\mathcal{O}}$): Up to this point of the article, we only cared of independence between events sent by the context to the system. In most cases, the system of transitions sends in return a set of output messages. For instance, we have seen that e_1 and e_2 were $I_{\mathcal{S}}$.

Now let us consider the *idle* state from the picture 1. This is the only state where the system responds to the input events e_1 and e_2. The question we have to wonder is the following: can the outputs produced after receiving e_1 and e_2 modify the independence relation between e_1 and e_2?

The independence relation depends on the property to be fulfilled. For instance if there exists a property verifying wether $down$ is always received before $light\_on$ or if the consumption of e_1 leads to the sending of $down$, then transitions containing e_1 and e_2 are observer dependent thus breaking one of the two previous independence. For instance, if e_2 is consumed in $idle$ before e_1, then $down$ is sent and in $busy_2$

[1] A complete formalization of the following independence relation could be found in [7].

e_1 is consumed. This is clearly the opposite of the property : e_1 implies $down$.

Let a_1 and a_2 two input events from $\mathcal{S}$. Let us consider a state s from this system; let us also consider the traces obtained when simulatinge $\mathcal{S}$ in s with $a_1 \cdot a_2$ and then with $a_2 \cdot a_1$. By definition, these traces are: $out_{\mathcal{S}}(a_1 \cdot a_2, s)$ and $out_{\mathcal{S}}(a_2 \cdot a_1, s)$.

Two events from the context a_1 and a_2 are observer independent if the traces generated by $a_1 \cdot a_2$ and $a_2 \cdot a_1$ (containing events emitted by the system in response) produce the same internal behaviour of the observer, meaning they reach the same state.

It's obvious that, if a_1 and a_2 are observer independent, then either both $a_1 \cdot a_2$ and $a_2 \cdot a_1$ satisfy the observer, either none of them statisfy the observer.

For instance the events e_1 and e_2 from the system 1 are $I_{\mathcal{S}}\mathcal{O}_2$. The scenarios generated by $e_1 \cdot e_2$ and $e_2 \cdot e_1$, if they are restricted to observed events by $\mathcal{O}_2$, are equal to the single event *down*. It is straightforward that these observers produce the same behaviour on $\mathcal{O}_2$ (and thus the same state).

Let us consider $\sigma_1 = close.open$ and $\sigma_2 = open.close$. In the picture 3(a), $tgt_o(\sigma_1, s_0) = \{s_0\}$ and $tgt_o(\sigma_2, s_0) = \{s_1\}$. As a consequence we obtain: $I_o(close, open) = FALSE$ It's obvious that, if a_1 and a_2 are observer independent, then either both $a_1 \cdot a_2$ and $a_2 \cdot a_1$ satisfy the observer, either none of them statisfy the observer.

For instance the events e_1 and e_2 from the system 1 are $I_{\mathcal{S}}\mathcal{O}_2$. The scenarios generated by $e_1 \cdot e_2$ and $e_2 \cdot e_1$, if they are restricted to observed events by $\mathcal{O}_2$, are equal to the single event *down*. It is straightforward that these observers produce the same behaviour on $\mathcal{O}_2$ (and thus the same state).

The observer independence is described in the figure the algorithm 2.

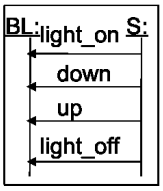
3) Context independence (I_C): Let us consifer two input events a_1 and a_2 from $\mathcal{S}$, and a state s of $\mathcal{S}$. Let us consider the set of output sequences produced by $\mathcal{S}$ in s in response to $a_1 \cdot a_2$ and to $a_2 \cdot a_1$ respectively. Let us denote $out_{\mathcal{S}}(a_1 \cdot a_2, s)$ and $out_{\mathcal{S}}(a_2 \cdot a_1, s)$ these sets.

In this case we can't argue as previously on the reached context states by those two sets of scenarios; the number of context states could be potentially huge, the construction of this independence relation would face again a new combinatorial explosion.

Instead we introduce a stronger independence criterion only needing a static analysis of the context model. We say that a_1 and a_2 are context independent with respect to C if the two sets of traces restricted to the vocabulary of C (we remove events not taken into account by C) are such that for all trace r_1 from the first set (generated by the first scenario), there exists a trace r_2 from the second set

- which differs from r_1 only by the order of events in the trace
- and such that if (b, b') is a couple of events different from (a_1, a_2) or (a_2, a_1), and ordered in a different manner in r_1 and in r_2 (for instance b is before b' in r_1, and inversely in r_2), then b and b' appear in parallel components in the context which doesn't specify the order of b and b'.

The idea is to distinguish among the scenarios,those which are inducing interactions in the same order than specified by the context, and those which differ. Let us consider the context C given figure 2. Scenarios e_1e_2 and e_2e_1 produce the sequences *down·light_on* and *light_on· down* respectively. These two sequences differ by the order between the two events. However, *down* and *light_on* are received in the environment by two independent actors (in parallel). Their order is then irrelevant. Consequently, e_1 and e_2 are context independent. The method computing I_C is given by algorithm 3. Let us now consider a new context C' (fig. 4) obtained by removing from C actors B and L and by replacing them by the actor BL.



In this case *down* and *light_on* are sequentially received by the context. As a consequence, only scenario (e_2e_1) is compliant with this new context. The other leads to a context deadlock. e_1 and e_2 are then dependent with respect to C'.

Fig. 4. One more parallel actor

Finally, let a_1 and a_2 two input events from $\mathcal{S}$, we say that a_1 and a_2 are independent, denoted $(a_1, a_2) \in I_{glob}$ if they are system independent, observer independent and context independent: $I_{glob} \stackrel{\text{def}}{=} I_{\mathcal{S}} \cap I_{\mathcal{O}} \cap I_{\mathcal{C}}$

Algorithm 1: System Independence algorithm

```
I_S(a, b, S):
foreach state s ∈ S.Σ do
    if (¬(tgt(a.b, s) = tgt(b.a, s))) then
        return false;
    end
end
return true;

tgt(a.b, s):
σ1 ← ∅; σ2 ← ∅;
if (a ∈ s.inputs) then
    σ1 ← tgt(a, s);
    foreach state st ∈ σ1 do
        σ2 ← (σ2 ∪ tgt(b, st));
    end
    return σ2;
else if (a ∉ s.inputs && a ∈ Sv(s)) then
    return tgt(b.a, st);
else if ((a, b) ∈ Sv(st)) then st.buf ← a, b; return st;
else
    return tgt(b, st);
end
```

V. Scenarios equivalence

An independence relation between events have been identified. We can now extend the independence between scenarios of input events relying on Mazurkiewicz traces.

Algorithm 2: Observer Independence algorithm

```
I_o(a, b, S, O):
r1 ← ∅; r2 ← ∅;
foreach state s ∈ S.Σ do
    r1 ← out_S(a.b, s); r2 ← out_S(b.a, s);
    if (¬(indep_o(r1, r2, s, O) && indep_o(r2, r1, s, O))
    then
        return false;
    end
end
return true;

indep_o(r1, r2, s, O) :
foreach trace t1 ∈ r1 do
    foreach trace t2 ∈ r2 do
        foreach state s_o ∈ O do
            if (tgt_o(r1, s_o) == tgt_o(r2, s_o)) then
                return true;
            end
        end
    end
end
return false;
```

Algorithm 3: Context Independence algorithm

```
I_c(a, b, S, C):
r1 ← ∅; r2 ← ∅;
foreach state s ∈ S.Σ do
    r1 ← out_S(a.b, s); r2 ← out_S(b.a, s);
    if (¬(indep_c(r1, r2, s, C) && indep_c(r2, r1, s, C))
    then
        return false;
    end
end
return true;

indep_c(r1, r2, s, C) :
foreach trace t1 ∈ r1 do
    foreach trace t2 ∈ r2 do
        if (equiv(t1, t2)) then
            return true;
        end
    end
end
return false;

equiv(t1, t2) :
if (¬(|t1| == |t2|)) then
    return false;
else
    foreach letter a ∈ t1 do
        if (¬(|t1|_a == |t2|_a) then
            return false;
        else
            foreach (a, b) ∈ t1 × t2 do
                foreach msc ∈ C do
                    if ((a, b) ∈ msc) then
                        return false;
                    end
                end
            end
        end
    end
end
return true;
```

a) Application to the scenarios reduction: Let $\mathcal{S}$ a system SDL, C its context and $\mathcal{O}$ an observer to be verified. Considering the free monoid partially commutative in which the words are the scenarios of events produced by C with the independence relation I_{glob}, applying [17] method it is now straightforward to compute the set of distinct scenarios (i.e., which have a different behaviour on the system or in the observer). Those scenarios are the equivalent classes $[w]$ of the monoid $\mathbb{M}(Sig_i, I_{glob})$ where Sig_i is the set of input events of the system.

The important result of this reduction, detailed in the following, is that all scenarios from a same class satisfy in the same manner (i.e., satisfy or not) the observer $\mathcal{O}$.

Proposition 1: Let a class $[w]$ of equivalent scenarios in the sense of the free monoid partially commutative $\mathbb{M}(Sig_i, I_{glob})$, then: $\forall C_{\sigma_1}, C_{\sigma_2} \in [w], \quad C_{\sigma_1} \| (init, \mathcal{S}) \models \mathcal{O} \Leftrightarrow C_{\sigma_2} \| (init, \mathcal{S}) \models \mathcal{O}$ where C_{σ_1} and C_{σ_2} are sequential contexts respectively playing the scenarios σ_1 and σ_2.

This is a fundamental result. It allows to reduce, sometimes considerably (see the results in the next section) the number of different scenarios from the context, and finally the state space of the verification.

b) Foata normal form: In order to compute the equivalent classes of Mazurkiewicz traces, we need a decision procedure to compute them. The idea behind Foata normal form is that given a trace and a set of rewriting rules, this trace has a unique rewriting trace called the Foata form. All the traces describing the same Foata form then belong to the same class.

A word x of Σ^* is in the Foata normal form if it is the empty word or it there exist an integer $n > 0$ and non empty words $x_i (1 \leq i \leq n)$ such that

1) $x = x_1 ... x_n$,
2) for each i, the word x_i is a product of pairwise independent letters and x_i is minimal with respect to the lexicographic ordering,
3) For each $1 \leq i \leq n$ for each letter a of x_{i+1} there exists a letter b of x_i such that (a, b) are dependent

c) Decision procedure to compute Foata normal form: Considering a monoid of traces $M(\Sigma, I)$, a stack is used for each letter. The trace is scanned from right to left. For each letter, it is pushed in its own stack. A marker is pushed on each stack dependent with the current scanned letter. To get the Foata normal form, the set of all letters being on the top

of all stack are taken. They are arranged in a lexicographic order and poped from the stack. A marker is poped in every stack coresponding to a letter which is dependent with a letter of the selected set. The procedure is depicted in algorithm 4.

d) example:

*	b		
a	*	*	*
a	*	*	d
*	*	*	*
*	b	c	*

Let us consider the following alphabet $\Sigma = \{a, b, c, d\}$ and the independence relation $I = \{(a, d), (b, c)\}$; the Foata normal form of badacb is (b)(ad)(a)(bc).

VI. Experimental results

We applied the methodology presented in this paper to three case studies from which two subsets of real avionics application.

System	Process	Actors	Scn Initial	Scn reduction
TRAIN	1	4	37800	3300
CPDLC	1	3	840	2
AFN	3	4	1238	301

e) Rail-crossing system: The system TRAIN introduced above is composed by a single process with 6 SDL states, 7 input events and 3 output events. This simple system transformed into IF by the IFx model-checker generates 300.000 states in the global state space. On the other part, the generation of the scenarios when the context is expanded lead to 37.800 scenarios. At last the application of the method presented in this article relying on the independence I_{glob} implying a reduction of 91%. At the end, only 3300 scenarios have to be simulated on the system. (3300 little "model-checking").

f) Controller Pilot Data Link Communication system: The second line corresponds to the results obtained on a unique avionics process CPDLC (Controller Pilot Data Link Communication) allowing the exchange of messages between pilots and control tower by datalink. The simplified version of the SDL process is depicted on the picture 5(a). The system begins by sending a starting request with the message $startReq$ to the global system and reaches the state $waitStart$ waiting for the response. There are two different manner to start the system: a cold start corresponding to the signal $start1$ and a reset start correspnding to the sugnal $start2$. One can notice those two events lead to the same output message and to the same destination state. Hence, one can reasonably expect the order of those two events to be irrelevant. The consumption of these two events lead to the registration request to the server 1 with the message $sv1Reg$. This server connects the CPDLC with different on board services such the FMS. Let us notice that this server has to be started before all the other applications. Once the CPDLC has been registered to the server 1 by the confirmation message $sv1Cf$ in the state $waitSv1$ the CPDLC is asking for the registration of the AFN application allowing the notification of an ATC center with the message $AFNReg$. Then the process reaches the state wait-AFN waiting for the effective registration with the message $AFNCf$. From now on, the CPDLC has to be registered to one more server handling the IHM by the message $sv2Reg$ and waiting in reponse $sv2Cf$ as a confirmation. At this moment,

Algorithm 4: Foata Normal Form Algorithm

$FoataNormalForm(tr, I)$:
$res \leftarrow \emptyset; Nb \leftarrow 0; hlex \leftarrow \emptyset; hstack \leftarrow \emptyset; list \leftarrow \emptyset;$
$listLex \leftarrow \emptyset; elem \leftarrow \emptyset$

/* To simulate the lexicographic order of the letters of the alphabet Σ, we associate each letter with an integer. To compute the foata normal form, one stack is created for each letter of the alphabet */

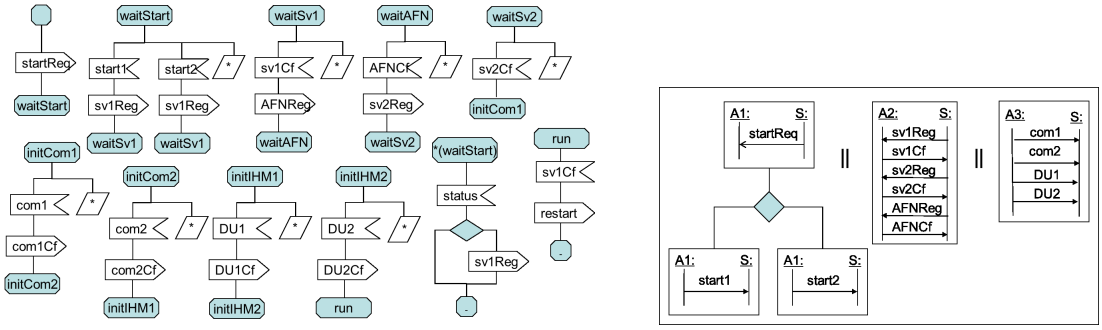
foreach $a \in \Sigma$ **do**
  if $(a \notin hlex.key)$ **then**
    $hlex.push(a, Nb++)$;
    $hstack.push(a, createStack(a))$;
  end
end

/* Each letter from tr is scanned from right to left. Each letter is pushed in its own stack and a marker "*" is pushed in all stacks where the letter is dependent with all the current scanned one */

for $i \leftarrow (tr.length - 1)$ **to** 0 **do**
  $elem \leftarrow tr.i$; $(hstack.get(elem)).push(elem)$;
  foreach $key\ k \in hstack.key\ with\ key \neq elem$ **do**
    if $(elem, k) \notin I$ **then**
      $(hstack.get(key)).push("*")$
    end
  end
end

/* To get the Foata form we pop all letters on top of each stack and organize them in lexicographic order. We pop a marker on each stack where the letter is dependent with the letters on top of the stack */

while $hstack \neq \emptyset$ **do**
  $list \leftarrow getTopStackElem(hstack)$;
  $listLex \leftarrow getLexOrder(list)$; $res.add(listLex)$;
  foreach $letter\ l \in listLex$ **do**
    foreach $key\ k \in hstack.key$ **do**
      if $(l, k) \notin I$ **then**
        $(hstack.get(k)).pop$;
      end
    end
  end
end
$return\ res$;



(a) Simplified CPDLC manager process

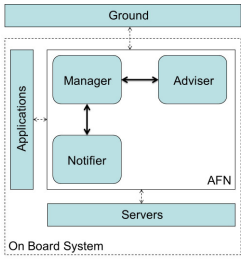
(b) CPDLC context

Fig. 5. CPDLC case study

the system has to establish the communication protocol to send and receive messages to and from the ground. This is done by sending messages $com1$ and $com2$ to the two corresponding communication layers and waiting their confirmation with the messages $com1Cf$ respectively $com2Cf$. Finally, the CPDLC initializes two scrren plays where the pilot and copilot could type and recevie messages. This step is done with the messages $DU1$ and $DU2$ with their corresponding confirmation $DU1Cf$ and $DU2Cf$. The special state $*(waitStart)$ means that all transitions described in this state can be executed in all states except in the state $waitStart$. In fact, the onboard system services offered via the the server 1 could potentially be out of service or the server 1 could be unavailable. In this case, the status of the server changes and a restart of the server is done. In the running state, all initializations have been made. Hence when the server 1 restarts, the CPDLC has to deal with the confirmation of the server 1 and restarts the services associated to this server.

The main difference in the save clauses of the CPDLC system is that all messages are saved (except those which can be consumed in the current state) in almost every states contrary to the TRAIN system where only a small subset of the overall messages where saved. In fact the CPDLC system can't determinate before the runtime execution which application sends a request first so that it has to predict every possible order of requests of every application. The general saving clause is dramatic for the global state space where in the worst case for n possible messages it could lead to (n+1)! intermediate states (where an intermediate states is the current state with a saved message added) We obtain approximatively 43828 states in the global SDL automata which has to be further composed with the context. In the picture 5(b) we have defined a particular context composed by three actors each one representing several applications to verify the correct initialization of the system from the initial state to the final state in a particular configuration. We have identified 3 main actors: the global system A1, the servers and the AFN application A2, the communication and the display unit A3. Initially, the CPDLC has to deal with every possible order of request coming from the different applications after the server 1 has been started. One can see that we modified the order of initialization of the AFN and the server 2 in the context A2. to potentially induce an expected behaviour of the system. Now if we look at necessary scenarios for the verification of the system, we can see that only 2 among the 840 are required which corresponds to a reduction of magnitude 99,9%. In fact those two scenarios represents the two possible initialisation of the system: with a start1 or a start2. he reason of this huge reduction comes from the system where in almost every state all the messages are seved making the order of the messages included in the scenario irrelevant since all messages are saved except those one which can be executed. As a consequence the save instances make the system more deterministic and more linear and thus reinforce the robusteness since it can be tolerant to the environment possible "mistakes" when sending their request in an irrelevant order.

g) Atis Facility Notification system: On the last line of the array, we present the results obtained for another avionics application: AFN(Atis Facility Notification) allowing the notification of an airbone to the ATC center and also the exchange of informations about the cockpit (software versions, serial number, tail number, ...). For information we present this time the reductions obtained in case of multiprocess system. The AFN system depicted in the picture 6 is componed by three processes.



- The Manager instanciating the two other processes;
- The Notifier which main rule is to make the connection with an ATC center;
- The Adviser which handles the migration from one center to another one.

Fig. 6. AFN System and its surrounding environment

From a context point of view, the AFN can communicate with the ground for the notification of a center. The AFN can communicate with the CPDLC to norify the current center. The AFN can exchange messages with different servers; for instance to gather some data informations or to display some

informations. We thus identified 4 actors. The reduction are less impressive because this time communications between inter processes can interfere in the independence computation. Here, the 3 processes contain approximatively 50 states each when expanded with the save clauses. The systems is modelled by the engineer and thus the complexity of the system is masterised and the number of SDL states will not be in general very important. The difference with the previous process resides in the fact that in the operational phase, the succession of the operations are less linear and less deterministic. Above all in the multiprocess case, the independence relation are parasited by the possible communication inter and extra processes. For instance, two contexts messages independent in a process can lead to different behaviour in the other process once output messages have been sent. In this case one has to verify that output messages sent in the transitions where the independent messages are consumed are independent in further processes. As an illustration let us consider the picture 6. The ground and the applications can communicate with the process manager so that to lead this process in a particular SDL state. Let us consider a a message sent from the ground to the process Adviser and b a message sent by an application from the bloc applications to the process manager. From a context point of view, those two events are independent since they are sent to a different process. In reality this is not always the case when for example the processes communicate. In this example, suppose that when a is consumed, then the process manager reaches a new state wiaiting the message c from the Adviser. Suppose that when consuming b the process Adviser sends c to the manager. Now let us consider the scenarios $a.b$ of the environment. If a is received by the process manager, then it reaches a state waiting for c. Then c is consumed and a new state is reached denoted. Now if we consider the scenarios $b.a$, this time, when b is consumed, c is sent to the process manager, which is waiting for a. As a consequence c is lost and the process remains in the same state. At the end, a is received and the process goes to the state waiting for c which will never be received. Consequently the behaviours indeuced by those two scenarios which where equivalent in case of mono process system are now dependent. We thus developped a new independece relation related to the multi process case. Eventually we obtained a reduction of magnitude 75,7% for the AFN system.

To summarize the multiprocess case, each process behaves like a context itself for neighboring processes. As a consequence one must figure out if the output messages of a process are independent in the neighbouring ones.

VII. Conclusion and perspectives

In this article we studied a partial-order method together combined with the context and observer method from the CDL language. We have experimented this method in case of monoprocess systems. This work has been extended to the multi process case and experimented on the AFN[2] avionics application. The results show the interest of the approach. In this paper we didn't introduce the data part: the affectations and the clock values. From a data point of view, we can expect a little loss of reduction because we already take into account the decision in our independence relation. In fact when a variable change of value, the behaviour of the system is modified only when a decision is present. The second point focus on the SDL timers. The SDL timer semantics will have to be determinated in order to obtain a new independence relation based on temporal scopes equivalence.

[2]Atis Facility Notification

References

[1] ITU-T Recommendation Z.100: Functional Specification and Description Language (SDL). ITU, Geneva, 1994

[2] S. Graf, B. Steffen, G. Lütggen: Compositional Minimisation of Finite State Systems using Interface Specifications, Formal Aspect of Computation, vol. 8, pp. 607–616, 1996

[3] S.C. Cheung and J. Kramer: Incorporation of Context Constraints for Compositional Reachability Analysis,1996

[4] R. E. Bryant: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Computing Survey, vol.24(3), pp.293–318,1992

[5] E. Clarke and A. Fehnker and Z. Han and B. Krogh and J. Ouaknine and O. Stursberg and M. Theobald: Abstraction and counterexample-guided refinement in model checking of hybrid systems. In: International Journal of Foundations of Computer Science, 14(4), 2003

[6] Cousot, P. and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 238–252, Los Angeles, California,ACM Press, New York, NY,1977.

[7] X. Dumas, F. Boniol, P. Dhaussy, E. Bonnafous. Context Modelling and Partial Order Application for Efficient SDL System Verification, submitted to the International Colloquium on Theoretical Aspects of Computing (ICTAC) 2010.

[8] E.A. Emerson and S. Jha and D. Peled: Combining Partial Order and Symmetry Reductions. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer Verlag, LNCS 1217, Enschede, The Netherlands, E. Brinksma, pp. 19–34, 1997

[9] D. Peled: Ten Years of Partial Order Reduction. In: CAV, pp. 17–28, 1998

[10] R. Alur and R. K. Brayton and Thomas A. Henzinger and Shaz Qadeer and Sriram K. Rajamani: Partial-Order Reduction in Symbolic State Space Exploration. In: Computer Aided Verification, pp. 340–351, 1997

[11] P. Godefroid: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. In: Springer-Verlag Inc., vol. 1032, pp. 142, New York, NY, USA, 1996

[12] A. Valmari: Stubborn sets for reduced state space generation. In: Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, Springer-Verlag, pp. 491–515, London, UK, 1991

[13] Dhaussy P, Pillain P-y, Creff S, Raji A, Le Traon Y et Baudry B.: Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation. In Springer Verlag Lecture Notes in Computer Science, Model Driven Engineering Languages and Systems, vol. 5795 (2009), pp. 438–452, 2009

[14] J.C. Roger: Exploitation de contextes et d'observateurs pour la validation formelle de modèles, PhD thesis, ENSIETA, 2006

[15] A. Mazurkiewicz, Introduction to trace theory, in: V. Diekert, G. Rozenberg (Eds.), The Book of Traces, World Scientific, Singapore, pp. 3-41, 1995

[16] P. Cartier and D. Foata: commutation and rearrangements, Lectures notes in Mathematics, Berlin, Springer-Verlag, 1969

[17] V. Diekert and Y. Mtivier: Partial Commutation and Traces,Handbook of formal languages, vol. 3: beyond words, pp. 457–533, Springer-Verlag, New York, 1997

[18] G.J. Holzmann. The model checker SPIN. Software Engineering, 23(5):279295, 1997.

[19] D. Bosnacki and G.J. Holzmann. Improving spins partial-order reduction for breadth-first search. In SPIN, pages 91105, 2005.