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 constrains the system to some interesting configurations so that to reduce 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 breadth 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 explode. 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 cancel 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 $\Sigma^*$ the free monoid containing all strings with the empty word $\varepsilon$. Let $I \subseteq \Sigma^* \times \Sigma^*$ a symmetric and irreflexive relation called the independence relation inducing a binary relation $\sim$ between pairwised events of $\Sigma^*$ such that two words are equivalent: $u \sim v$ if and only if there exist $x, y \in \Sigma^*$ and $(a, b) \in \Sigma^2$ such that $u = xaby$ and $v = xbag$. $M(\Sigma^*, I)$ 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 neighboring 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 corresponds 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 $S = (\Sigma, s_0, T, S_{\text{ig}_1}, S_{\text{ig}_o}, S_v)$ where:

- $\Sigma$ is a set of states, and $s_0 \in \Sigma$ is the initial state;
- $S_{\text{ig}_1}$ and $S_{\text{ig}_o}$ are respectively the sets of input and output events; we suppose $S_{\text{ig}_1} \cup S_{\text{ig}_o} = \emptyset$;
- $T \subseteq \Sigma \times S_{\text{ig}_1} \times S_{\text{ig}_o} \times \Sigma$ is the set of transitions; a transition is a tuple $(s_1, a, \sigma, s_2)$ 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 \(\sigma\)
is the ordered sequence of output events produced when
the transition is fired. \((\text{Sig}_o)\) is the set of finite words of
output events;

- \(\text{Sig}_i : \Sigma \rightarrow 2^{\text{Sig}_o}\) 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 \(e \notin \text{Sig}_i(s)\), \(e\) is lost (i.e., it is not
memorized in the input buffer). However, if \(e \in \text{Sig}_i(s)\), it is
memorized in the current buffer and it will be consumed (or
saved again) in the next reached state.

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 \(\text{down}\)
and \(\text{up}\)), and the ignition and extinction of the correspond-
ing traffic lights (events \(\text{light}_{\text{on}}\) and \(\text{light}_{\text{off}}\)). Following
the previous notations, this system is defined by \(S = (\Sigma, \text{idle}, T, \text{Sig}_i, \text{Sig}_o, \text{Sig}_v)\) with

- \(\Sigma = \{\text{idle}, \text{busy}_1, \text{busy}_2, \text{busy}_12, \text{safe}, \text{tempo}\}\);
- \(T\) is the set of 14 transitions represented figure 1;
- \(\text{Sig}_i\) = \{\(e_1, s_1, e_2, s_2, \text{close}, \text{open}, \text{tick}\}\) 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 \(i = 1, 2\); \(\text{close}\) is a closure order sent by a human
operator (for emergency closure for example); \(\text{open}\) is a
manual opening order sent after a \(\text{close}\) order;
- \(\text{Sig}_o\) = \{\(\text{down}, \text{up}, \text{light}_{\text{on}}, \text{light}_{\text{off}}\}\);
- \(\text{Sig}_v\) is the save function (memorization) associating
the empty set with \(\text{idle}\) and \(\text{tempo}\) states, associating
\{\(\text{close}\)\} with \(\text{busy}_1\), \(\text{busy}_2\), and \(\text{busy}_12\), and associating
\{\(e_1, e_2, s_1, s_2\)\} with \(\text{safe}\) (graphically, \(\text{Sig}_v\) is defined by
a beveled parallelogram associated; the absence of paral-
lelogram implies \(\text{Sig}_v(s) = \emptyset\)).

For instance, the first transition from \(\text{idle}\) to \(\text{busy}_1\) is guarded
by the \(e_1\) and produces the sequence \(\text{down}, \text{light}_{\text{on}}\). Notice
that if \(s_1\) is received when the system is in \(\text{idle}\), \(s_1\) is lost
(it is not saved). However, if \(e_1\) is received in \(\text{safe}\), then it
is memorized until the arrival in \(\text{tempo}\) (after reception of
\(\text{open}\)) and then consumed (in \(\text{tempo}\)). At last, to illustrate
the technical point developed in this article, let us consider
the scenarios \(e_1 \in \text{Sig}_i\) and \(e_2 \in \text{Sig}_i\) from state \(\text{idle}\). From an observational
point of view, those two scenarios have (almost) the same
impact in the system. They both lead to the state \(\text{busy}_12\)
and produce the same outputs (\(\text{down}\) and \(\text{light}_{\text{on}}\)) 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:

\[
C ::= M \mid C_1.C_2 \mid C_1 + C_2 \mid C_1||C_2 \mid M \\
M ::= 0 \mid a!; M \mid a?; M
\]

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
Sequence 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 (0), or sequence of emissions
and receptions.

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 \(C\) =

\[
(((T_1, T_1) || T_2 || T_2 || T_2 || L) || (L) || (L))\] with \(T_1 = e_1; s_1; 0, T_2 = e_2; s_2; 0, \text{clk} = \text{tick}; \text{tick}; 0 \text{ and } H = \text{close}; \text{open}; 0, B = \text{down}?, \text{up}?: 0 \text{ and } L = \text{light}_{\text{on}}?: \text{light}_{\text{off}}?: 0\text{. This context is composed in parallel by (a) two successive trains on rail 1 (T}_1 \text{ and T}_1\), (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 \text{light}_{\text{on}} and then \text{light}_{\text{off}} (the barrier is initially
of the system. Notice that this context is restricting the behaviour of the system allowing only one train simultaneously 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 written by the relation \((C, B) \xrightarrow{a} (\gamma(C), B')\) 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 \( \text{null}_a \) if \( C \) doesn’t evolve) and then becomes the new context \( \gamma(C) \) 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 sequentially 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 consuming asynchronous events. The global behaviour of a system \( S \) and a context \( C \) is written by the relation
\[
< (C, B) | (s, S, B_2) > \xrightarrow{a} < (C', B'_1) | (s', S, B'_2) >
\]
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 \( S \) 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 \( \sigma \) to the context, and arrives in a another internal state \( s' \). The sequence \( \sigma \) 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 \( S \) can react to the event stored in the head of its input buffer, the corresponding transition is fired, and the sequence \( \sigma \) produced by this transition is sent to the context. \( S \) then reaches the destination state of the transition, and \( \sigma \) 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 \( S \).
- 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 \([C\mid S]\) the set of runs (i.e., traces) of the system \( S \) closed by the context \( C \) from the initial state of \( S \).

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 \([C\mid S]\) are also finite. In the following, we consider that all runs are ended by the terminal event \( \text{end}_C \). 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 \([C\mid S]\)) 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 (\( P_2 \)), 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).

From a formal point of view, an observer is a SDL automaton \( O = (\Sigma_0, \text{init}_O, T_O, \text{Sig}, \{\text{reject}\}, S_v_0) \) which can produce a single output event called reject. One says that \( S \) closed by \( C \) satisfies \( O \), denoted \([C\mid S] \models O\), if and only if there is no run from \([C\mid S]\) which, when composed with \( O \), 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 example, 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 \( O_1 \) and \( O_2 \). 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 whole 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 \( S = (\Sigma, init, T, Sig_i, Sig_o, S_0) \), an observer \( O = (\Sigma_o, init_o, T_o, Sig_o, \{reject\}, S_{0_o}) \), and a context \( C \). We denote by \( tgt_S(a_1,a_2, s) \) (respectively \( tgt_O(a_1,a_2, s) \)) the set of \( S \) states (respectively \( S \) states) reached after executing the scenario \( a_1a_2 \) from state \( s \). We also denote by \( |w| \) the length of a word \( w \) and by \( |w|_a \) the occurrence of \( a \) in the word \( w \). The first independence (with respect to the system) is denoted \( I_S \). The second one (with respect to the observer) is denoted by \( I_O \). The last one (with respect to the context) is denoted \( I_C \).

1) System independence: \((I_S)\): 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 \( \sigma_1 \) the scenario \( e_1,e_2 \) and by \( \sigma_2 \) the scenario \( e_2,e_1 \):

<table>
<thead>
<tr>
<th>state</th>
<th>( tgt_S(\sigma_1, state) )</th>
<th>( tgt_S(\sigma_2, state) )</th>
<th>( I_S(e_1, e_2, s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>{busy_12}</td>
<td>{busy_12}</td>
<td>true</td>
</tr>
<tr>
<td>busy_12</td>
<td>{busy_12}</td>
<td>{busy_12}</td>
<td>true</td>
</tr>
<tr>
<td>tempo</td>
<td>{busy_12}</td>
<td>{busy_12}</td>
<td>true</td>
</tr>
<tr>
<td>busy_1</td>
<td>{busy_12}</td>
<td>{busy_12}</td>
<td>true</td>
</tr>
<tr>
<td>busy_2</td>
<td>{busy_12}</td>
<td>{busy_12}</td>
<td>true</td>
</tr>
<tr>
<td>safe</td>
<td>{safe}</td>
<td>{safe}</td>
<td>true</td>
</tr>
</tbody>
</table>

In all states \( s \), \( tgt_S(\sigma_1, s) = tgt_S(\sigma_2, s) \). \( e_1 \) et \( e_2 \) are then independent with respect to \( S \).

Let us now consider events \( e_1 \) and \( close \). There exists a state \((idle)\) such that the set of reached states is different: \( tgt_S(e_1, close, idle) = busy_1 \) and \( tgt_S(close, e_1, idle) = safe \). They are then dependent (they do not commute).

At the end, we obtain the following set of pairwise independent events \( I_S = \{(e_1,e_2), (e_1,open), (e_2,open), (e_1,s2), (e_2,s1), (s1,s2), (s1,open), (s2,open)\} \).

The method computing \( I_S \) is given by algorithm 1.

2) Observer independence: \((I_O)\): 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 \( I_S \).

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 whether \( down \) 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 $S$. Let us consider a state $s$ from this system; let us also consider the traces obtained when simulating $S$ in $s$ with $a_1 \cdot a_2$ and then with $a_2 \cdot a_1$. By definition, these traces are: $\text{out}_S(a_1 \cdot a_2, s)$ and $\text{out}_S(a_2 \cdot a_1, s)$.

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 is 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 satisfy the observer.

For instance the events $e_1$ and $e_2$ from the system 1 are $I_S \cdot O_2$. The scenarios generated by $e_1 \cdot e_2$ and $e_2 \cdot e_1$, if they are restricted to observed events by $O_2$, are equal to the single event down. It is straightforward that these observers produce the same behaviour on $O_2$ (and thus the same state).

Let us consider $\sigma_1 = \text{close.open}$ and $\sigma_2 = \text{open.close}$. In the picture 3(a), $\text{tgt}_0(\sigma_1, s_0) = \{s_0\}$ and $\text{tgt}_0(\sigma_2, s_0) = \{s_1\}$. As a consequence we obtain: $I_0(\text{close.open}) = \text{FALSE}$. 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 satisfy the observer.

For instance the events $e_1$ and $e_2$ from the system 1 are $I_S \cdot O_2$. The scenarios generated by $e_1 \cdot e_2$ and $e_2 \cdot e_1$, if they are restricted to observed events by $O_2$, are equal to the single event down. It is straightforward that these observers produce the same behaviour on $O_2$ (and thus the same state).

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

3) Context independence ($I_C$): Let us consider two input events $a_1$ and $a_2$ from $S$, and a state $s$ of $S$. Let us consider the set of output sequences produced by $S$ in response to $a_1 \cdot a_2$ and to $a_2 \cdot a_1$ respectively. Let us denote $\text{out}_S(a_1 \cdot a_2, s)$ and $\text{out}_S(a_2 \cdot a_1, s)$ 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 $e_1 e_2$ and $e_2 e_1$ produce the sequences $\text{down} \cdot \text{light on}$ and $\text{light on} \cdot \text{down}$ 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 $(e_2 e_1)$ is compliant with this new context. The other leads to a context deadlock. $e_1$ and $e_2$ are then dependent with respect to $C'$.

Finally, let $a_1$ and $a_2$ two input events from $S$, 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: $I_{glob} \overset{\text{def}}{=} I_S \cap I_O \cap I_C$

Algorithm 1: System Independence algorithm

\begin{verbatim}
I_S(a, b, S):
    foreach s ∈ S do
        if ¬(tgt(a, b, s) = tgt(b, a, s)) then
            return false;
    end
    return true;

tgt(a, b, s):
    σ₁ ← ø; σ₂ ← ø;
    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
\end{verbatim}

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

\[ I_o(a, b, S, O) : \]
\[ r_1 \leftarrow \emptyset; r_2 \leftarrow \emptyset; \]
\[ \text{foreach state } s \in S, \Sigma \text{ do} \]
\[ r_1 \leftarrow outS(a, b, s); r_2 \leftarrow outS(b, a, s); \]
\[ \text{if } \neg \left( \text{indep}_o(r_1, r_2, s, O) \land \text{indep}_o(r_2, r_1, s, O) \right) \]
\[ \text{then} \]
\[ \quad \text{return false;} \]
\[ \text{end} \]
\[ \text{return true;} \]
\[ \]
\[ \text{indep}_o(r_1, r_2, s, O) : \]
\[ \text{foreach trace } t_1 \in r_1 \text{ do} \]
\[ \quad \text{foreach trace } t_2 \in r_2 \text{ do} \]
\[ \quad \quad \text{foreach state } s_o \in O \text{ do} \]
\[ \quad \quad \quad \text{if } (tgt_o(r_1, s_o) = tgt_o(r_2, s_o)) \]
\[ \quad \quad \quad \text{then} \]
\[ \quad \quad \quad \quad \text{return true;} \]
\[ \quad \quad \text{end} \]
\[ \quad \text{end} \]
\[ \text{return false;} \]

Algorithm 3: Context Independence algorithm

\[ I_c(a, b, S, C) : \]
\[ r_1 \leftarrow \emptyset; r_2 \leftarrow \emptyset; \]
\[ \text{foreach state } s \in S, \Sigma \text{ do} \]
\[ r_1 \leftarrow outS(a, b, s); r_2 \leftarrow outS(b, a, s); \]
\[ \text{if } \neg \left( \text{indep}_c(r_1, r_2, s, C) \land \text{indep}_c(r_2, r_1, s, C) \right) \]
\[ \text{then} \]
\[ \quad \text{return false;} \]
\[ \text{end} \]
\[ \text{return true;} \]
\[ \]
\[ \text{indep}_c(r_1, r_2, s, C) : \]
\[ \text{foreach trace } t_1 \in r_1 \text{ do} \]
\[ \quad \text{foreach trace } t_2 \in r_2 \text{ do} \]
\[ \quad \quad \text{if } (\text{equiv}(t_1, t_2)) \]
\[ \quad \quad \text{then} \]
\[ \quad \quad \quad \text{return true;} \]
\[ \quad \quad \text{end} \]
\[ \quad \text{end} \]
\[ \text{return false;} \]
\[ \]
\[ \text{equiv}(t_1, t_2) : \]
\[ \text{if } \neg (|t_1| = |t_2|) \]
\[ \quad \text{then} \]
\[ \quad \quad \text{return false;} \]
\[ \quad \text{else} \]
\[ \quad \quad \text{foreach letter } a \in t_1 \text{ do} \]
\[ \quad \quad \quad \text{if } \neg (|t_1|_a = |t_2|_a) \]
\[ \quad \quad \quad \quad \text{then} \]
\[ \quad \quad \quad \quad \quad \text{return false;} \]
\[ \quad \quad \quad \quad \text{else} \]
\[ \quad \quad \quad \quad \quad \text{foreach } (a, b) \in t_1 \times t_2 \text{ do} \]
\[ \quad \quad \quad \quad \quad \quad \text{if } ((a, b) \in C) \]
\[ \quad \quad \quad \quad \quad \quad \quad \text{then} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \text{return false;} \]
\[ \quad \quad \quad \quad \quad \quad \quad \text{end} \]
\[ \quad \quad \quad \quad \quad \text{end} \]
\[ \quad \text{end} \]
\[ \text{end} \]
\[ \text{return true;} \]

a) Application to the scenarios reduction: Let \( S \) a system SDL, \( C \) its context and \( O \) 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 \([w]_o\) of the monoid \( M(Sig_i, I_{glob}) \) 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 \( O \).

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

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 \( \Sigma^* \) 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) \( x = x_1 \ldots x_n \),
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 popped from the stack. A marker is popped in every stack corresponding to a letter which is dependent with a letter of the selected set. The procedure is depicted in algorithm 4.

d) example:

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.

<table>
<thead>
<tr>
<th>System</th>
<th>Process</th>
<th>Actors</th>
<th>Scn Initial</th>
<th>Scn reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAIN</td>
<td>1</td>
<td>4</td>
<td>37800</td>
<td>3300</td>
</tr>
<tr>
<td>CPDLC</td>
<td>1</td>
<td>3</td>
<td>840</td>
<td>2</td>
</tr>
<tr>
<td>AFN</td>
<td>3</td>
<td>4</td>
<td>1238</td>
<td>301</td>
</tr>
</tbody>
</table>

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 corresponding to the signal $start2$. 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 as 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 $waitAFN$ 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 response $sv2Cf$ as a confirmation. At this moment,
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 screens plays where the pilot and copilot could type and receive 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 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 were 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. the reason of this huge reduction comes from the system where in almost every state all the messages are saved 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 a airborne 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 compone 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.

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 notify 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 approximately 50 states each when expanded with the save clauses. The systems is modelled by the engineer and thus the complexity of the system is mastered 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 waiting 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 induced 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 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 determined in order to obtain a new independence relation based on temporal scopes equivalence.

REFERENCES


\^Atis Facility Notification