Application of partial-order methods for the verification of closed-loop SDL systems

Xavier Dumas
CS-SI, Toulouse, France
xavier.dumas@c-s.fr

Frédéric Boniol
ONERA, Toulouse, France
frederic.boniol@onera.fr

Philippe Dhaussy
ENSIETA, Toulouse, France
dhaussy@ensieta.fr

Eric Bonnafous
CS-SI, Toulouse, France
eric.bonnafous@c-s.fr

ABSTRACT

This article is concerned with the verification of closed-loop asynchronous reactive systems. Such systems, specified for instance with the industrial SDL (Specification and Description Language) language, communicate with their environment through buffers which memorize occurrences of events. Such a communication mechanism is quite interesting for specifying systems connected to several asynchronous external actors. However, it leads to a verification model possibly composed of a huge number of states (due to the state-space of the buffers). This article shows how this combinatorial explosion could be reduced by specifying the environment of the system to be verified, and by using partial-orders methods both on the system and its environment.

After presenting the formal modeling languages SDL (for the reactive system) and CDL Context Description Language (for its environment), the main points of our work are two-fold: (1) we define an independence relation between input events for a given specification $< C, S, \varphi >$ where S is the specification of the system (in SDL), C is the behavior of its external environment (in CDL), and φ the property to verify. The key point is that this independence relation is separately computed on S, C and φ, without building the global synchronization product of the system; (2) we apply the Mazurkiewicz theory for defining the set of scenarios (sequences of input events) which exactly covers the environment C and which is sufficient for verifying φ on S. We finally show on two industrial case-studies that this approach leads to an interesting reduction in verification time.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software/Program Verification, Testing and Debugging

General Terms

Verification

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

Keywords

Partial-order methods, asynchronous systems verification

1. INTRODUCTION

Reactive systems are becoming extremely complex with the soaring of high technologies. Despite of technical improvements, the increasing size of the systems makes the introduction of a wide range of potential errors easier. Among reactive systems, the asynchronous systems communicating by exchanging some asynchronous messages via buffer queues are often characterized by a huge number of possible behaviors.

To cope with this difficulty, manufacturers of industrial systems make significant efforts in testing and simulation to succeed the certification process. Nevertheless finding errors and bugs in this huge number of behaviors remains a very difficult activity.

An alternative method is to adopt formal methods, and to use exhaustive and automatic verification tools.

1.1 The addressed problem: combinatorial explosion

Several model-checker have been developped to help the verification of concurrent asynchronous systems. Especially, the SPIN model-checker [**?**] based on the formal language PROMELA. This tool allows the verification of LTL properties encoded in "never claim" formalism and further converted into Büchi automata. 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 [**?**]. In [**?**] 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.

Methods using symbolic model-checking based on BDD (Binary Decision Diagram [**?**]) have encountered a big success. Nevertheless the drawback of this method is pointed out with the necessity to transform the application into a boolean circuit possibly leading to irretrievable number of variables.

Combined together, all these methods made a lot of progress, and allow to push the limits of the combinatorial explosion. However, their efficiency often remains insufficient in case of real industrial systems.

Other techniques emerged in the 90's taking advantage

of the structure of the system rather than the global state space. Among them, partial-order methods [?, ?] aim at eliminating equivalent sequences of transitions in the global state space without modifying the falsity of the property under verification. These methods, exploiting the symmetries of the systems, seemed to be interesting and were integrated in a lot of verification tools (for instance SPIN). Nevertheless, as shown in section 4, they often remain unable to tackle with industrial systems (see for example the experiment with SPIN on the AFN system in section 4).

1.2 Verification by context modeling

In this paper we follow another approach called OBP (Observer Based Prover) proposed by [?] which consists to reduce the set of possible behaviors (and then indirectly the state space) by closing the system under verification with a well defined environment. Indeed, in the context of embedded reactive systems, the environment of each system is often well known, and it is more efficient to define this context than to search to reduce the state space of the system to explore. In other words, the objective is to circumvent the problem of the combinatorial explosion by restricting the system behavior with a specific surrounding environment describing the different configurations in which one wants to verify the system. Moreover, properties are often related to specific use cases (such as initialization, reconfiguration, degraded modes...) so that it is not necessary for a given property to take into account all possible behaviors of the environment, but only the subpart concerned by the verification. The context description thus allows a first limitation of the explored space search, and hence a first reduction of the combinatorial explosion.

The second idea exploited by [?] is that, if the context is finite (i.e., there is non infinite loop in the context), then the two following verification processes are equivalent: (a) compose the context and the system, and then verify the resulting global system; (b) unroll the context into N scenarios (i.e., a sequence of events), and successively compose each scenario with the system and verify the resulting composition. In other words, the authors transform the initial global verification problem into N smaller verification subproblems.

However, in realistic cases, the number N of unrolled scenarios may become very huge (see for instance the small rail-crossing example in section 4). In that sense, authors of [?] have just transformed the initial combinatorial explosion in space into a combinatorial explosion in time.

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 under verification in the context and in the properties.

1.3 Contribution of the paper

Despite its limits, the approach proposed in [?] seems to us interesting and well adapted to the verification activity in industrial context. On the other hand, asynchronous reactive system often contain symmetries (because the systems are often designed to be insensitive to the arrival order of certain events). We propose then in this article to improve the work of [?] by adapting the partial-order methods to their approach.

More precisely, we study a method to reduce the number of scenarios to be verified by exploiting the symmetries of the system and its context. The aim is to factorize a set of equivalent scenarios into a single one. To this purpose, we define an independence relation between events. Intuitively, two events are considered as independent if they can commute in the context without modifying the behavior of the system. We then propose a decision procedure relying on Mazurkiewicz traces and Foata Normal Form to decide wether two scenarios are equivalent (i.e., the first one can be obtained from the second by some permutations of successive independent events). We show then that the OBP method of [?] combined with our partial-order method is more efficient than SPIN on two examples.

In the following, we focus our approach on the language SDL for modeling the asynchronous systems to be verified, and on the language CDL proposed by [?] in OBP for modeling a context (i.e., a system environment).

In the sequel the document is built as follows. Section 2 gives a brief recall on the SDL language and presents the CDL language. This language being defined informally by [?], we propose in this section a formal operational semantics for CDL. Section 3 explores then an independence relation between events relying on their commutation. We deduce then an equivalence relation between scenarios based on this independence relation. Finally we present in section 4 some experimental results.

2. THE MODELING FRAMEWORK: SDL + CDL + OBSERVERS

As described above we are concerned in this article with the verification of reactive asynchronous systems closed by an environment. The model under verification is composed of three parts: (a) the system modeled in SDL [?], (b) its environment modeled in CDL [?] and (c) the properties to verify encoded as observers, i.e., state machines which observe the system and its environment, and which product a *reject* event whenever the verification fails. This section briefly presents this modeling framework.

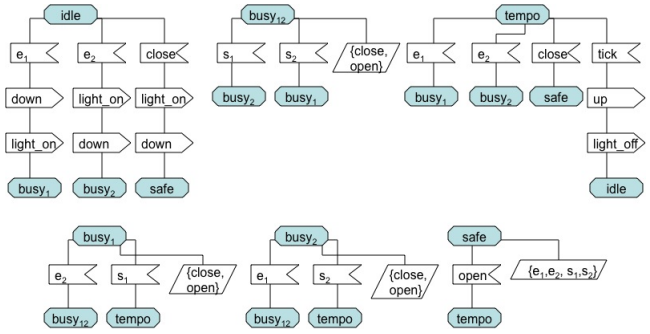
2.1 The SDL part

SDL (Specification Description Language) is a formal hierarchical language standardized by the ITU-T Z.100 firstly dedicated to the specification of protocol communication systems, and extended for modeling embedded systems. SDL is a graphical language. It is structured in four complementary views: (a) an architectural view, (b) a communication view, (c) a behavioral view, and (d) a data view. We only consider in the following the behavioral view.

The semantics of SDL relies on an extended state machine with asynchronous messages. Readers not familiar with SDL can refer to [?] for a detailed presentation. In this article, we only focus on a subset of SDL (mono process, without timers and assignments).

SDL models.

An SDL model is a transition system $\mathcal{S} = \langle \Sigma, s_o, T, Sig_i, Sig_o, Sv \rangle$ where: Σ is a set of finite states, and $s_o \in \Sigma$ the initial state of $\mathcal{S}$; Sig_i and Sig_o are respectively the set of input and output events of $\mathcal{S}$; we'll suppose Sig_i and Sig_o are disjoint; $T \subseteq \Sigma \times Sig_i \times Sig_o^* \times \Sigma$ is the set of transitions; a transition is a tuple (s_1, a, σ, s_2) where s_1 is the source of the transition, s_2 is the target state, a is the event from Sig_i



(a) Graphical model

init = *idle*
$\Sigma = \{idle, busy_1, busy_2, busy_{12}, safe, tempo\}$
$Sig_i = \{e_1, s_1, e_2, s_2, close, open, tick\}$
$Sig_o = \{down, up, light\_on, light\_off\}$
$Sv : \Sigma \rightsquigarrow 2^{Sig_i}$
- $idle,\ tempo \rightarrow \emptyset$
- $busy_1,\ busy_2,\ busy_{12} \rightarrow \{close,\ open\}$
- $safe \rightarrow \{e_1, e_2, s_1, s_2\}$

(b) Textual version

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

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

Notice that if e is received when the system is in state s and if e can't be consumed in the state s (i.e., there is no transition from s guarded by e), then if $e \notin Sv(s)$, e is lost (i.e., it is not memorized in the input buffer). In the other case, if $e \in Sv(s)$, it is memorized in the current buffer and tried to be consumed again in the next reached state.

Let's consider the example figure 1. This model specifies an SDL process which controls a rail-crossing system. According to the previous notations, this process is textually defined on the right part of the figure, whereas the left part gives its graphical version. This model specifies a process receiving events from Sig_i and producing events from Sig_o. e_i (resp. s_i, with $i = 1, 2$) is emitted by a train entering (resp. leaving) the level crossing via rail i. *close* is an emergency order sent by a human operator, while *open* is the corresponding manual order sent in order to open again the barriers. *down*, *up*, *light_on*, *light_off* are the output events produced by the system in order to control the barriers and the lights. For instance, the first transition, from *idle* to $busy_1$ is guarded by e_1, and produces the sequence *down.light_on* when it is fired. Notice that if s_1 is received when the system is in state *idle*, then s_1 is lost. Conversely, if e_1 is received in state *safe*, then it is memorized until *tempo* is reached (after receiving *open*) and then consumed.

At last, to illustrate the technical point developed in this article, let's consider the scenarios $e_1.e_2$ and $e_2.e_1$. From an observational point of view, those two scenarios have (almost) the same impact on the system. They both lead to the state $busy_{12}$ from *idle*, and they both produce the same outputs (*down* and *light_on*) but in a different order. If this order is irrelevant for the environment, then we'll consider those two scenarios as equivalent.

SDL Semantics.

An SDL model is an automaton communicating with its environment in an asynchronous way, by receiving input events in a buffer, and consuming these events in their arrival order. The simplest way to define the SDL semantics is to express it as an evolution relation between tuple (s, B) where s is the current state of the system and B its current input buffer (i.e., a finite ordered word of events from Sig_i). This semantics relation is defined inductively by the following rules. To simplify, we note $null_\sigma$ for the empty buffer (i.e., empty word). We also denote by $a \cdot B$ the buffer composed of the event a followed by buffer B. $B_1 \cdot B_2$ is the buffer composed by the concatenation of B_1 and B_2 (B_1 before B_2).

$$\frac{(s, a, \sigma, s') \in \mathcal{S}.T}{(s, \mathcal{S}, a \cdot B) \ \overrightarrow{\sigma}\ (s', \mathcal{S}, B)} \quad [\text{trans}]$$

$$\frac{\begin{array}{c} a \notin Sv(s) \\ \forall s', \forall \sigma, (s, a, \sigma, s') \notin \mathcal{S}.T \end{array}}{(s, \mathcal{S}, a \cdot B) \ \overrightarrow{null_\sigma}\ (s, \mathcal{S}, B)} \quad [\text{discard}_S]$$

$$\frac{\begin{array}{c} a \in Sv(s) \\ (s, \mathcal{S}, B) \ \overrightarrow{w}\ (s', \mathcal{S}, B') \\ \forall s', \forall \sigma, (s, a, \sigma, s') \notin \mathcal{S}.T \end{array}}{(s, \mathcal{S}, a \cdot B) \ \overrightarrow{w}\ (s', \mathcal{S}, a \cdot B')} \quad [\text{save}]$$

The *trans* rule expresses that if the event a is at the head of the current buffer, and if from s (the current state) there exists a transition guarded by a, then this transition is fired, the sequence σ of output events of the transition is produced, the destination state is reached, and the event a is consumed.

The rule *discard* expresses that if the event a is at the head of the current buffer, and if from s (the current state) there is no transition guarded by a, and if a is not specified to be memorized in s, then a is lost (i.e., removed from the buffer). The current state remains the same. No output events are produced.

Lastly, the *save* rule expresses that whenever a is at the head of the current buffer, if from s (the current state) there is no transition guarded by a, and if a is specified to be memorized in s, then a is memorized (it remains at the head of the buffer), but the system evolves (if possible) by exploiting the remaining events of the buffer.

In the sequel $(s, \mathcal{S}, B) \not\rightarrow$ expresses that no rule can be applied to $\mathcal{S}$ in the state s with the buffer B, meaning that $\mathcal{S}$ cannot evolve.

2.2 The CDL part

As explained in introduction, we are concerned with the verification of an SDL model closed by an environment. For that purpose, we consider in the following that such environment is modeled in CDL, a formal language based on MSCs (Message Sequence Charts) introduced in [**?**].

A CDL model (also called "context") is a finite generalized MSC C, following the formal grammar:

$$\begin{array}{lcl} C & ::= & M \mid C_1.C_2 \mid C_1 + C_2 \mid C_1 \| C_2 \\ M & ::= & \mathbf{0} \mid a!; M \mid a?; M \text{ with } a \neq null_\sigma \end{array}$$

In other words, a context is either (a) a single MSC M composed as a sequence of event emission $a!$ and event reception $a?$ terminated by the empty MSC ($\mathbf{0}$) which does nothing, or (b) a sequential composition of two contexts ($C_1.C_2$), or (c) a non deterministic choice between two contexts ($C_1 + C_2$), or (d) a parallel composition between two contexts ($C_1 \| C_2$).

For instance, let us consider the context figure 2 graphically described on the left part, and formalized with the above textual grammar on the right part. This context describes the environment we want to consider for the verification of the rail-crossing system. This context is composed of several actors running in parallel or in sequence. Two actors represent the barrier (B) and the light (L) waiting for orders from the system. Note that each actor waits for exactly one instance of each order. *down* (resp. *light_on*) is supposed to be received before *up* (resp. *light_off*). However, the reception order between *down* and *light_on* is not constrained. They can be received in parallel in any order. H is a (human) actor which sends *close* and *open* to the system (in that order, but at any time). *clk* is a clock which produces two ticks, and one tick at the end of the context. And finally T_1 and T_2 represent trains on rail 1 and 2 respectively. Note that this context supposes that two trains go through the rail-crossing successively on rail 1, and only one on rail 2. All these actors run in parallel (except for the last *tick*), and interleave their behavior.

CDL Semantics.

Let's define a function $wait(C)$ associating the context C with the set of events waited in its initial state:

$$\begin{array}{lcl} wait(\mathbf{0}) & \stackrel{\text{def}}{=} & \emptyset \\ wait(a!; M) & \stackrel{\text{def}}{=} & \emptyset \\ wait(a?; M) & \stackrel{\text{def}}{=} & \{a\} \\ wait(C_1 + C_2) & \stackrel{\text{def}}{=} & wait(C_1) \cup wait(C_2) \\ wait(C_1.C_2) & \stackrel{\text{def}}{=} & wait(C_1) \\ wait(C_1 \| C_2) & \stackrel{\text{def}}{=} & wait(C_1) \cup wait(C_2) \end{array}$$

According to the SDL semantics, we consider that a context is a process communicating in an asynchronous way with the system, memorizing its input events (from the system) in a buffer. The semantics of CDL is defined by the relation $(C, B) \xrightarrow{a} (C', B')$ to express that the context C with the buffer B "makes" a (which can be a sending or a receiving signal, or the $null_\sigma$ signal if C doesn't evolve) and then becomes the new context C' with the new buffer B'. This relation is defined by the 8 rules figure 3:

- The *pref1* rule (without any preconditions) specifies that a MSC beginning with a sending event $a!$ emits this event and continues with the remaining MSC.
- The *pref2* rule expresses that if a MSC begins by a reception $a?$ and is faced to an input buffer containing this event at the head of the buffer, it consumes this event and continues with the remaining MSC.
- The *seq1* rule establishes that a sequence of contexts behaves like the first one until it has terminated. The *seq2* rule says that if the first context terminates (i.e., becomes $\mathbf{0}$), then the sequence becomes C_2.
- The *alt* rule expresses that the alternative context $C_1 + C_2$ behaves either like C_1, either like C_2.
- The *par1* and *par2* rules say that the semantics of the parallel operation is based on an asynchronous interleaving semantics.
- Finally, the *discard* rule says that if an event a at the head of the input buffer is not expected, then this event is lost (removed from the head of the buffer).

Note that this asynchronous semantics (corresponding to the true asynchronism between the system and its environment) can lead to an explosion of the number of scenarios. For instance, the CDL context depicted figure 2 contains 18,900 different scenarios.

2.3 Composition context + system

We can now formally define the "closure" composition $< (C, B_1) \mid (s, \mathcal{S}, B_2) >$ of a system $\mathcal{S}$ in a state s with an input buffer B_2, with its context C with an input buffer B_1 (note that each component, system and context, has its own buffer). The semantics of this composition is defined by the three following rules:

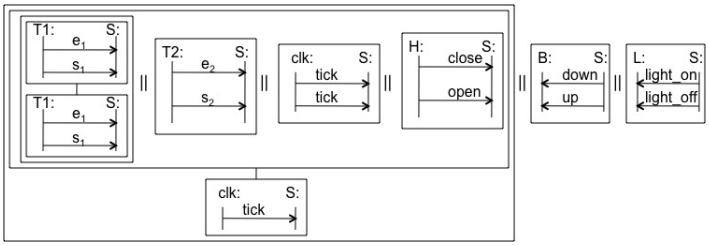
$$\frac{(s, \mathcal{S}, B_2) \stackrel{\rightarrow}{\sigma} (s', \mathcal{S}, B_2')}{< (C, B_1) | (s, \mathcal{S}, B_2) > \xrightarrow[\sigma]{null_e} < (C, B_1.\sigma) | (s', \mathcal{S}, B_2') >} \quad [\text{cp1}]$$

$$\frac{(C, B_1) \xrightarrow{a!} (C', B_1')}{< (C, B_1) | (s, \mathcal{S}, B_2) > \xrightarrow[null_\sigma]{a} < (C', B_1') | (s, \mathcal{S}, B_2.a) >} \quad [\text{cp2}]$$

$$\frac{(C, B_1) \xrightarrow{a?} (C', B_1')}{< (C, B_1) | (s, \mathcal{S}, B_2) > \xrightarrow[null_\sigma]{null_e} < (C', B_1') | (s, \mathcal{S}, B_2) >} \quad [\text{cp3}]$$

- rule *cp1*: if $\mathcal{S}$ can produce σ, then $\mathcal{S}$ evolves and σ is put at the end of the buffer of C.
- rule *cp2*: if C can emit a, C evolves and a is queued in the buffer of $\mathcal{S}$.
- rule *cp3*: at last, if C can consume a, then it evolves whereas $\mathcal{S}$ remains the same.

Note that the "closure" composition between a system and its context can be assimilated to an asynchronous parallel composition: the behavior of C and of $\mathcal{S}$ are interleaved, and they communicate through asynchronous buffers.



(a) Graphical version

$$
\begin{array}{lcl}
C & = & (((T_1.T_1)\|T_2\|clk\|H).(tick!.\mathbf{0}))\|B\|L \\
T_1 & = & e_1!; s_1!; \mathbf{0} \\
T_2 & = & e_2!; s_2!; \mathbf{0} \\
clk & = & tick!; tick!; \mathbf{0} \\
H & = & close; open; \mathbf{0} \\
B & = & down?; up?; \mathbf{0} \\
L & = & light\_on?; light\_off?; \mathbf{0}
\end{array}
$$

(b) Textual version

Figure 2: Example of a CDL environment: control of a rail-crossing

$$\frac{}{(a!; M, B) \xrightarrow{a!} (M, B)}\ [\text{pref1}] \qquad \frac{}{(a?; M, a.B) \xrightarrow{a?} (M, B)}\ [\text{pref2}] \qquad \frac{C_1' \neq \mathbf{0} \quad (C_1, B) \xrightarrow{a} (C_1', B')}{(C_1.C_2, B) \xrightarrow{a} (C_1'.C_2, B')}\ [\text{seq1}]$$

$$\frac{(C_1, B) \xrightarrow{a} (\mathbf{0}, B')}{(C_1.C_2, B) \xrightarrow{a} (C_2, B')}\ [\text{seq2}] \qquad \frac{(C_1, B) \xrightarrow{a} (C_1', B')}{\begin{array}{l}(C_1 + C_2, B) \xrightarrow{a} (C_1', B') \\ (C_2 + C_1, B) \xrightarrow{a} (C_1', B')\end{array}}\ [\text{alt}] \qquad \frac{C_1' \neq \mathbf{0} \quad (C_1, B) \xrightarrow{a} (C_1', B')}{\begin{array}{l}(C_1\|C_2, B) \xrightarrow{a} (C_1'\|C_2, B') \\ (C_2\|C_1, B) \xrightarrow{a} (C_2\|C_1', B')\end{array}}\ [\text{par1}]$$

$$\frac{(C_1, B) \xrightarrow{a} (\mathbf{0}, B')}{\begin{array}{l}(C_1\|C_2, B) \xrightarrow{a} (C_2, B') \\ (C_2\|C_1, B) \xrightarrow{a} (C_2, B')\end{array}}\ [\text{par2}] \qquad \frac{a \notin wait(C)}{(C, a.B) \xrightarrow{null_\sigma} (C, B)}\ [\text{discard}_C]$$

Figure 3: Context semantics

We will denote $< (C, B)|(s, \mathcal{S}, B') > \not\rightarrow$ to express that the system and its context cannot evolve (the system is blocked or the context terminated).

We then define the set of traces (called *runs*) of the system closed by its context from a state s, by:

$$\begin{array}{l}[\![C|(s, \mathcal{S})]\!] \stackrel{\text{def}}{=} \{a_1 \cdot \sigma_1 \cdot \ldots a_n \cdot \sigma_n \cdot end_C \mid \\ \quad < (C, null_\sigma)|(s, \mathcal{S}, null_\sigma) > \xrightarrow[\sigma_1]{a_1} < (C_1, B_1)|(s_1, \mathcal{S}, B_1') > \\ \quad \xrightarrow[\sigma_2]{a_2} \ \ldots \ \xrightarrow[\sigma_n]{a_n} < (C_n, B_n)|(s_n, \mathcal{S}, B_n') > \not\rightarrow \ \}\end{array}$$

$[\![C|(s, \mathcal{S})]\!]$ is the set runs of $\mathcal{S}$ closed by C from the state s. Note that a context is built as sequential or parallel compositions of finite loop free MSCs. Consequently the *runs* of an SDL model closed by a CDL context are necessarily finite. We then extend each *run* of $[\![C|(s, \mathcal{S})]\!]$ by a specific terminal event end_C allowing the observer to catch the ending of a scenario and "testing" possible liveness properties.

2.4 The observers part

The third part of the formalization relies on the expression of the properties to be fullfiled. For the sake of simplicity, we consider in this article that properties are modeled as observers. An observer is an automaton (described in SDL in order to simplify the notations) which *observes* the set of events exchanged by the system $\mathcal{S}$ and its context C (and thus events occuring in the *runs* of $[\![C|(init, \mathcal{S})]\!]$) and which produces an event *reject* whenever the property becomes false.

For example let us consider again the rail-crossing figure 1 and its verification context figure 2 . Let us consider the two following properties P_1 and P_2: (a) P_1: after the *close* order (sent by the context), no opening order *up* (decided by the system) can occur before the context sends *open*; (b) P_2: at the end of the context, barriers are opened. Those two properties can be modeled by the two observers depicted figures 4(a) and 4(b).

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

Figure 4: Observers' examples

We consider in the following that an observer is an SDL automaton $\mathcal{O} = \langle \Sigma_o, init_o, T_o, Sig, \{reject\}, Sv_o \rangle$ (a) emiting a single output event: *reject*, (b) where Sig is the set of catched events by the observer; events produced and received by the system and its context and (c) such that all transitions labelled *reject* arrive in a specific state called "unhappy".

Semantics.

We say that $\mathcal{S}$ in the state $s \in \mathcal{S}.\Sigma$ closed by C satisfies $\mathcal{O}$, denoted $C|(s, \mathcal{S}) \models \mathcal{O}$, if and only if no execution of $\mathcal{O}$ faced to the runs r of $[\![C|(s, \mathcal{S})]\!]$ produces a *reject* event. This means:

$C|(s,\mathcal{S}) \models \mathcal{O} \iff \forall r \in [\![C|(s,\mathcal{S})]\!], (init_o, \mathcal{O}, r) \xrightarrow{null_\sigma}$
$(s_1, \mathcal{O}, r_1) \xrightarrow{null_\sigma} \dots \xrightarrow{null_\sigma} (s_n, \mathcal{O}, r_n) \not\rightarrow$

Remark: executing $\mathcal{O}$ on a run r of $[\![C|(s,\mathcal{S})]\!]$ is equivalent to put r in the input buffer of $\mathcal{O}$ and to execute $\mathcal{O}$ with this buffer. This property is satisfied if and only if only the empty event ($null_\sigma$) is produced (i.e., the *reject* event is never emitted).

3. AN INDEPENDENCE RELATION

3.1 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 [**?**], and further described in section 2.2, consists to circumvent this explosion by closing the system under verification with a restricted operational context. Even though this method leads to reductions of the global state space, it is still not practical for huge systems with large contexts. Hence a complementary approach is necessary.

As described in the introduction, the aim of this article is to study a new reduction method of the global state space relying on the exploitation of symmetries of the model and its context. Those symmetries result from an independence relation between events emitted from the context to the system. Let's consider again the system depicted figure 1 together with its context C (figure 2 and the two properties figures 4(a) and 4(b). This system presents some symmetries. For instance let us consider the two sequences $e_1.e_2$ and $e_2.e_1$. In every state of the system, the execution of those two events leads to the same states. For instance, from *idle*, the two scenarios lead both to $busy_{12}$. However they produce the same output events (*down* and *light_on*) but in different order. Nevertheless these events are consumed by the environment in two MSCs in parallel. That means that their emission order is not relevant for the context. Moreover *light_on* is not relevant as well for the observers $\mathcal{O}_1$ and $\mathcal{O}_2$. Hence the two output sequences *down.light_on* and *light_on.down* may be considered as equivalent with respect to the context and observers. As a consequence we can say that e_1 and e_2 commute (we say that they are independent), meaning their occurrence order is irrelevant for the verification.

Note that to prove that e_1 and e_2 commute we did not need to explore the global state automaton obtained by the composition of the system and its context. We only require to separately analyse the SDL model, the CDL context and the observers. These models being designed by human engineers, one can expect this exploration does not suffer from the combinatorial explosion. This is the main advantage of the approach proposed by this article.

3.2 The independence relation

We'll say that two events a_1 and a_2 produced by the context are independent if and only if: (a) they are independent for the SDL system, meaning that they satisfy the "diamond property" (produced ouput events and states reached after $a_1.a_2$ and after $a_2.a_1$ remain the same), (b) they are independent with the produced output events for the specified observer, and (c) these events and the produced output events are context independent. The first independence will be denoted $S\_Indep_{\mathcal{S}}$ (the structural independence). The second one will be denoted by $O\_Indep_{\mathcal{S},\mathcal{O}}$ handling the independence on the system runs and on the observers. At last, the third one will be denoted $C\_Indep_{\mathcal{S},C}$ handling the independence on system runs and on the context model.

In the sequel we'll 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.

3.2.1 Preliminary

The independence relation between two events a_1 and a_2 from Sig_i relies on the diamond property on the SDL model and on the causality relation between context events. We now introduce the two following functions:

- $tgt\_states_{\mathcal{S}} : Sig_i^* \times \Sigma \rightsquigarrow 2^{(\Sigma \times Sig_i^*)}$. $tgt\_states_{\mathcal{S}}(\sigma, s)$ is a function which returns the set of pairwised (state,buffer) of the SDL automaton $\mathcal{S}$ reached from a state s after receiving the input sequence σ, defined by:
  $tgt\_states_{\mathcal{S}}(\sigma, s) \stackrel{\text{def}}{=} \{(s', \sigma') \mid (s, \mathcal{S}, \sigma) \xrightarrow[r_1]{} (s_1, \mathcal{S}, \sigma_1) \xrightarrow[r_2]{} \dots \xrightarrow[r_n]{} (s', \mathcal{S}, \sigma') \not\rightarrow \}$.
  For instance, if we focus our attention on the picture 1, $tgt\_states_{\mathcal{S}}(e_1 \cdot e_2, s_0) = \{(s_2, null_\sigma)\}$.

- $out_{\mathcal{S}} : Sig_i^* \times \Sigma \rightsquigarrow 2^{(Sig_i \cup Sig_o)^*}$. $out_{\mathcal{S}}(\sigma, s)$ is a function which returns the set of output events, interleaved with the input events flow σ, produced by the SDL automaton $\mathcal{S}$ from the state s in response to σ. $out_{\mathcal{S}}(\sigma, s)$ is defined by:
  $$out_{\mathcal{S}}(a_1 \cdot \dots \cdot a_n, s) \stackrel{\text{def}}{=} [\![(a_1! \cdot \dots \cdot a_n!; \mathbf{0})|(s, \mathcal{S})]\!]$$
  $out_{\mathcal{S}}(a_1 \cdot \dots \cdot a_n, s)$ is the response of $\mathcal{S}$ in the state s reacting to the context composed of a unique sequence $a_1, \dots, a_n$. For instance, in the system figure 1, $out_{\mathcal{S}}(e_1 \cdot e_2, s_0) = \{(e_1 \cdot down \cdot light\_on \cdot e_2), (e_1 \cdot e_2 \cdot down \cdot light\_on)\}$. The first trace is obtained by sending e_1 to the system, by waiting the reaction of the system, and then by sending e_2. The second trace is obtained by sending e_1 and e_2 in that order but quickly enough without letting time to the system to react between e_1 and e_2.

3.2.2 System independence: $S\_Indep_{\mathcal{S}}$

We say that two input events a_1 and a_2 from $\mathcal{S}$ are independent if and only if they commute, i.e., that from every state the sequences $a_1 \cdot a_2$ and $a_2 \cdot a_1$ lead to the same states. Formally, we define the independence relation $S\_Indep_{\mathcal{S}} \subseteq Sig_i \times Sig_i$ by:
$(a_1, a_2) \in S\_Indep_{\mathcal{S}} \Leftrightarrow \forall s \in \mathcal{S}.\Sigma, tgt\_states_{\mathcal{S}}(a_1 \cdot a_2, s) = tgt\_states_{\mathcal{S}}(a_2 \cdot a_1, s)$. For instance, e_1 and e_2 in the figure 1 are $S\_Indep_{\mathcal{S}}$. In every system state, the scenarios $e_1 \cdot e_2$ and $e_2 \cdot e_1$ lead to the same states. Nevertheless, *close* and e_1 are not $S\_Indep_{\mathcal{S}}$ since in *idle*, $e_1 \cdot close$ leads to $busy_1$ while $close \cdot e_1$ leads to *safe*.

3.2.3 Observer independence: $O\_Indep_{\mathcal{S},\mathcal{O}}$

Let a_1 and a_2 two input events from $\mathcal{S}$. Let us consider a state s from $\mathcal{S}$. Then let us consider the traces obtained when stimulating $\mathcal{S}$ in s with $a_1 \cdot a_2$ and then with $a_2 \cdot a_1$. a_1 and a_2 are independent with respect to $\mathcal{O}$ iff these two sets of traces produce the same internal behavior in $\mathcal{O}$. It's obvious that, if a_1 and a_2 are $\mathcal{O}$ independent, then either both $a_1 \cdot a_2$ and $a_2 \cdot a_1$ satisfy the observer, either none of

them satisfy it. For instance e_1 and e_2 are $O\_Indep_{\mathcal{S},\mathcal{O}_2}$. The scenarios generated by $e_1 \cdot e_2$ and $e_2 \cdot e_1$, restricted to events observed by $\mathcal{O}_2$, are equal to *down*.

3.2.4 Context independence: $C\_Indep_{\mathcal{S},C}$

Let two input events a_1 and a_2 from $\mathcal{S}$. Let us consider a state s from $\mathcal{S}$. Then let us consider the traces obtained when stimulating $\mathcal{S}$ in s with $a_1 \cdot a_2$ and then with $a_2 \cdot a_1$. We'll say that a_1 and a_2 are C independent if these 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 (a) which doesn't differ from r_1 only by events order, and (b) 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' and r_1, and inversely in r_2), then b and b' appear in parallel in the context (the context doesn't specify the order of b and b'). For instance, let $C' = light\_on?; down?; \mathbf{0}$. C' supposes that *light_on* must be received (by the context) before *down*. Let us consider the two events e_1 and e_2. The scenario $e_1 \cdot e_2$ produces the output sequence $down \cdot light\_on$, whereas the reverse scenario produces the reverse output sequence. However, C' waits for these two events in a particular order. Hence, e_1 and e_2 are dependent in C'.

3.2.5 Global independence

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

3.3 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.

Definition of the Mazurkiewicz traces.

The idea behind Mazurkiewicz traces is that relying on an independence relation, two traces could be considered as equivalent if the first one can be obtained from the second one by permutation of neighboring independent events [**?**]. Let $I \subseteq Sig \times Sig$ the independence relation between two events (two events in relation by I commute). $\mathbb{M}(Sig, I)$ is a free monoid partially commutative representing the set of traces quotiented by the least congruence $\equiv_I$ on Sig^* such that $ab \equiv_I ba\ \forall a, b \in I^2$. The traces of $\mathbb{M}(Sig, I)$ quotiented by $\equiv_I$ are called Mazurkiewicz traces. Let w a word from Sig^*, we denote by $[w]$ the Mazurkiewicz trace of w (for a given independence relation).

Foata Normal Form.

The question is then, from an independence relation, how to compute the different sets of $[w]$, without taking into account all possible permutations of neighboring events. For that purpose, we use the notion of Foata Normal Form [**?**]. The idea consists in rewriting a word until a unique irreductible word is obtained. The obtained word being unique, all the traces with the same normal form belong to the same Mazurkiewicz class. [**?**] describes a transformation procedure to generate the Foata Normal Form modulo an independence relation I.

The reduction method.

Let $\mathcal{S}$ an SDL system, C its context and $\mathcal{O}$ an observer to be verified. Let us consider the free monoid partially commutative in which the words are the scenarios of events produced by C and the independence relation is $Indep_{\mathcal{S},\mathcal{O},C}$. We apply the method described in [**?**] to compute the set of distinct scenarios (i.e., which have a different behavior with respect to the system or to the observer). Those scenarios are the equivalent classes $[w]$ of the monoid $\mathbb{M}(Sig_i, Indep_{\mathcal{S},\mathcal{O},C})$ where Sig_i is the set of input events of the system.

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

THEOREM 1. *Let $[w]$ a class from $\mathbb{M}(Sig_i, Indep_{\mathcal{S},\mathcal{O},C})$, $\forall C_{\sigma_1}, C_{\sigma_2} \in [w],\ \ C_{\sigma_1} \| (init, \mathcal{S}) \models \mathcal{O} \Leftrightarrow C_{\sigma_2} \| (init, \mathcal{S}) \models \mathcal{O}$ where C_{σ_1} and C_{σ_2} are sequential contexts respectively playing the scenarios σ_1 and σ_2.*

This result directly comes from the independence relation between events. By definition of the Mazurkiewicz traces, it is possible to go from σ_1 to σ_2 by successive permutations of independent events. However, by definition of the independence relation, the permutation between two independent events does not modify the behavior of the system neither of the observer. As a consequence such a permutation does not modify the truth or falsity of the observer. As a consequence, both σ_1 and σ_2 satisfy (or not) the observer.

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

4. EXPERIMENTAL RESULTS

In order to test the performance of our method, we have compared, on two different systems, the verification time obtained with SPIN 5.2.5, and the verification time obtained by the OBP toolset after being optimized with our partial-order method[1]. The results of these comparisons are given table 1. The two systems are: the rail-crossing control system, called Train in the table, and an industrial avionics system, called AFN (Atis Facility Notification).

Rail-crossing system.

We verified the two properties modeled by the observers $\mathcal{O}_1$ and $\mathcal{O}_2$. In Promela (the language of SPIN), these observers are translated into the linear temporal logic LTL by respectively the two formulas: $close \rightarrow (\neg up\ \mathcal{U}_w\ open)$ and $down \rightarrow (\neg end_C\ \mathcal{U}_w\ open)$ where $\mathcal{U}_w$ is the weak until operator. The verification time with SPIN (with its internal partial-order option) of this system with the context figure 2 takes 42.3 seconds for the first formula and 67.6 seconds for the second one. These performances could be explained by the fact the train system is rather small and is thus weel suited for classical *model-checking* method. Following our method, the verification process begins by computing the indepedence relation, and then builds the Mazurkiewicz classes of the context (in our example there are 3,444 classes among 18,900 scenarios). We then separately verify one scenario of each class (precisely the Foata Normal Form of each

[1]The experiments were run on a 32 bit Linux host (Intel Dual Core T6400 @2.00GHz) with 3.5GB of DDR2 memory.

System	Nb of scn after reduction	Nb of scn with SPIN	Verification time with OBP	Verification time + partial-order distributed on 10 computers
Train ($\mathcal{O}_1$)	18,900	3,444	42.3s	0.38s + 443s/10 = 44.68s
Train ($\mathcal{O}_2$)			67.6s	0.38s + 460s/10 = 46.38s
AFN	495	90	∞ (lack of memory)	1.10s + 65.10^3s/10 = 6,500s

Table 1: Experimental results

class) on the system (by using SPIN again). The first steps (independence relation and Mazurkiewicz classes identification) lasts 0.38 seconds. The second step (i.e., the verification of all the 3,444 classes) lasts 443 seconds for the first property and 460 seconds for the second one if the verification is made with only one computer. In this case, our method is less efficient of the classical *model-checking* because of the compilation time of each scenario with the system and the observer. Actually, SPIN doesn't allow separate compilation and the system and the observers have to be compiled 3,444 times instead of one time. As a comparison, with only one compilation of the system and the observer, the verification with our method with the first formulae would take 5.23 seconds and 4.28 seconds for the second one. Nevertheless, contrary to SPIN model-checker, our method allows parallel verification on several computers by distributing the scenarios fairly on each computer. As a consequence, our method could equal the SPIN performances on this case study if we would distribute the 3,444 scenarios on 10 computers as shown in the table 1.

Atis Facility Notification system.

The AFN system is an avionics function which manages the notifications and some data transmissions between the aircraft and the ATC centers. This system is composed of 3 SDL processes of about 50 states each. Its context is composed of 4 actors in parallel (other avionics systems). This model, translated in Promela, takes about 10,800 lines of code (in comparison, the Promela train model takes 380 lines of code). As shown in table 1, the verification with SPIN does not terminate. In contrast, our verification process takes about 18 hours because of the time compilation (with separate compilation, our method would take 10.52 seconds). Nevertheless if we distribute the verification on 10 computers as previously, the verification would terminate in about 1.8 hours (table 1).

5. CONCLUSION AND PERSPECTIVES

In this article we studied a partial-order method combined with the OBP methodology based on the context modeling. From the context of a system, this methodology builds all the scenarios of the context and verifies each of them on the system. We show that it is possible to reduce the number of these scenarios by defining an independence relation between events, and by eliminating all the redundant scenarios (i.e., scenarios which can be obtained from another one by permuting independent events). We show then on two case studies that this method should be very efficient, in comparison with other model-checkers such as SPIN.

The next work to do concerns data and timers in SDL and CDL models. In our opinion, data does not complexify our approach, since they may be abstracted by causal dependences between transitions in SDL models. However, timers are more difficult to take into account. A possible way could be to abstract timers by four events: *set* (beginning of the timer), *reset*, *abort* and *end*. However, such an abstraction may lead to an explosion of the number of possible scenarios. Another approach, for instance based on a timed semantics, should be likely necessary. Another work is to optimise the compilation time induced by the verification of each scenario to make the method even more efficient.

6. REFERENCES