

Context-aware Verification of a Cruise-Control System

Ciprian Teodorov, Luka Leroux, Philippe Dhaussy

UEB, Lab-STICC Laboratory UMR CNRS 6285

ENSTA Bretagne, France

`firstname.name@ensta-bretagne.fr`

Abstract. Despite the high-level of automation, the practicability of model-checking large asynchronous models is hindered by the state-space explosion problem. To address this challenge the Context-aware Verification technique relies on the identification and explicit specification of the environment (context) in which the system-under-study operates.

In this paper we apply this technique for the verification of a Cruise-Control System (CCS). The asynchrony of this system renders traditional model-checking approaches almost impossible. Using the Context-aware Verification technique this task becomes manageable by relying on two powerful optimisation strategies enabled by the structural properties of the contexts: automatic context-splitting, a recursive state-space decomposition strategy; context-directed semi-external reachability analysis, an exhaustive analysis technique that reduces the memory pressure during verification through the use of external memory.

In the case of the CCS system, this approach enabled the analysis of up to 5 times larger state-spaces than traditional approaches.

Keywords: formal verification; context-aware model-checking; OBP; observer-automata...

1 Introduction

Since their introduction in the early 1980s, model-checking [17, 5] provides an automated formal approach for the verification of complex requirements of hardware and software systems. This technique relies on the exhaustive analysis of all states in the system to check if it correctly implements the specifications, usually expressed using temporal logics. However, because of the internal complexity of the systems studied, model-checking is often challenged with unmanageable large state-space, a problem known as the state-space explosion problem [6, 15]. Numerous techniques, such as symbolic model-checking [3] and partial-order reduction [19], have been proposed to reduce the impact of this problem effectively pushing the inherent limits of model-checking further and further.

The Context-aware Verification has been recently introduced [9] as a new technique of state-space decomposition that enables compositional verification of requirements. This technique reduces the set of possible behaviors (and thus

the state-space) by closing the SUS with a well defined finite and acyclic environment. The explicit and formal specification of this environment enables at least three different decomposition axes: *a)* the environment can be decomposed in contexts, thus isolating different operating modes; *b)* these contexts enable the automatic partitioning of the state-space into independent verification problems; *c)* the requirements are focused on specific environmental conditions.

In this study we apply the Context-aware Verification technique for modelling and requirement validation of a automotive Cruise-Control System, a system that automatically controls the speed of cars. Using this approach we verified three important requirements of the CCS, identifying one subtle concurrency bug that could lead to very dangerous situations. Furthermore, the importance of the Context-aware Verification approach is emphasised through the successful analysis of up to 4.78 larger state-space than traditional approaches. Result which was made possible by relying on the complementarity of two powerful optimisation strategies enabled by the explicit environment specification: a recursive state-space decomposition strategy and an exhaustive analysis technique that reduces the memory pressure during verification through the use of external memory.

This study starts by introducing the Context-aware Verification approach in Section 2 along with the CDL language (Section 2.1) and two innovative analysis techniques addressing the state-space explosion problem (Section 2.2). The CCS specifications are introduced in Section 3 and the CDL encoding of the requirements and environment are overviewed in Section 3.2 and Section 3.3. The verification results are presented in Section 3.4. Section 4 overviews related research emphasising the complementarity with the Context-aware Verification. Section 5 concludes this study presenting future research directions.

2 Context-aware Verification

Context-aware Verification, focuses on the explicit modeling of the environment as one or more contexts, which then are iteratively composed with the system-under-study (SUS). The requirements are associated and verified in the contexts that correspond to the environmental conditions in which they should be satisfied, and automated context-guided state-space reduction techniques can be used to further push the limits of reachability analysis. All these developments are implemented in the OBP *Observation Engine* [9] and are freely available¹.

When verifying properties, through explicit-state model checking, the system explores all the behaviors possible in the SUS and checks whether the verified properties are true or not. Due to the exponential growth of system states relative to the number of interacting components, most of the time the number of reachable configurations is too large to be contained in memory. Besides using techniques like the ones described in Section 4, to alleviate this problem the system designers manually tune the SUS to restrict its behaviors to the ones

¹ OBP *Observation Engine* website: <http://www.obpcdl.org>

pertinent relative to the specified requirements. This process is tedious, error prone and poses a number of methodological challenges since different versions of the SUS should be kept sound, in sync and maintained.

To address these issues, Context-aware Verification technique proposes to restrict the model behaviors by composing it with an explicitly defined environment that interacts with the SUS. The environment enables a subset of the behaviors of the model. This technique reduces the complexity of the exploration by limiting its scope to a reduced set of behaviors related to specific environmental conditions. Moreover, this approach solves the methodological issues, since it decouples the SUS from its environment, thus allowing their refinement in isolation.

Context-aware reduction of system behaviors is particularly interesting in the case of complex embedded system, such as automotive and avionics, since they exhibit clearly identified operating modes with specific properties associated with these modes. Unfortunately, only few existing approaches propose practical ways to precisely capture these contexts in order to reduce formal verification complexity and thus improve the scalability of existing model checking approaches.

2.1 Environment Modeling with CDL formalism

The *Context Description Language*²(CDL) was introduced to formalize the environment specification [8]. The core of the CDL language is based on the concept of **context**, which has an acyclic behavior communicating asynchronously with the system. The environment is specified through a number of such contexts. The interleaving of these contexts generates a labelled-transition system representing all behaviors of the environment, which can be fed as input to traditional model-checkers. Moreover, the CDL enables the specification of requirements through properties that are verified by the OBP *Observation Engine*.

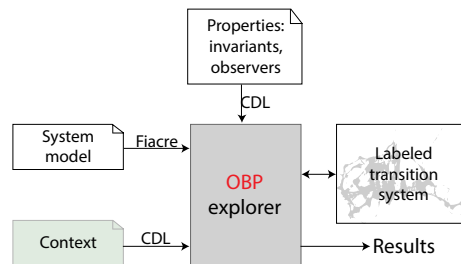


Fig. 1: OBP *Observation Engine* overview

Fig. 1 shows a global overview of the OBP *Observation Engine*. The SUS is described using the formal language Fiacre [11], which enables the specification of interacting behaviors and timing constraints through timed-automata.

² For detailed CDL syntax and semantics, see www.obpcdl.org.

The surrounding environment and the requirements are specified using the CDL formalism.

2.2 Context-guided State-space Decomposition and Reachability

Automatic Context Splitting: OBP *Observation Engine* integrates a powerful context-guided state-space reduction technique which relies on the automated recursive partitioning (splitting) of a given context in independent sub-contexts [8]. This technique is systematically applied by OBP *Observation Engine* when a given reachability analysis fails due to lack of memory resources to store the state-space.

After splitting $context_i$, the sub-contexts are iteratively composed with the model for exploration, and the properties associated with $context_i$ are checked for all sub-contexts. Therefore, the global verification problem for $context_i$ is effectively decomposed into K_i smaller verification problems.

Context-directed semi-external reachability analysis: OBP *Observation Engine* also implements a new exhaustive analysis algorithm that reduces the memory consumption by using the external storage to store the "past-states" of the SUS [18]. This algorithm, named PastFree[ze], relies on the isolation of the acyclic components of the SUS, which are used to drive the reachability analysis. The nodes of the graph induced by the context identifies "clusters of states" that can be freed from memory and saved to disk.

3 Case Study: Cruise-Control System

This section provides a description and some of the requirements of an automotive Cruise-Control System (CCS), which is the case study studied in this paper.

3.1 SUS Model

Functional Overview. The CCS main function is to adjust the speed of a vehicle. After powering the system on, the driver first has to capture a target speed, then it is possible to engage the system. This target speed can be increased or decreased by $5km/h$ with the tap of a button.

There are also several important safety features. The system shall disengage as soon as the driver hits the brake/clutch pedal or if the current vehicle speed (s) is off bounds ($40 < s < 180km/h$). In such case, it shall not engage again until the driver hits a "resume" button. If the driver presses the accelerator, the system shall pause itself until the pedal gets released.

Physical architecture. The CCS is composed of four parts (cf. Fig. 2). A *control panel* providing the controls needed to operate the system. An *actuation* that is able to capture the current speed and, once enabled, to adjust the vehicle speed toward the defined target. A *health monitoring* component that detects critical events and relays them to the other components. A *system-center* component that acts as a controller.

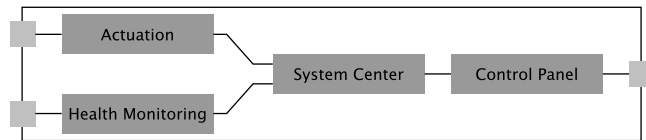


Fig. 2: CCS Physical Architecture.

The *control panel* provides the buttons needed by the driver to operate the system:

- PowerOn: Turns the system on;
- PowerOff: Turns the system off;
- Set: Capture the current speed of the vehicle as the target speed;
- Resume: Engage the control speed function of the system;
- Disengage: Disengage the control speed function of the system;
- Inc: Increments the current target speed by 5kmh;
- Dec: Decrements the current target speed by 5kmh.

The *control panel* is not responsible for handling those operations. However it should relay them to the *system-center*.

The *actuation* provides the tools for the system to interact with the vehicle. It can capture the current speed of the vehicle and set it as the new target speed. Once the CCS enabled, the *actuation* is responsible for controlling the vehicle speed accordingly.

The *health monitoring* is responsible for monitoring the system and the vehicle for events that can potentially impact the behavior of the CCS:

- The driver hits the brake pedal (induces disengagement);
- The driver hits the clutch pedal (induces disengagement);
- The speed of the vehicle goes out of bounds (induces disengagement);
- The driver presses the accelerator pedal (pauses the speed control function);
- The driver releases the accelerator pedal (resumes the speed control function);

As for the *control panel*, this component is not responsible of handling the consequences of such events. However, it should relay them to the *system center* which shall handle them.

The *system center* is the "core" of the CCS. It is responsible for handling events detected by both the *control panel* and the *health monitoring* components. To do so, it shall be able to impact the behaviors of all other components.

3.2 Requirements

This section lists three requirements of the CCS system and shows how to model them using the CDL formalism.

REQ₁: **When** an event inducing a disengagement is detected, the actuation component **should not** be allowed to control the vehicle speed **until** the system is explicitly resumed.

REQ₂: The target speed **should never** be lower than 40km/h nor higher than 180km/h.

REQ₃: **When** the system is powered off (PowerOff button), the target speed should be reset and be considered unset **when** the system is turned on again.

REQ₁ can be encoded using the observer automaton presented in Fig. 3. To encode this observer using the CDL formalism we first need to introduce the events triggering the transitions.

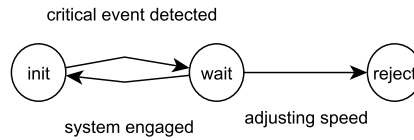


Fig. 3: Observer automaton for Req₁.

Listing 1: Event declaration in CDL language

```

1 predicate disengageIsRequested is {
2     HealthMonitoring@DisengageRequested }
3
4 event criticalEventDetected is {
5     disengageIsRequested becomes true }
6
7 event systemAdjustingSpeed is {
8     send any from Actuation to Car }
9
10 event systemEngaged is {
11     SystemCenter@Engaged becomes true }
  
```

In Listing 1, *disengageIsRequested* is a predicate on the HealthMonitoring process returning true if the process is in DisengageRequested state. *disengageIsRequested becomes true* expresses a rising edge of the predicate, which is an observable event in OBP *Observation Engine*. Thus, the event *criticalEventDetected* expresses that the HealthMonitoring process just entered in DisengageRequested state from a different state. *systemAdjustingSpeed* (lines 7-8) expresses an observable event that triggers when the Actuation process sends any message to the Car process. Since, in our model, the only messages going through this channel are speed adjusting requests it matches an attempt from the actuation to control the speed. Using these events the observer automaton in Fig. 3 is specified in Listing 2.

Listing 2: The observer automaton for REQ₁ in CDL language

```

1 property REQ1 is {
2     start    — criticalEventDetected -> wait;
3     wait     — systemAdjustingSpeed -> reject;
4     wait     — systemEngaged -> start }

```

REQ₂ can be encoded by declaring a predicate matching this property, see Listing 3 (lines 4-7). Since the requirements need to account for the SUS model, the allowed range for the target speed is extended to $[40..180] \cup \{Unset\}$.

Listing 3: Specifying REQ₂ as a predicate and REQ₃ as an observer automaton using CDL

```

1 predicate targetSpeedIsUnSet is {
2     Actuation@UnSet or Actuation@UnsetSetting }
3
4 predicate REQ2 is {
5     (Actuation:targetSpeed >= 40
6     and Actuation:targetSpeed <= 180)
7     or     targetSpeedIsUnSet }
8
9 property REQ3 is {
10    start — (not targetSpeedIsUnSet)
11            $\wedge$  systemTurnsOn          -> reject
12 }

```

The REQ₃ encoding is presented in Listing 3 (lines 9-12), and it can be interpreted as: "If the target speed is already set (*not targetSpeedIsUnset*) when the system turns-on (*systemTurnsOn*) the verification fails ($- > reject$)".

3.3 Environment Modeling

In the case of Context-aware Verification, the environment modeling should be seen as a methodological phase that needs to balance two important constraints while building the context. First the context has to cover enough behaviors to be considered valid for a given property. But at the same time it has to be small enough to be possible to exhaustively explore the product of its composition with the SUS. In the case of the CCS the environment, presented in Listing 4, is built from three distinct actors modeling: *a)* a nominal scenario, *b)* a perturbator *c)* and "ticks". The basic scenario can be seen as a linear use case of the CCS that covers all the functionality involved by the properties we aim to verify. The perturbator is a wide alternative including changes of the vehicle speed within the allowed range or not, pressions on the pedals and the panel buttons. The perturbator stresses the SUS against a number of possible unexpected behaviors of the environment. Once these two actors are composed, we get a wide range of variations of the basic scenario using the capabilities of the perturbator at all stages. A "tick" is an event sent to the car to trigger a broadcast of its speed to the involved components in the CCS. In other words, each tick allows the system to "read" the current speed once. The basic scenario we use involves 2 changes

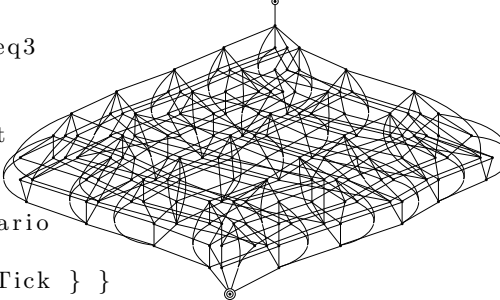
of speed, the perturber can add another, so for the system to be able to react we need at least 3 "ticks". Adding a "tick" allows to see how the system reacts if there is no change of the current speed so it covers more possibilities. While 4 "ticks" is enough, the more "ticks" we add, while still being able to explore the behaviors of the system, the higher the coverage of the possible behaviors. We also use this variable (the number of "ticks") as a way to make the exploration bigger to stress our tools.

Listing 4: Context description and the corresponding LTS representation.

```

1  cdl myContext is {
2      properties req1, req3
3      assert req2
4      init is {
5          evtBtnStart
6      }
7      main is {
8          basic_scenario
9          ||    perturber
10         ||    loop 4 evtTick } }

```



3.4 Verification Results

This section presents the results obtained for the verification of the three requirements previously presented, emphasizing the importance of the Context-aware Verification approach, which through the use of the PastFree[ze] algorithm enabled the analysis of a 2.4 times larger state-space and through the joint use of PastFree[ze] and automatic split technique 4.78 times larger state-space compared to traditional breadth-first search (BFS) reachability algorithms.

The results presented in this study were obtained on a 64 bit Linux computer, with a 3.60GHz Intel Xeon processor, and 64GB RAM memory. We used OBP *Observation Engine* distribution version 1.4.6, which includes an implementation of the PastFree[ze] algorithm³.

During the exploration, the observer encoding Req_1 reaches its reject state meaning the property is not verified. This happens because of a flaw in the model. Upon receiving a "tick" event, the car broadcasts its current speed to both Actuation and Health Monitoring. Both components will react, the former by sending back an adjusting request, the later by detecting this critical event and by requesting a disengagement. The model should be adjusted so that the Actuation doesn't not attempt to adjust the speed if out of bounds, for example by filtering it first via the Health Monitoring.

³ The raw results presented in this study along with the source files and an OBP *Observation Engine* distribution are available for download on the OBP *Observation Engine* website at <http://www.obpcdl.org>

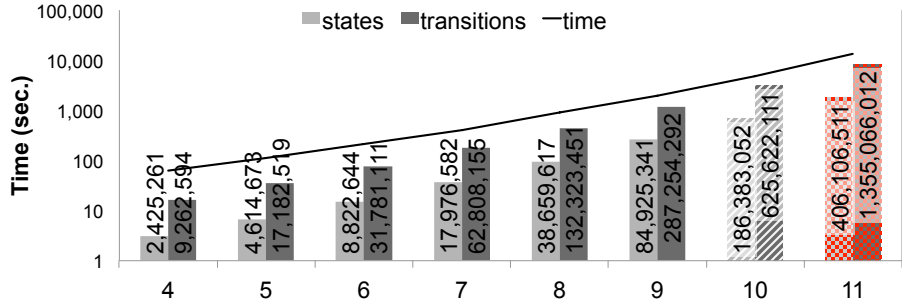


Fig. 4: Exploration results for contexts sending from 4 to 11 ticks

Exploration with BFS search. Fig. 4 presents the results in terms of states, transitions and exploration time obtained for the CCS model by varying the number of "ticks" sent by the context presented in Listing 4 from 4 to 11. The traditional BFS reachability analysis algorithm failed to explore all the state-space in the case of 10 and 11 ticks.

Exploration with PastFree[ze]. The PastFree[ze] analysis successfully explored the 10 ticks case obtaining state-space 2.2 times larger than the 9 tick case. However with the 64GB memory limit this technique failed to finish the exploration for 11 ticks. The results shown in Fig. 4 for the 11 ticks case were obtained on a computer with 128GB memory and were included just as a reference for better understanding the advantages of the automatic-split state-space decomposition presented in the following paragraph.

Exploration with splitting. To verify the three requirements for the 11 tick context we used the automatic-split technique which produced 9 sub-contexts which have been independently analysed by the OBP *Observation Engine* with the PastFree[ze] algorithm. The results of these explorations are presented in Fig. 5. It should be noted that in this case the traditional BFS algorithm would have failed to explore at least 3 of the obtained sub-contexts (the 1st, 2nd and 9th one) hence needing another split step for these cases, which was not needed for the PastFree[ze] technique. Another important observation is that while with the automatic-split technique the state-space was decomposed in 9 partitions these partitions are not disjoint. Hence their exploration analysed 779 739 813 states and 2 611 647 510 transitions which represents the analysis of 1.92 times more states and 1.93 more transitions than the exact state-space presented in Fig. 4 (the 11 ticks bars). Nevertheless, we believe that this is a small price to pay for the possibility of analysing a 4.78 times larger state-space without the need of doubling the physical memory of the machine.

4 Related Work

Model checking is a technique that relies on building a finite model of a system of interest, and checking that a desired property, typically specified as a temporal

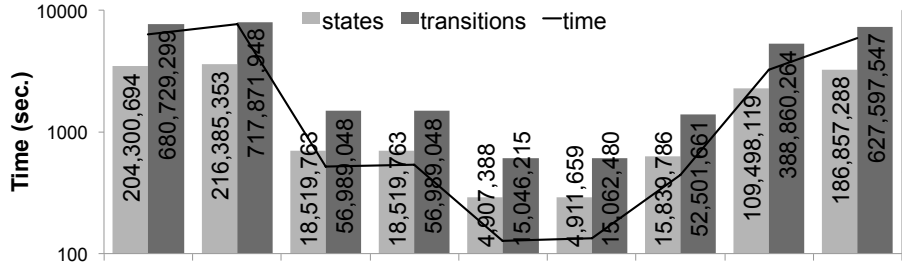


Fig. 5: Results for the analysis of the 9 partitions of the 11 tick context

logic formula, holds for that model. Since the introduction of this technology in the early 1980s [17], several model-checker tools have been developed to help the verification of concurrent systems [14, 1].

However, while model-checking provides an automated rigorous framework for formal system validation and verification, and has successfully been applied on industrial systems it suffers from the state-space explosion problem. This is due to the exponential growth of the number of states the system can reach with respect to the number of interacting components. Since its introduction, model checking has progressed significantly, with numerous research efforts focused on reducing the impact of this problem, thus enabling the verification of ever larger systems. Some of these approaches focus on the use of efficient data-structures such as BDD [3] for achieving compact state-space representation, others rely on algorithmic advancements and the maximal use of the available resources such as external memories [10]. To prune the state-space, techniques such as partial-order reduction [13, 16, 19, 13] and symmetry reduction [7] exploit fine-grain transition interleaving symmetries and global system symmetries respectively. Yet other approaches, like bounded model-checking [4] exploit the observation that in many practical settings the property verification can be done with only a partial (bounded) reachability analysis.

The successful application of these methods to several case studies (see for instance [2] for aerospace examples) demonstrates their maturity in the case of synchronous embedded systems. However, even though these techniques push the limits of model-checking ever further, the state-space explosion problem remains especially in the case of large and complex asynchronous systems.

Besides the previously cited techniques that approach the property verification problem monolithically, compositional verification [12] focus on the analysis of individual components of the system using assume/guarantee reasoning (or design-by-contract) to extract (sometimes automatically) the interactions that a component has with its environment and to reduce the model-checking problem to these interactions. Once each individual component is proved correct the composition is performed using operators that preserve the correctness.

Our approach can be seen as a coarse-grain compositional verification, where instead of analyzing the interactions of individual components with their neighboring environment we focus on the interactions of the whole system with its surrounding environment (context). Conversely to "traditional" techniques in

which the surrounding environment is often implicitly modeled in the system (to obtain a closed system), we explicitly describe it separately from the model. By explicitly modeling the environment as one (or more) formally defined context(s) and composing it with the system-under-study we can conduct the full system verification.

5 Conclusion and Perspectives

In this paper we have used the Context-aware Verification technique for the analysis of three requirements of a Cruise-Control System. The asynchrony of this system renders traditional model-checking approaches almost impossible. Using the environment reification through the CDL formalism this task becomes manageable by relying on two powerful optimisation strategies. These strategies rely on the structural properties of the CDL contexts and enable the reachability analysis of orders of magnitude larger models.

While the approach presented in this paper offers promising results, for this technique to be used on industrial-scale critical systems, we are currently working on a sound methodological framework that formalizes the context coverage with respect to the full-system behavior and assist the user on initial context specification.

References

1. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In: Proc. of Workshop on Verification and Control of Hybrid Systems III. pp. 232–243. No. 1066 in Lecture Notes in Computer Science, Springer-Verlag (Oct 1995)
2. Boniol, F., Wiels, V., Ledinot, E.: Experiences using model checking to verify real time properties of a landing gear control system. In: Embedded Real-Time Systems (ERTS). Toulouse, France (2006)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: 5th IEEE Symposium on Logic in Computer Science. pp. 428–439 (1990)
4. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19(1), 7–34 (2001)
5. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) *Logics of Programs*, Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer Berlin Heidelberg (1982), <http://dx.doi.org/10.1007/BFb0025774>
6. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8(2), 244–263 (1986)
7. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9(1-2), 77–104 (1996)
8. Dhaussy, P., Boniol, F., Roger, J.C.: Reducing state explosion with context modeling for model-checking. In: 13th IEEE International High Assurance Systems Engineering Symposium (Hase'11). Boca Raton, USA (2011)

9. Dhaussy, P., Boniol, F., Roger, J.C., Leroux, L.: Improving model checking with context modelling. *Advances in Software Engineering* ID 547157, 13 pages (2012)
10. Edelkamp, S., Sanders, P., Šimeček, P.: Semi-external LTL model checking. In: Gupta, A., Malik, S. (eds.) *Computer Aided Verification, Lecture Notes in Computer Science*, vol. 5123, pp. 530–542. Springer Berlin Heidelberg (2008)
11. Farail, P., Gauffillet, P., Peres, F., Bodeveix, J.P., Filali, M., Berthomieu, B., Rodrigo, S., Vernadat, F., Garavel, H., Lang, F.: FIACRE: an intermediate language for model verification in the TOPCASED environment. In: *European Congress on Embedded Real-Time Software (ERTS)*. SEE, Toulouse (january 2008)
12. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: *SPIN'03* (2003)
13. Godefroid, P.: The Ulg partial-order package for SPIN. *SPIN Workshop* (1995)
14. Holzmann, G.: The model checker SPIN. *Software Engineering* 23(5), 279–295 (1997)
15. Park, S., Kwon, G.: Avoidance of state explosion using dependency analysis in model checking control flow model. In: *Proceedings of the 5th International Conference on Computational Science and Its Applications (ICCSA '06)*. vol. 3984, pp. 905–911. Springer-Verlag, LNCS (2006)
16. Peled, D.: Combining Partial-Order Reductions with On-the-fly Model-Checking. In: *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*. pp. 377–390. Springer-Verlag, London, UK (1994)
17. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. pp. 337–351. Springer-Verlag, London, UK (1982)
18. Teodorov, C., Leroux, L., Dhaussy, P.: Past-free reachability analysis. reaching further with DAG-directed exhaustive state-space analysis. (submitted to) *29th IEEE/ACM International Conference on Automated Software Engineering* (2014)
19. Valmari, A.: Stubborn sets for reduced state space generation. In: *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*. pp. 491–515. Springer-Verlag, London, UK (1991)