ASAP: An Extensible Platform for State Space Analysis*

Michael Westergaard[1], Sami Evangelista[1] and Lars Michael Kristensen[2]

[1] Department of Computer Science, Aarhus University, Denmark
mw@cs.au.dk, evangeli@cs.au.dk
[2] Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

Abstract. The ASCoVeCo State space Analysis Platform (ASAP) is a tool for performing explicit state space analysis of coloured Petri nets (CPNs) and other formalisms. ASAP supports a wide range of state space reduction techniques and is intended to be easy to extend and to use, making it a suitable tool for students, researchers, and industrial users that would like to analyze protocols and/or experiment with different algorithms. This paper presents ASAP from these two perspectives.

1 Introduction

State space analysis (or model checking) is one of the main approaches to model-based verification of concurrent systems and is one of the most successfully applied analysis methods for formal models. Its main limitation is the *state explosion problem*, i.e., that state spaces of systems may have a large number of reachable states, meaning that they are too large to be handled with the available computing power (CPU speed and memory). Methods for alleviating this inherent complexity problem is an active area of research and has led to the development of a large collection of *state space reduction methods*. These methods have significantly broadened the class of systems that can be verified and state spaces can now be used to verify systems of industrial size. A computer tool supporting state space analysis must implement a wide range of reduction algorithms since no single method works well on all systems. The software architectures of many such tools, e.g., CPN Tools [5], SPIN [1], make it difficult to support a collection of state space reduction methods in a coherent manner and to extend the tools.

This paper presents the ASCoVeCo State Space Analysis Platform (ASAP) [2] which is currently being developed in the context of the ASCoVeCo research project [3]. ASAP represents the next generation of tool support for state space exploration and analysis of CPN models [13] and other formal models. The aim and vision of ASAP is to provide an open platform suited for research, educational, and industrial use of state space exploration. This means that ASAP

* Supported by the Danish Research Council for Technology and Production.

supports a wide collection of state space exploration methods and has an architecture that allows the research community to extend the set of supported methods. Furthermore, we aim at making ASAP sufficiently mature to be used for educational purposes, including teaching of advanced state space methods, and for use in industrial projects as has been the case with CPN Tools and Design/CPN [6].

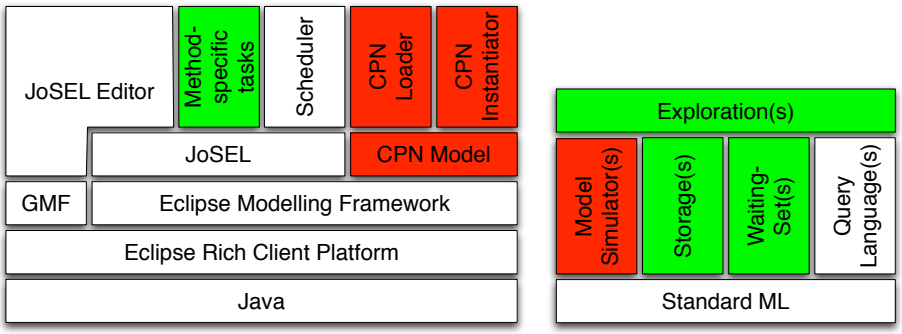
This paper is structured as follows. The next section is an overview of the architecture of our tool. Section 3 briefly describes how the tool can be extended with new verification algorithms or modeling languages. A few benchmarks comparing our tool with CPN Tools and DiVinE [7] are presented in Section 4. Section 5 concludes this work and presents some future extensions to our tool.

2 Architecture of ASAP

The ASAP platform consists of a graphical user interface (GUI) and a state space exploration engine (SSE engine). Figure 1(a) shows the software architecture of the graphical user interface which is implemented in Java based on the Eclipse Rich Client Platform [8]. The software architecture of the SSE engine is shown in Fig. 1(b). It is based on Standard ML and implements the state space Exploration and model checking algorithms supported by ASAP. The choice of SML for the SSE engine is primarily motivated by its ability to easily specify and extend algorithms. The state space exploration and model checking algorithms implemented rely on a set of Storage and Waiting Set components for efficient storage and exploration of state spaces. Furthermore, the SSE engine implements the Query Languages(s) used for writing state space queries and to verify properties.

User interface. The ASAP GUI makes it possible to create and manage *verification projects* consisting of a collection of *verification jobs*. Verification jobs are constructed and specified using the verification Job Specification and Execution Language (JoSEL) [17] and the JoSEL Editor. We will briefly highlight the key ideas of JoSEL later. JoSEL and the JoSEL Editor are implemented using the Eclipse Modeling Framework and GMF, the Graphical Modeling Framework. The ASAP GUI additionally has a Model Loader component and a Model Instantiator component that can load and instantiate, e.g., CPN models [13] created with CPN Tools [5]. It is worth noticing that only the dark gray (red) boxes in Fig. 1 (CPN Model Loader, CPN Model Instantiator, and CPN Model Representation as well as the Model Simulator(s) component of the SSE engine) are language specific; all other components are independent of any concrete modeling language, and indeed we have implemented components for loading models specified in DVE, the input language of the DiVinE model checker.

The GUI has two different *perspectives* for working with verification projects: An *editing perspective* for creating and editing verification jobs, and a *verification perspective* for inspecting and interpreting verification results. Figure 2(a) shows a snapshot of the graphical user interface in the editing perspective. The user



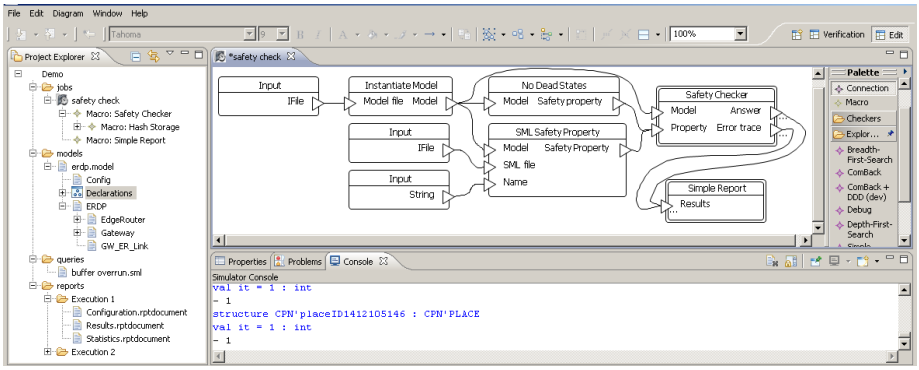
(a) Graphical user interface (b) SSE engine

Fig. 1. ASAP platform architecture.

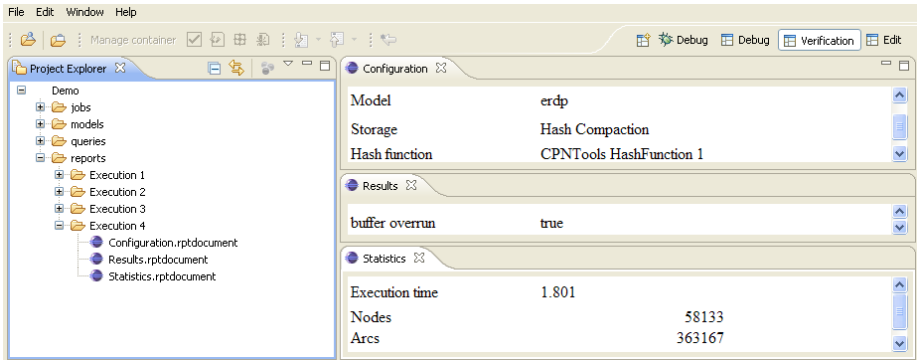
interface consists of three main parts apart from the usual menus and tool-bars at the top. To the left is an overview of the verification projects loaded, in this case just a single project named Demo is loaded. A verification project consists of a number of verification jobs, models, queries, and generated reports. In this case there is one verification job, safety check, concerned with checking safety properties. A CPN model named ERDP is loaded, which is a CPN model of an edge router discovery protocol from an industrial case study [14]. We have one query, checking if buffers of the model overflow, and two reports from two different verification job executions. At the bottom is a console making it possible to interact directly with the SSE engine using SML. This allows experimenting with the tool and issuing queries that need not be stored as part of the verification project. The area at the top-right is the editing area used to edit queries and verification jobs. Here, the safety checker job is being edited and the window shows its graphical representation in JoSEL. Other components may be added to this job by using the tool palette adjacent to the editing area.

A snapshot of the verification perspective is shown in Figure 2(b). Here, a verification report is opened. It consists of three parts. The Configuration report lists general information like the model name or the different reduction techniques enabled, e.g., hash compaction in this case. The Results report specifies which properties were checked and whether or not they hold. In case of error, it also displays a counter-example that proves why the property does not hold. The Statistics report gives information on the state space, the exploration time and additional information depending on the reduction techniques used.

The Job Specification and Execution Language. JoSEL [17] is a graphical language inspired by data-flow diagrams that makes it possible to specify the formal models, queries, state space explorations, and processing of analysis results that constitute a verification job. The top-right panel in Fig. 2(a) shows a graphical representation of a JoSEL job that we use to illustrate the different key ideas behind this language.



(a) The editing perspective



(b) The verification perspective

Fig. 2. Snapshots of the graphical user interface.

In JoSEL, *tasks* are the basic units of computation used as blocks to construct jobs. A task can, for instance, load a CPN model from a file or explore the state space of a model using a specific search algorithm. Tasks are graphically represented by rounded boxes.

A *job* consists of a set of interconnected tasks. Connections are used to specify a producer/consumer scenario: tasks can produce data that can be in turn used by another task. Each task has a set of *input ports* that specify the type of data it waits for in order to be executed. Once its execution is finished, the production of the task is specified by *output ports*. Both are graphically represented using small triangles placed at the left of tasks for inputs and at the right for outputs. In our example, the Instantiate Model task takes as input a CPN Model file, and from it, produces a Model which is an SML representation of the CPN model usable by the SSE engine (see the next paragraph and Section 3). This one can be consumed by the No dead states and SML Safety Property tasks that instantiate two properties, absence of deadlock and a user defined property. These are analyzed by the Safety checker task.

At this level, the type of algorithm or reduction techniques used by the Safety checker are not visible to the user. This is because this task has been defined as a *macro*. A macro is at the same time a job (described by the user with tasks and connections) and a task that can be part of other jobs. The graphical representation of macros differs from the one of tasks in that the rounded box is drawn using a double outline. Besides the advantages of clarifying the view of jobs and allowing the reuse of macros along different jobs, their use allows different levels of abstraction. Many users are not interested in the details of the safety checker whereas some with more background in model checking would perhaps like, for a specific model, to use a specialized search algorithm assumed to be especially efficient in that particular case. Double-clicking on the Safety checker macro expands the view of the macro and allows the user to tune the way properties are checked.

The main motivation of JoSEL is to provide the user with an intuitive and graphical language that allows different level of abstractions for users with different background in model checking such as: students, researchers, and industrial users.

The state space search engine. It is commonly agreed upon in the model checking community that no reduction technique works well on all models and that algorithms and methods are usually designed for a specific stage of the verification process. Therefore tools have to support several algorithms in order to prove useful.

Currently, ASAP supports checking deadlocks in systems and user specified safety properties. ASAP implements a broad range of techniques and below we briefly mention some of them.

Bit-state hashing and *hash-compaction* [12] are incomplete methods based on hashing and are generally used prior to any other analysis technique for their ability to quickly discover errors rather than proving their absence. The *sweep-line* method [15] exploits a notion of progression exhibited by many protocols to garbage collect states during the search thereby lowering peak memory requirements. The recently developed *ComBack* method [18,11] is especially suited to models having complex states, e.g., with many tokens in places for CPNs. It makes it possible to represent states with 16–24 bytes independently from the model, hence achieving a memory reduction close to the one provided by hash compaction without the loss of precision associated with that method. If, using these techniques, memory is still lacking, the user can switch to efficient *disk-based* algorithms [4,10]. In experiments reported in [10] we were able to explore large state spaces with 10^8–10^9 states in a reasonable time (4–20 hours).

3 Extending ASAP

An important design guideline of ASAP is to provide a flexible and modular architecture that can be easily extended with new algorithms or modeling languages. We give in this section a brief overview of how this can be done.

```
functor SweepLineExploration(
   structure Model  : MODEL
   structure Storage: STORAGE
   val progressValue: Model.state -> int): EXPLORATION =
struct
   fun explore initialStates = ...
end
```

(a) First of lines of the sweep-line search algorithm in the SSE engine.

```
class SweepLineExplorationTask implements FunctorTask {
   String getName () { return "Sweep Line Exploration"; }
   String getFunctor () { return "SweepLineExploration"; }
   Value getReturnType () {
        return new Value ("Traversal", Exploration.class); }
   Value[] getParameters () {
        return new Value[] {
           new Value ("Model", Model.class),
           new Value ("Storage", Storage.class),
           new Value ("Progress Measure", Measure.class) }; }
   Exploration exec (Model m, Storage s, Measure p) {
        Exploration e = new Exploration (m.getSimulator ());
        m.getSimulator ().evaluate (
           e.getDeclaration () + " = " + getFunctor () +
           "(structure Model   = " + m.getStructure () +
           " structure Storage = " + s.getStructure () +
           " val progressValue = " + p.getName () + ")");
        return e; }
}
```

(b) Creation of a Sweep-Line Exploration task (see Fig 4) in the JoSEL editor.

Fig. 3. Integration of the sweep-line method in ASAP.

Integrating new algorithms. Let us suppose that we wish to integrate the sweep-line method [15] into ASAP. Only the light gray (green) boxes in Fig. 1 are method specific and have to be considered for this integration. On the SSE engine side we have to implement the search algorithm used by this method. Since this one is independent from any storage method, and uses its own waiting set component to store unvisited states, we only have to implement an exploration component. The engine is based on a number of SML signatures (the equivalent of JAVA interfaces) among which the most important ones are: `EXPLORATION` that describes search algorithms, e.g., depth-first search; `STORAGE` that describes data structures used to store visited states, e.g., hash tables; and `MODEL` used to describe language dependent features of the analyzed model, e.g., how states are represented (see the next paragraph). The `SweepLineExploration` of Figure 3 is a generic `EXPLORATION`, i.e., an SML functor, that requires three parameters to

be instantiated: `Model`, the model of which the state space is explored; a `Storage` data structure used to store visited states; and a function, `progressValue`, that maps each state to a progress value, here an integer (see [15] for details). As this functor implements the `EXPLORATION` signature, it has to define a function, `explore`, that explores the state space from some starting state(s) and returns a storage containing the set of visited states upon termination of the algorithm. Because of space limitations we have left out the implementation of the `explore` function.

For these changes to be visible in the graphical interface, we then have to extend the JoSEL language with the Method-specific tasks of Fig. 1. The main one, Sweep Line Exploration is graphically represented in Fig. 4. It corresponds to the instantiation of functor `SweepLineExploration` and is implemented in the JoSEL editor by the `SweepLineExplorationTask` class of Fig. 3. This one inherits from `FunctorTask`, which is used to describe JoSEL tasks that simply consist of the instantiation of a functor. The methods `getName` and `getFunctor` return the name of the task, i.e., the label appearing in the graphical representation of the task, and the name of the underlying SML functor. The input and output ports (their names and types) of the task are specified by the `getParameters` and `getReturnType` methods. Note that a `FunctorTask` can only have one output port, namely, the SML structure resulting from the instantiation of the functor. Also, there usually is a one-to-one mapping between the parameters of the functor and the items returned by method `getParameters`, as it is the case here. The last method, `exec`, specifies the SML code that is interpreted as the task is executed. Its parameters match the list of output ports specified by method `getParameters`.

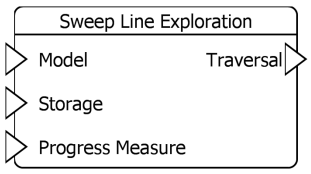


Fig. 4. Graphical representation of the Sweep Line Exploration task

Integrating new modeling languages. All search algorithms implemented by the SSE engine receive a model, which from the SSE engine point of view, is an SML structure implementing the `MODEL` signature (see Fig 5). To be valid, such a structure must define two types: the type of states and the type of events. For CPNs, the `state` type consists of a set of multi-sets over typed tokens and an `event` is a pair composed of a transition identifier and an instantiation for the variables of this transition. To be able to explore the state space of the model, the engine must know its initial state(s) and from a given state how to calculate its successor(s). This is the purpose of functions `initialStates` and `succ`. Non-deterministic systems are supported. Indeed, both `initialStates` and `succ` return a list of states (rather than single states) along with their enabled events. Functions `stateToString` and `eventToString` return a user readable representation of states and events that can be used, for instance, to display counter-examples.

```
signature MODEL = sig
   type state
   type event
   val initialStates:  unit -> (state * event list) list
   val succ:  state * event -> (state * event list) list
   val stateToString: state -> string
   val eventToString: state -> string
end
```

Fig. 5. The `MODEL` signature

Note that providing a model structure is the minimal requirement to be able to use the SSE engine. Many algorithms or reduction techniques expect more information, e.g, a state serializer for external algorithms or an independence relation for partial order reduction.

The easiest way to integrate a new specification language is to write a compiler that, from a specification file, can produce an SML `MODEL` structure. Since the other components of the engine are independent of any concrete language, those will remain unchanged. Although our work focuses on the development of language independent algorithms, the architecture of the SSE engine does not prevent us from integrating algorithms or reduction techniques specifically tailored for a specific language, e.g., search algorithms that exploit Petri net invariants. It is sufficient to define a new signature that extends `MODEL` with the desired features.

4 Benchmarks

The ability of ASAP to load CPN and DVE models makes it possible to experimentally compare different algorithms and reduction techniques on models from our own collection [3], e.g., [9,14], and on the numerous models of the BEEM database [16].

For comparison, we have shown in Table 1 the performance of ASAP compared to CPN Tools and DiVinE. For each Model, the table shows its number of States, the full exploration time with the Basis tool (CPN Tools or DiVinE) and with ASAP (with and without the ComBack method). Times are in seconds, and Speedup is the ratio between Basis and ASAP time. We see that ASAP performs significantly better than CPN Tools, achieving speedups of several orders of magnitude for both full state space generation and the ComBack method compared to full generation in CPN Tools. The performances of DiVinE and ASAP are comparable although slightly in favor of ASAP. On 50 models we observed an average speedup of 1.4 without using reduction. Even with the ComBack method, ASAP was able to perform at 0.7 of the speed of DiVinE despite the time overhead of the ComBack method.

Table 1. Performance of ASAP, CPN Tools, and DiVinE on some models.

	Model	States	Time			Time (ComBack)	
			Basis	ASAP	Speedup	ASAP	Speedup
CPN Tools	Dining Philosophers	$40 \cdot 10^3$	6,614	27	245	55	120
	Simple Protocol	$204 \cdot 10^3$	7,084	33	215	54	131
	ERDP	$207 \cdot 10^3$	19,351	112	173	197	98
	DYMO	$114 \cdot 10^3$	7,403	308	24	355	21
		Average on 4 models			164		92
DiVinE	brp2.6	$5.7 \cdot 10^6$	39	17	2.29	90	0.43
	firewire_tree.5	$3.8 \cdot 10^6$	227	525	0.43	388	0.59
	plc.4	$3.7 \cdot 10^6$	55	45	1.22	67	0.81
	rether.4	$9.5 \cdot 10^6$	51	34	1.52	191	0.27
		Average on 50 models			1.39		0.72

5 Conclusion

ASAP is a graphical tool based on Eclipse for the analysis of CPN models and other formalisms. It provides the user with an intuitive and graphical language, JoSEL, for specification of verification jobs. To alleviate the state explosion problem, ASAP implements several algorithms and reduction techniques. Among these are: hash compaction, the sweep-line method, the ComBack method and external memory algorithms. The tool has been designed to be easily extended with new algorithms or specification languages and its modular architecture allowed us to write a sweep-line plug-in and a DVE plug-in to load DVE models in a few days without modifying the rest of the code. ASAP is also very useful for experimenting with and comparing algorithms as it gives the possibility to analyze more than 60 CPN and DVE models from our test-suite or from the BEEM database. Last but not least, ASAP significantly outperforms CPN Tools regarding performance and performs as well as DiVinE for DVE models. For these reasons we believe the tool to be suitable for students, researchers, and industrial users who would like to analyze CPN models or to experiment with different verification algorithms.

ASAP has replaced CPN Tools in our group to perform verification tasks and has been used to analyze an edge router discovery protocol [14] and a dynamic mobile ad-hoc network routing protocol [9]; and to experiment with state space algorithms [18,10,11]. It is also intended to be used in a future advanced state space course at Aarhus University.

We are currently considering adding new features to ASAP. In its current version, ASAP can analyze deadlocks and verify user defined safety properties. In the design phase of communication protocols, properties are, however, often specified in a temporal logic, e.g., LTL or CTL. The integration of temporal logic in ASAP is therefore one of our main priorities.

Since parallel machines are nowadays widespread, it is crucial that model checkers take advantage of this additional computational power. As the SSE engine of ASAP is currently single-threaded we consider extending it with parallel and distributed algorithms. In particular, we are working on a parallel version of the ComBack method of which we briefly mentioned the principle in [11].

Availability ASAP is a stand-alone tool available for Windows XP/Vista, Linux, and Mac OS X. The current version, 1.0, can be freely downloaded from our web page [2].

References

1. SPIN homepage. http://www.spinroot.com/.
2. ASAP download. http://www.daimi.au.dk/~ascoveco/download.html.
3. The ASCoVeCo Project: Advanced State Space Methods and Computer tools for Verification of Communication Protocols . http://www.daimi.au.dk/~ascoveco/.
4. T. Bao and M. Jones. Time-Efficient Model Checking with Magnetic Disk. In *TACAS'2005*, volume 3440 of *LNCS*, pages 526–540. Springer, 2005.
5. CPN Tools homepage. http://www.daimi.au.dk/CPNTools/.
6. Design/CPN. http://www.daimi.au.dk/designCPN/.
7. DiVinE Homepage. http://divine.fi.muni.cz/.
8. Eclipse homepage. http://www.eclipse.org/.
9. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *ATPN'2008*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.
10. S. Evangelista. Dynamic Delayed Duplicate Detection for External Memory Model Checking. In *SPIN'2008*, volume 5156 of *LNCS*, pages 77–94. Springer, 2008.
11. S. Evangelista, M. Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *Transactions on Petri Nets and Other Models of Concurrency*, 2009. To appear.
12. G.J. Holzmann. An Analysis of Bitstate Hashing. *FMSD'1998*, 13:289–307, 1998.
13. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, In preparation.
14. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *INT'2004*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
15. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *FME'2002*, volume 2391 of *LNCS*, pages 549–567. Springer, 2002.
16. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'2007*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007. http://anna.fi.muni.cz/models/.
17. M. Westergaard and L.M. Kristensen. JoSEL: A Job Specification and Execution Language for Model Checking. In *CPN'2008*, volume 588 of *DAIMI-PB*, pages 83–102, 2008.
18. M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The ComBack Method – Extending Hash Compaction with Backtracking. In *ATPN'2007*, volume 4546 of *LNCS*, pages 445–464. Springer, 2007.