Software Architecture of Modern Model Checkers

Fabrice Kordon[1(✉)], Michael Leuschel[2], Jaco van de Pol[3], and Yann Thierry-Mieg[1]

[1] Sorbonne Université, CNRS UMR 7606 LIP6, 75005 Paris, France
{Fabrice.Kordon,Yann.Thierry-Mieg}@lip6.fr
[2] Institut für Informatik, Univ. Düsseldorf, Universitätsstr. 1, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[3] Department of Computer Science,
University of Twente, Enschede, The Netherlands
J.C.vandePol@utwente.nl

Abstract. Automated formal verification using model checking is a mature field with many tools available. We summarize the recent trends in the design and architecture of model checking tools. An important design goal of modern model checkers is to support many input languages (front-end) and many verification strategies (back-end), and to allow arbitrary combinations of them. This widens the applicability of new verification algorithms, avoids duplicate implementation of the analysis techniques, improves quality of the tools, and eases use of verification for a newly introduced high-level specification, such as a domain specific language.

1 Introduction

The evolution of model-based engineering and domain specific languages (DSL [73]) in industrial practice has led to a proliferation of small executable languages dedicated to a specific purpose. Model checking is a mature field [20] with many technological solutions and tools that can guarantee behavioral correctness of such specifications.

However, due to the complexity of the problem in general, different model checking tools are better at tackling different classes of systems, and it is difficult for an end-user to know beforehand which technique would be most effective for his or her specific model. It is thus highly desirable to embed such expert knowledge in a tool that integrates several solution engines (*e.g.* partial order reduction, decision-diagram based encoding, SAT/SMT techniques, etc.) behind a unified front-end.

Ideally a modern model checker should be adaptive, able to transparently select for a given model instance and a given property the best verification strategy. This design goal forces the software architecture of model checkers to evolve from tightly integrated or *monolithic* approaches to more open architectures that rely on *pivot representations* to support both many languages and many verification strategies.

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 393–419, 2019.
https://doi.org/10.1007/978-3-319-91908-9_20

The objective of this paper is to summarize the current situation of modern model checking tools in terms of architecture and usage to solve typical industrial-like problems, where specifications may not be written in a traditional verification language such as PROMELA [39], CSP [38], Petri nets [33], B [1] or TLA$^+$ [49].

Recent work also considers software verification (*i.e.* analyzing programs directly at the code level). Program verification mainly relies on strong abstractions of programs to cope with the combinatorial explosion caused by analysis at the instruction level, thus generating abstract models from software. This abstraction process for software verification is not considered directly in this paper; we focus on verification engines for model checking.

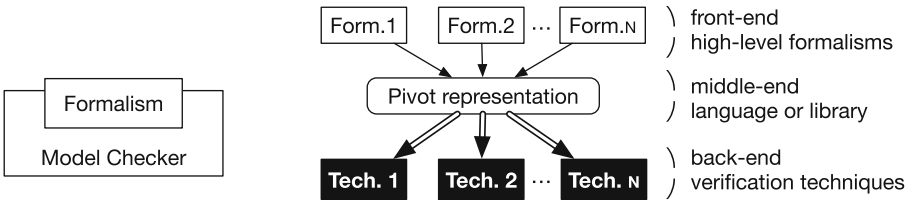
Another approach worth mentioning is the Electronic Tool Integration (ETI) platform [69]. This platform focuses on integration of tools (rather than algorithms) by providing a distributed coordination mechanism, enabling verification tasks that would not be possible in a single tool. Its successor, jETI [56] uses webservices technology and Eclipse support for seamless tool integration and graphical user interfaces. While ETI focuses on integration and coordination of existing tools, this paper focuses on integrating verification algorithms within a single, modular tool.

Section 2 presents the current trends in architectures for model checkers; Sect. 3 shows a first approach involving ProB and existing languages (*e.g.* Prolog, Java) while Sects. 4 and 5 are presenting updated visions (language based and library based) of such an architecture when analyzing high-level languages. Finally, Sect. 6 details two typical examples before a conclusion.

2 Trends on the Architecture for Model Checking

Model checkers exist now for more than three decades and have proven their usefulness to understand and debug complex systems. However, their software architecture is evolving, following a similar evolution as compilers, which were once monolithic but are now structured for a better reuse of code.

Figure 1 depicts such an evolution. On the left (Fig. 1a) is the "traditional" architecture, where a model checker is associated with a dedicated formalism



(a) Monolithic (b) Modular

Fig. 1. Evolution of model checking tool's architecture.

and proposes its verification algorithm, possibly with numerous variants and enhancements to perform its task efficiently. The most emblematic example is Spin [39].

Unfortunately, such an architecture has several drawbacks. First, the single entry point of the tool is the formalism it processes. Second, adapting the verification engine is also quite difficult since all the features of the input language are exploited and become naturally twisted with the algorithms themselves.

Progressively, several attempts have tried to separate the verification engine from the input formalism. Then, the notion of "pivot representation" naturally arises as the interface to an "upper level" dealing with the input specification. Below this "pivot representation", is a set of "verification engines" being able to process this pivot representation. Languages such as AltaRica [5], NUPN [31], or FIACRE [11], as well as a tool like the model checking Kit [64], could be seen as early attempts of this approach. Fixed-point equations have also been proposed as pivot representation to generalize multiple model checking questions [68]. In a similar fashion, the introduction of SMT-LIB [8] as a standard format for automated reasoning over data theories can be viewed as the successful introduction of a pivot representation.

Such an architecture (Fig. 1b) is similar to the font-end + middle-end + back-end architecture of current compilers. It has two main advantages. First, it decouples the high-level specification language from its verification. Then, the specification language designer may work independently from the verification mechanics as long as they provide a sound and formal semantics to their notation. This is of particular interest when input languages are numerous, because it does not hinder the access to efficient verification engines via the pivot representation. The bridging of AADL with FIACRE for verification purposes is a typical example of this interest [21].

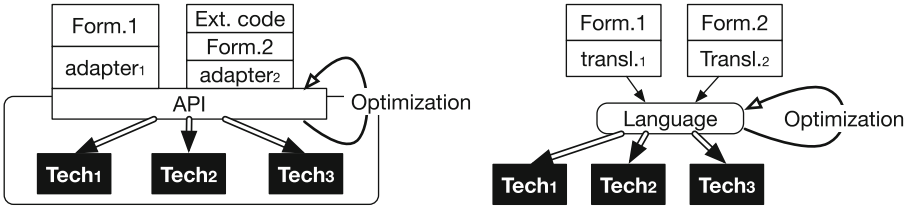
The second important advantage is the emphasis on the fundamentals of the semantics (expressed in the pivot representation) required to perform efficient model checking, thus providing a better access to various verification technologies (*e.g.* algorithms based on different approaches such as partial order reduction, the use of decision diagrams, the use of SAT/SMT solvers, etc.).

Moreover, it avoids, when dealing with the analysis of high-level specification, to choose between selecting (a priori) one verification technology, or performing as many translations as the number of selected verification engines.

The modular architecture of Fig. 1b can be interpreted in several ways (see Fig. 2):

- components may be linked together as object files and libraries, (see Fig. 2a), as this is the case for LTSmin [42] or SPOT [26],
- components may collaborate via an intermediate language (see Fig. 2b), as this is the case for ITS-Tools [71] (originally relying on enhanced decision diagrams) or Boogie [6] (originally relying on SMT solvers).

In the library based vision of the modular architecture of model checking tools, the pivot representation is materialized as an API. The role of such an



(a) Library based (b) Language based

Fig. 2. The two interpretations of the modular architecture of model checkers

interface is to expose the internal structure of the pivot representation, so that, efficient algorithms can be built on the one hand, while it remains easy to connect a higher formalism module on the other hand. Basically, formalisms are connected through adapters implementing the main semantic characteristics of the input formalism like the definition of a state or the successor function.

The main advantages of the library vision are: *(i)* it isolates the algorithms from the input pivot notation, allowing only access to relevant data structures, and *(ii)* it easily allows to embed executable code in the input formalism (with the necessary precautions to preserve the soundness of the input formalism semantics) that can be executed during model checking. Insertion of such code (`e.g.` data computation) is performed by the adapter.

Its main drawback is that one must cope with the existing data-structures when adding a new verification technology. This may hinder the addition of some new technology for which the existing data structures are not adapted (like connecting SAT/SMT based algorithms alongside existing automata-based ones).

In the language based vision of the modular architecture of model checking tools, the pivot representation is a language itself. Such a language offers a semantic support that is "agnostic" in the sense it can support various execution models. Connection with high-level languages is done thanks to a transformation into the pivot language, thus acting as an "assembly language" dedicated to verification.

The main advantages of this vision are: *(i)* it provides a strict barrier between high-level formalisms and the implemented verification modules that can use various relevant data structures suitable for the corresponding verification technology, and *(ii)* it enables possible optimization at the pivot language level so that the underlying selected verification algorithm features can be fully exploited. So, it is easier to plug new verification engines based on very different theory since adapted data structure can be then developed for this module only.

Unfortunately, it is quite complex to link executable code to a high-level formalism (under the necessary precautions to preserve the soundness of the input formalism semantics) without a heavy and complex support included in the pivot language itself. Such a feature is used in tools like Spin [39] (monolithic approach) or LTSmin [42] (modular/library based approach).

Obviously, the two interpretations of the modular architecture can be combined, thus exposing either a pivot language based on an API, or an API using a pivot language to connect to some underlying verification technology. The next section introduces some high-level logic based formalisms and investigates how they can be mapped to an efficient model checking engine.

3 High-Level Logic-Based Input Languages

High-level logic-based languages, i.e., specification languages which are not necessarily executable [36], can provide a convenient way to translate a wide variety of domain specific formalisms. Logic and set theory pervade large areas of computer science, and are used to express many formalisms, properties and concepts. On the one hand this explains the popularity of SAT and SMT solvers: many properties from wide areas of computer science can be expressed or compiled to logic. Similarly, the dynamic behaviour of a wide variety of languages and formalisms can be easily expressed in terms of a state-based formal method using logic and set theory.

Several formal methods have a common foundation in predicate logic, set theory and arithmetic: B [1], Event-B [2], TLA$^+$ [49], VDM and Z [66] are the most commonly used ones. Their high abstraction level make them a target for conveniently modelling a large class of systems to be validated. Indeed, the high abstraction level helps avoiding errors in the modelling process and can lead to a considerable reduction in modelling time [63]. These methods are also convenient for expressing the semantics of domain specific formalisms and develop model checking tools for them. E.g., the following tools use a translation to B to obtain a model checking tool for the source formalism: UML-B [65], SAP choreography [67], SafeCap [41], Coda [18].

One drawback of such a high-level input language is performance: determining the successor states of a given state may require constraint solving of logical predicates with quantification over higher-order data structures. As a simple example, we present the B encoding of derivation steps for a formal (possibly context-sensitive) grammar with productions P over an alphabet A. It is maybe not the most typical model checking example, but shows how easy one can translate a mathematical definition such as a formal grammar derivation step [40] into a high-level language like B. The B/Event-B model would just have a single event with four parameters L, R, a, b defined as

$$\textbf{event } rewrite(L, R, a, b) =$$

$$\textbf{when } (L \mapsto R) \in P \wedge a \in seq(A) \wedge b \in seq(A) \wedge cur = a \frown L \frown b$$

$$\textbf{then } cur := a \frown R \frown b \textbf{ end}$$

This is very close to the mathematical definition in theoretical computer science books such as [40]. The main difference is the use of $\frown$ for concatenating sequences and $seq(A)$ for finite sequences over the set A. Executing this event within a

model checker, however, requires a limited form of constraint solving: to compute the next state for a given value of *cur*, one needs to determine the possible decompositions of *cur* into three substrings a, L, b such that L is a left-hand side of a grammar production in P. E.g., given $P = \{N \mapsto [y, N, z]\}$ and $cur = [x, N, N, x]$, there are two ways to execute the event, leading to two possible successor states with $cur = [x, y, N, z, N, x]$ and $cur = [x, N, y, N, z, x]$.

In this section we will focus on B [1] and TLA$^+$ [49], illustrated by the model checkers ProB [50] and Tlc [76].

3.1 Monolithic Approach: Directly Encoding the Semantics

One approach for model checking a high-level specification language is exhibited by the Tlc model checker. It directly encodes the operational semantics expressed in Java in the model checker; i.e., it follows the classical monolithic approach.

This leads to a quite efficient explicit state model checker (albeit slower than e.g. Spin) where library functions can be directly written in Java. Tlc can be parallelised, and can run in the cloud.

The disadvantage is that the model checker is really intertwined with the TLA$^+$ implementation and language and cannot be easily used for other languages, unless these are translated to TLA$^+$. Tlc also cannot perform constraint solving, meaning that the above rewrite specification cannot be handled. Such specifications have to be re-written manually, so that left-to-right evaluation leads to finite and reasonably efficient enumeration.

3.2 Prolog as an Intermediate Verification Language

From its conception, the animator and model checker ProB was designed to target multiple specification languages and to use Prolog as a pivot language, or more precisely as an intermediate verification language (cf. Sect. 4.1) for specifying language semantics. As such, the B semantics (or rather a superset thereof, denoted by B+ in Fig. 3, which is a pivot language in itself) is expressed using a Prolog interpreter, which specifies the set of initial states, the successor relation and the state properties.

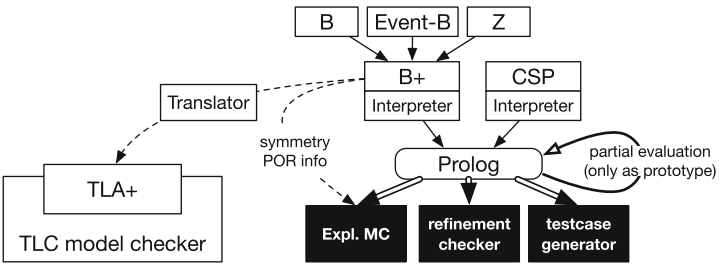


Fig. 3. The Tlc and ProB model checkers

This approach has a few advantages. It is easy to use the tool for other languages by providing an interpreter (or compiler). This is helped by the fact that Prolog, aka logic programming, is quite convenient to express the semantics of various programming and specification languages, in particular due to its support for non-determinism. E.g., the operational semantics rules of CSP [61] can be translated into Prolog clauses [51]. Furthermore, within Prolog one can make use of constraint logic programming for dealing with complex specifications, such as the grammar rewriting specification above. Finally, it is relatively straightforward to combine or integrate several formalisms, as was done for CSP ‖ B [17].

On the negative side, a Prolog interpreter will be slower (but easier to write) than a C or Java interpreter or even a compiler. Also, complex Prolog code such as the B interpreter of ProB, is not suited for analyses required for model checking optimisations, e.g., dependence information for partial order reduction or symmetry information. Within ProB such information is provided in an ad-hoc manner per supported language. Better solutions to this will be shown later, either by better pivot languages Sect. 4 or by the greybox approach Sect. 5.

Quite a few other tools also use Prolog as an intermediate input language. E.g, the XMC model checker [60] provides an explicit-state CTL model checker, targeting languages such as CCS via an interpreter or via translation. Techniques such as partial evaluation [52] and unfold-fold transformations [29] can be used for optimization, but also for a form of infinite state model checking. Finally, constraint programming techniques can be used for various validation tasks [23, 24]. Similarly, in Horn-Clause Verification, SMT solvers are applied to Prolog or Datalog like specifications [12,58].

3.3 Other High-Level Languages

There are many other high-level modelling languages. The languages VDM and ASM are very similar in style to B and TLA$^+$, and some translators between these languages exist. The process algebra CSP [61] also features sets and sequences as data types, but its operational semantics is quite different. The successful FDR tool [62] is to some extent a monolithic model checker (or more precisely refinement checker), even though it also performs an internal linearisation. CSP has also been a popular target for domain specific formalisms such as Casper [54] for security protocols or Circus [74]. For the latter there is also a recent translation to CSP‖B [75], and Circus itself is sometimes the target for other formalisms such as UML [16].

The toolset around mCRL, a process algebra with abstract datatypes, is based on an internal linearisation technique [34]. In the mCRL toolset, linear processes are viewed as an intermediate verification language (in the sense of Sect. 4.1). Due to their flattened form, they can be subjected to further optimization, and they are well-suited for adaptation to an on-the-fly API (in the sense of Sect. 5).

We would also like to mention the PAT model checker [53,77]. Its conception is similar to ProB but using C-Sharp instead of Prolog as an intermediate language.

Finally, instead of validating high-level specifications, it is also quite common to work directly with programming languages such as Java or C. The Java Pathfinder [35] tool translates a Java program to Promela for modelling with the Spin model checker [39]. Here, only certain aspects of the programming language are modelled (such as concurrency), abstracting away from other aspects. Another successful tool is CBMC [45] for bounded model checking, which provides bit-precise modelling for C and checking specific properties such as buffer overflows and exceptions. An alternative to model checking is abstract interpretation, such as used by the ASTREÉ analyzer [22] which has been successfully used for verification of C programs.

3.4 Summary

In summary, high-level logic-based languages are very popular for modelling and for expressing domain specific formalisms. We have shown how an intermediate pivot language like Prolog provides a good way to integrate formalisms, and allows a model checker to target a variety of dialects and related formalisms. The downside is performance: efficient model checking is very difficult to achieve in Prolog, and some information like symmetry and dependence for partial order reduction is difficult to extract from more involved Prolog representations. The approaches in the following sections will provide solutions to this. Section 4 provides other internal representations, while Sect. 5 presents a greybox API approach, which enables to connect a low-level model checking engine written in C with interpreters for higher-level languages. In Sect. 6.1 we will actually show how this has led to the latest generation model checking technique for B, by combining ProB's Prolog interpreter with LTSmin's model checking C engine.

4 Using an Intermediate Language as a Pivot

As discussed in Sect. 2, the role of an intermediate representation is to allow separate evolution of input languages with respect to model checking and verification algorithms. This section focuses on approaches reifying this pivot representation using an intermediate verification language (IVL). Section 4.1 presents the general approach, while Sect. 4.2 details a specific instance of an IVL called Guarded Action Language.

4.1 Intermediate Verification Language

An IVL is a language specifically designed to fit the role of pivot: rather than a language particularly comfortable for end users, it is designed as a general purpose input for a verification engine. The focus when designing an IVL is on providing a small set of semantic bricks while preserving good expressivity. The

end-user manipulates a user-friendly domain specific language (DSL) [73] that is translated into the IVL prior to the actual model checking or verification.

Historically, most model checkers were built in monolithic fashion, with a single supported input language and a single solution engine. This prevented a lot of reuse of actual code between model checkers, similar algorithms being reimplemented for each language. In this setting, to use a particular solver, you need to translate manually or automatically your specification to the solver's language.

For instance Promela the language of Spin [39] has often been used as a target for translation [15]. However it is a complex language with many semantic idiosyncrasies such as the support for embedded C code or the behavior attached to the *atomic* keyword. It also offers a wide variety of syntactic constructs, that make direct modeling in Promela comfortable for end-users. These features make life hard for a provider of a new algorithm or verification strategy. Because the language is complex, the development cost of supporting Promela in a tool is high. Many third-party tools for Promela analysis [42,71] only support a limited subset of Promela (typically excluding C code, and/or dynamic task creation).

IVL in the Literature. Hence while Promela has been used as an IVL it is not particularly well suited for that purpose, since it was not a design goal of the language. However many recent verification efforts include the definition of an intermediate languages, explicitly designed to be an intermediate verification language (e.g. [5,6,11,71]).

The SMV language [19] was designed to support symbolic verification (using either BDD or SAT based solvers) and serves as language based front-end to these technologies. The semantics is synchronous and thus well adapted to modeling of hardware components, but makes expression of asynchronous behaviors cumbersome.

In Sect. 3.2 we have already discussed the use of Prolog as a pivot language, and its limitations, e.g., related to partial order reduction or symmetry detection.

For program verification, the Boogie language (Microsoft) [6] is expressly designed as an intermediate language, helping to bridge the gap from programs to SMT based verification engines. Initially designed to support Spec#, i.e. annotated .Net bytecode, it has been extended to cover a host of programming languages using this intermediate language approach. All of these input languages thus benefit from improvements made to the verification engine (development of interpolants, new verification conditions,...).

The standard format SMT-lib [7] for SMT problems is itself a pivot intermediate language sharing many design goals with an IVL, but with a broader scope than the pivot languages considered in this paper.

Focusing more on concurrent semantics and finite state systems, the Guarded Action Language (GAL) [71] is an IVL that is supported by a decision diagram based symbolic verification engine. It helps bridge the gap between asynchronous and concurrent systems expressed in a variety of formalisms (Promela, Petri nets, timed automata,...) and a symbolic expression of their transition relation.

Section 4.2 presents the design choices we made when defining this language and the architecture of the ITS-tools model checker built around it.

Domain Specific Languages and Verification. This intermediate language approach integrates well with current model-based industrial practice. It helps solve two large stumbling blocks that prevent more widespread adoption of model checking. Firstly, due to automated translations, the end-user is isolated from ever needing to know about the specifics of how the verification is performed. This reduces adoption cost since training software engineers to build formal models is a difficult task, and helps achieve the "push-button" promise of automated verification. Secondly, the DSL models are developed with several purposes in mind, that typically include code generation or simulation. This means the models developed have precise behavioral semantics necessary for analysis, and also reduces the gap between what you prove correct (the formal model) and the running system. Provided the translations are correct and consistent with one another, the running system and the formal model both conform to the semantics of the DSL. Verification of the more abstract DSL is however usually easier than analyzing models extracted from actual implementations of the design.

Language Engineering. Language engineering using metamodeling technology has evolved rapidly over the last two decades, pushed by the OMG consortium and the development of the UML standard. Because UML is a particularly complex language, with a very broad scope, new technologies for model definition and manipulations were defined based on the concept of metamodel. These tools are now mature with industry strength quality (e.g. EMF [70]), and can be applied to a variety of models and languages that bear no relationship with UML.

In a model-centric approach, a metamodel is defined to describe a language, where models are instances of this metamodel. Because the metamodel is itself an instance of a metametamodel, common to all language definitions, powerful tools can be engineered that take a language (a metamodel) as input.

Tools such as XText [28] make development of new languages easier, with a full-blown modern end user experience (code completion, on the fly error detection...) available at a very low development cost.

Using model transformations to build formal models expressed in an IVL can thus be done using several alternative technological paths [27], and is well-understood by modern software engineers. This facilitates third-party adoption.

Technology Agnostic. The underlying verification engine is weakly constrained by an intermediate language approach. Model checking can use structural analysis, SAT technology, decision diagrams, explicit state... with solvers implemented in any programming language.

Because an IVL offers a complete view of the semantics to the analysis tools (in the absence of black-box behavior such as embedded code) it is still possible to write property specific abstractions such as slicing and simplifications such as constant removal. Such abstractions can usually be expressed as a transformation to a simpler model expressed in the same language. Hence all analysis tools

benefit from their existence. Section 5.3 will present how some of these issues can be addressed using a *greybox* API (e.g. to provide partial order reduction), but the abstractions that can be offered using an IVL are more powerful in general.

Modular Decomposition. Support for modular definition of a specification in the IVL is highly desirable. It helps support modular verification scenarios where only part of the system is analyzed to prove system-wide properties. This requires some weak hypothesis on how a component interacts with its environment to make compositional reasoning possible. The Mocha environment [4] uses such compositional reasoning, thanks to founding the semantics with reactive modules [3]. Other examples based on I/O automata [55], assume/guarantee contracts for components [59], or asynchronous composition such as in CADP [32] try to exploit compositional reasoning to provide simpler proofs.

4.2 GAL Within ITS-Tools

ITS-tools offers model checking (CTL, LTL) of large concurrent specifications expressed in a variety of formalisms: communicating process (Promela, DVE), timed specifications (Uppaal timed automata, time Petri nets) and high-level Petri nets. The tool is focused on verification of (large) globally asynchronous locally synchronous specifications. Its architectures is presented in Fig. 4.

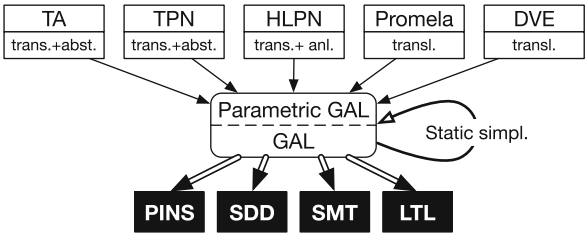


Fig. 4. Instantiation of the modular architecture for ITS-tools

It leverages model transformation technology to support model checking of domain specific languages (DSL). Models are transformed to the Guarded Action Language (GAL), a simple yet expressive language with finite Kripke structure semantics.

Guarded Action Language. GAL is a pivot language that essentially describes a generator for a labeled finite Kripke structure using a C like syntax. This simple yet expressive language makes no assumptions on the existence of high-level concepts such as processes or channels. While direct modeling in GAL is possible (and a rich eclipse based editor is provided), the language is mainly intended to be the target of a model transformation from a (high-level) language closer to the end-users.

A **GAL** model contains a set of integer variables and fixed size integer arrays defining its state, and a set of guarded transitions bearing a label chosen from a finite set. We use C 32 bit signed integer semantics, with overflow effects; this ensures all variables have a finite (if large 2^{32}) domain. GAL offers a rich signature consisting of all C operators for manipulation of the `int` and `boolean` data type and of arrays (including nested array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. In any state (i.e. an assignment of values to the variables and array cells of the GAL) a transition whose boolean guard predicate is true can fire executing the statements of its body in a single atomic step. The body of the transition is a sequence of statements, assigning new values to variables using an arithmetic expression on current variable values. A special $call(\lambda)$ statement allows to execute the body of any transition bearing label λ, modeling nondeterminism as a label based synchronization of behaviors.

Parametric GAL. specifications may contain parameters, that are defined over a finite range. These parameters can be used in transition definitions, compactly representing similar alternatives. They can also be used to define finite iterations (for loop), and as symbolic constants where appropriate. Parameters do not increase expressive power, the verification engine does not know about them, as specifications are instantiated before model checking. The tool applies rewriting strategies on parametric transitions before instantiation, in many cases avoiding the polynomial blowup in size resulting from a naive parameter instantiation. Rewriting rules that perform static simplifications (constant identification, slicing, abstraction...) of a GAL benefit all input formalisms.

Model to Model Transformations. Model-driven engineering (MDE) proposes to define domain specific languages (DSL), which contain a limited set of domain concepts [73]. This input is then transformed using model transformation technology to produce executable artifacts, tests, documentation or to perform specific validations. In this context GAL is designed as a convenient target formally expressing model semantics. We thus provide an EMF [70] compliant meta-model of GAL that can be used to leverage standard meta-modeling tools to write model to model transformations. This reduces the adoption cost of using formal validation as a step of the software engineering process.

Third-Party Support. We have implemented translations to GAL for several popular formalisms used by third party tools. We rely on XText for several of these: with this tool we define the grammar and meta-model of an existing formalisms, and it generates a rich code editor (context sensitive code completion, on the fly error detection,...) for the target language. For instance, we applied this approach to the Promela language of Spin [39] and the Timed Automata of Uppaal [9].

For Promela, channels are modeled as arrays, processes give rise to control variables that reflect the state they are in. A first analysis of Promela code is necessary to build the underlying control flow graph (giving an automaton for each process). There is currently no support for functions and the C fragment

of Promela. The support for TA and TPN uses discrete time assumptions, and will be detailed in Sect. 6.2.

Solution Engines. The main solution engine offered by ITS-tools is a symbolic model checker relying on state of the art decision diagram (DD) technology. A more recent addition is an SMT based encoding of GAL semantics, that enables a bounded model checking/induction decision procedure for safety properties. This SMT encoding also enables many static analysis tests such as computing interference between events that enable precise partial order reductions. A bridge from GAL to the PINS API (see Sect. 5) enables the many solution engines offered by LTSmin.

GAL thus successfully plays the pivot role of an intermediate verification language, allowing to separately choose the input language and the solution engine for verification. This approach is, however, not always applicable, e.g., when embedded code is associated with a model or when executing the high-level source language requires constraint solving not present in the intermediate language (cf., Sect. 3.2). The API approach presented in the next section is one solution for this problem.

5 The API Approach to Reusing Verification Engines

The focus in this section is on generic programming interfaces (API) between formal specification languages and model checking algorithms. The underlying wish is to reuse software implementations of model checking algorithms for specifications in different formal languages.

Semantically, the operational semantics of a specification language gives rise to a transition system, with labels on states or transitions, or both. Model checking algorithms can be viewed as graph algorithms over these transition systems. Many model checking algorithms operate *on-the-fly*, intertwining state space generation with analysis. In many cases, in particular when hunting for counter examples, only a fraction of the complete state space is visited. To facilitate this, the state space graph is often exposed to the algorithm through an API, providing the functionality to compute the desired part of the graph.

Black-Box API. Clearly, a black-box view on states and transitions would provide maximal genericity. Here states are opaque objects for the model checker and it just needs a function to retrieve the initial state, and another one to compute the next states of any given state. All information on the internal structure of states and transitions are nicely encapsulated in language modules specific to a formal language.

A prominent example of this approach is the Open/Caesar interface [30], which allows the CADP toolset to operate on input models in various process algebra-oriented languages, like Lotos, LNT, EXP and FSP. This facilitates the reuse of backend algorithms in CADP for model checking, bisimulation checking and reduction, test generation, simulation and visualisation. The Open/Caesar architecture also allowed to link external toolsets, for instance μCRL and LTSmin.

Greybox API, Pins. The disadvantage of a black-box API is that it prohibits many methods for mitigating the state space explosion. For instance, state space compression techniques, symbolic model checking and partial-order reduction require information on the structure of states and transitions. For this reason, the toolset LTSmin [14,42] introduced a *greybox API*, called Pins, the Partitioned Interface to the Next-State function, cf. Fig. 5. Here states are partitioned in vectors of N chunks, and the transition relation is partitioned into M subtransitions that operate on a *part* of the state vector. Depending on the specification language and the intended granularity, chunks can represent state variables, program counters, or subprocesses. Transitions could represent lines or blocks of code, or synchronized communication actions. Finally, the language frontend provides a static Dependency Matrix (DM) that declares which chunks in the state vector are affected by a certain transition group. Thus, locality of transitions is exposed to the model checking algorithms. See Table 1 for further details.

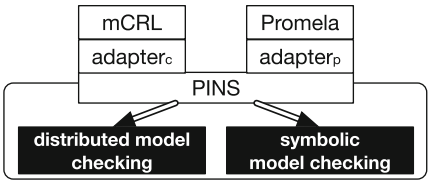


Fig. 5. Original instantiation of the modular architecture in LTSmin

Table 1. Parameters and functions of the PINS greybox API

N	Fixed length of the state vector
M	Number of disjunctive transition groups
init()	Function that returns the initial state vector
next(s, i, f)	Function that calls back f on any successor of s in transition group i
DM$[M][N]$	Dependency Matrix of Booleans: DM$[i][j]$ means transition group i depends on variable j

In the sequel, we demonstrate how gradually exposing more structure enables more and more model checking techniques to be applied, basically following the historical development of the LTSmin toolset.

5.1 Distributed and Multi-core Model Checking

In distributed model checking, it must be frequently tested whether a locally generated successor state already exists globally. This is usually solved by sending (batches of) states over the network to the machine that "owns" them.

Ultimately, the network bandwidth forms the performance bottleneck of this approach. In this section, we show how partitioning the state vector enables state space compression and leads to a reusable solution.

A distributed database and compression scheme was proposed for the μCRL toolset [13], which reduced the bandwidth to roughly two integers per state. That compression approach depends on (recursively) indexing the first and second half of a state vector, thus forming a binary tree of databases. The leaves of this tree consist of an index of algebraic data terms of μCRL. A piggy-backing scheme ensured global consistency of all databases.

The original motivation of the LTSmin toolset [14] was to offer this approach to multiple model checkers with their own specification languages, in particular to Promela models in SPIN. There were three considerations to combine these languages for distributed model checking: First, the interface had to support the action-based process algebra μCRL, as well as the state-based Promela models of SPIN. Also, besides the algebraic data-types of μCRL, it had to support the direct machine integer-representation of SPIN models. Finally, the database compression technique required access to various parts of a single state. These considerations led to the greybox Pins interface (Table 1), supporting both state and edge labels, and assuming and exposing a fixed-length state vector.

The separation provided by Pins turned out to be quite versatile for further experimentation: Initially, we conceived to link the MPI/C code of the distributed model checker directly to SPIN generated code, but this was deemed to be too fragile. The Pins interface allowed to switch freely to NIPS-VM, a virtual machine to interpret Promela models, and to SpinJa, a compiler for SPIN models. Actually, these experiments can be viewed as instances of combining a fixed API to various intermediate language representations in the spirit of Sect. 4.

Currently, LTSmin supports an arbitrary number of edge and state labels, allowing to handle for instance Mealy machines (input/output), probabilistic automata (actions/probabilities) and games (actions/players). By now, several more language modules have been constructed, enabling to reuse the same model checking algorithms for DVE (DiViNE), PetriNets (PNML), mCRL2, Timed Automata (Uppaal, cf. Sect. 6.2), B models (ProB, cf. Sect. 6.1), etc.

Finally, when we developed new multi-core algorithms, based on concurrent hash tables in shared memory and concurrent tree compression [48], the Pins interface allowed to effortlessly and directly carry out scalability experiments on benchmark models from this large variety of specification languages.

5.2 Symbolic BDD-Based Model Checking

The effectiveness of state compression can be explained from the locality of transitions, leading to the relative independence of the system components (e.g. processes). Binary Decision Diagrams (BDD) provide even more opportunities to compress sets of state vectors, by sharing common prefixes and suffixes. Can we gain more than just a concise representation? Here we want to emphasize that by exposing transition locality explicitly, we can also achieve computations

on sets of states. That is, we obtain the benefits of traditional symbolic model checking for models that are only provided through an on-the-fly API, without requiring a symbolic specification of the transition relation.

The main idea is that the static dependency matrix DM provided by Pins allows to deduce much information from one next-state call, in particular when the dependency-matrix is sparse (i.e., there is a lot of locality). Consider a state vector $x_0, \ldots, x_n$ in which a transition group t_k is enabled, that only affects $x_0, \ldots, x_i$, according to the DM. Then we can deduce the following two facts:

- All successors are of the form $x'_0, \ldots, x'_i, x_{i+1}, \ldots, x_n$
- All states of the form $x_0, \ldots, x_i, y_{i+1}, \ldots, y_n$ have successors from transition group t_k of the form $x'_0, \ldots, x'_i, y_{i+1}, \ldots, y_n$.

The short pair $x_0, \ldots, x_i \mapsto x'_0, \ldots, x'_i$ can be stored in a local BDD R_k and reused in relational product computations during further state space generation.

So, the Pins interface allows full-fledged symbolic model checking for explicit-state specification languages (Promela, mCRL2, DVE, ProB, etc.) without the need for manual symbolic encodings or automated model translations. The price to pay is that every language module should define transition groups at some level of granularity, and perform some kind of static analysis to identify the dependencies on state variables. Rough overapproximations of the dependency matrix are still correct, but more precise analyses expose more locality. This effort has to be performed for every specification language only once, and a precise analysis is rewarded by a more efficient model checker for that language.

Again, the Pins architecture proved to be very flexible, allowing experiments with among others Multiway Decision Diagrams, List Decision Diagrams, and also scalable multi-core implementations of decision diagrams [25] on a wide variety of benchmark models in many specification languages.

Another lesson learnt was that exposing more information leads to more efficient model checking. This seems obvious, but the sweet spot is not clear. In [57] we experimented with splitting transition groups in *guards* and *updates*, refining the Dependency Matrix to distinguish *read-* from *write*-dependencies. This led to considerable performance gains in symbolic model checking.

Note that existing language modules wouldn't profit from this refinement, but at least they don't break. Implementing the refined analysis for some specification language is incentivized by a more efficient model checking procedure.

5.3 Other Extensions as Pins2Pins Wrappers

So far we showed that the Pins-API allows combining multiple model checking algorithms with multiple specification languages, increasing the efficiency for the whole research community. We can take this one step further: a single state space optimization technique could be reused for *any* model checking algorithm and *any* specification language. This is supported by rewiring, using so-called Pins2Pins-wrappers, as in Fig. 6, which remotely resemble Unix-pipes: The original model is available on-the-fly to the Pins2Pins wrapper, which in

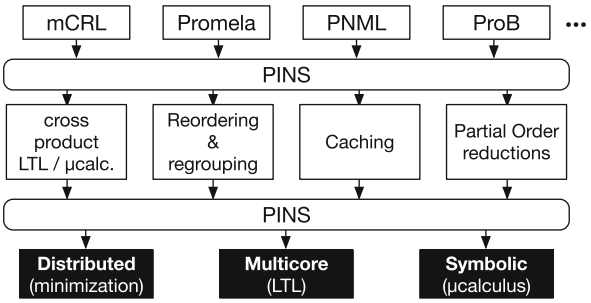


Fig. 6. On-the-fly state space transformers provided as PINS2PINS-wrappers in LTSmin

turns provides the reduced state space in an on-the-fly manner to the actual model checker. In reality, this involves a quite complicated rewiring of the call-back mechanism.

We will discuss a couple of instances. A simple instance is transition-caching: For highly expressive specification languages the next-state calculation will be slow. In case of high locality (sparse Dependency Matrix), it could pay off to have an intermediate caching layer that stores the results of all next-state calls for future reuse. This cache has to be implemented once, and can be reused for all models in all supported specification languages and for all model checking algorithms. (Note that this is not helpful for the symbolic model checker, since it already stores these transitions in the local BDDs R_k.)

A second example is reordering state variables and regrouping similar transition groups. It is well-known that the variable order greatly influences the efficiency of symbolic model checking. We investigated if the information from the read-write Dependency Matrix is sufficient to compute a good static variable order. The PINS interface with its DM allowed to apply many bandwidth reduction algorithms on matrices out-of-the box, and enabled us to compare them experimentally across multiple specification languages and multiple decision diagram types [25]. At the same time, we noticed that having too many transition groups leads to a considerable overhead. So the regrouping layer also recombines several transition groups that indicate the same (or similar) dependencies.

A third example is the computation of cross-products. For LTL model checking, the cross-product with a Büchi automaton is conveniently provided as a PINS2PINS-wrapper. For μ-calculus model checking, another PINS2PINS-wrapper computes the product of an LTS and a Boolean Equation System, resulting in a Parity Game (using the fact that LTSmin supports multiple edge labels to encode players and priorities). A more generic product automata wrapper, that could support compositional model checking, is under construction.

Finally, we shortly discuss some experiments with partial-order reduction. We investigated if the DM contains sufficient information to implement state space reduction based on the stubborn-set approach to POR [72]. The bad news

is that the achieved reductions would be suboptimal: from the DM it can only be deduced that two subtransitions are independent (e.g. t_k doesn't modify variables that t_ℓ reads or writes). However, to achieve the full effectiveness of POR we had to extend the DM with new matrices, basically indicating whether transition groups can enable each other. More precisely, one can exploit refined guard splitting: A new matrix indicates whether executing transition group t_ℓ could enable or disable guard g_i.

The good news is that extending PINS with information on enabling/disabling relations between transition groups, allows the full reduction power of stubborn-set POR method [47]. LTSmin comes up with a reasonable default for the new POR-related matrices. Language modules that take the effort to derive more precise transition dependencies are again rewarded with more effective state space reduction power. Thus, a partial-order reduction block can be provided, which is suitable for all specification language modules implementing Pins and potentially supports all model checking algorithms based on the Pins interface.

One may wonder if this provides effective partial-order reduction for symbolic model checking? Unfortunately, after partial order reduction all dependency information is lost, so symbolic model checking on the reduced state space would be correct, but not effective. Similarly, in the case of timed automata, all transitions involve manipulating the clocks, so partial-order reduction of TA is correct, but not effective. Positive cases, where POR is effective, are the explicit multi-core model checking algorithms, both for safety and LTL properties, applied to mCRL2, Promela, DVE, PNML, or B models.

6 Application Examples

This section shows how the variants of the modular approach (library-based or language-based) can be instantiated in real situations.

6.1 ProB to LTSmin API: Linking High-Level Languages with Other Model Checkers

In [10] we have presented a first integration of ProB with LTSmin. We thereby managed to keep the full power of the constraint solving of ProB's Prolog interpreter to compute successor states for complicated events (see Sect. 3). But we also gained access to the symbolic model checking engine of LTSmin, to construct a symbolic BDD-style representation of the reachable states. For some experiments, this resulted in the reduction of the model checking time of an order of magnitude or more. The crux lies in the fact that through the greybox API, LTSmin gains information about read/write dependencies of events, which is crucial to build up the symbolic representation of the state space. Note that ProB's representation of B's datastructures are hidden to LTSmin: LTSmin does not need to know about higher-order sets and relations, nor about symbolic representations for infinite B functions, just to mention a few possible data values.

All LTSmin needs to know is the variables of the B model and the read-write dependencies. For example, suppose we have a state $\langle x = 10, y = \{\{2\}, \{4\}\}\rangle$ (where the variable y is a set of sets) and the event *inc* produces the single successor state $\langle x = 11, y = \{\{2\}, \{4\}\}\rangle$. Given the information that *inc* reads and writes only x, LTSmin knows, without having to query ProB, that the only successor of $\langle x = 10, y = \{\{1, 2\}\}\rangle$ is $\langle x = 11, y = \{\{1, 2\}\}\rangle$.

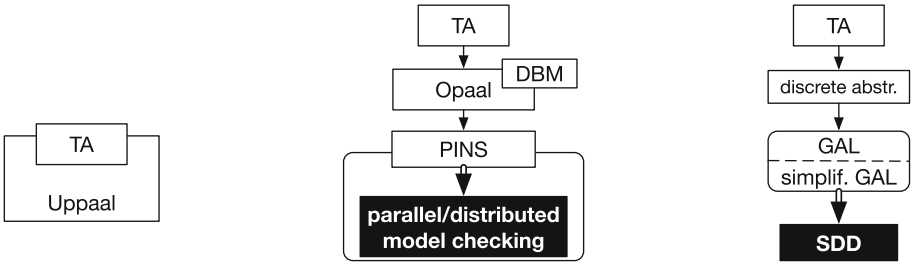
On a technical side, the communication was achieved by ZeroMQ. In ongoing work [44] the bridge has been extended to support partial order reduction and parallel LTL model checking, again with sometimes impressive speedups compared to ProB's internal explicit state model checker.

6.2 Analysis of Timed Automata

Uppaal's networks of timed automata are the de facto standard for the high-level specification of real time systems, with a well-integrated tool support in the Uppaal tool suite. For this reason, Uppaal is also used as a target of model transformation, as an IVL. Uppaal's efficient solver is based on zone based abstraction with subsumption. However, due to its tight integration, Uppaal uses a monolithic approach (Fig. 7a): all algorithms are tightly connected to Uppaal models, and not available as open source components, except the DBM library, which offers zone abstraction through Difference Bound Matrices.

We discuss two approaches to analyze Uppaal models using the API approach (linking Uppaal to LTSmin) or the IVL approach (translating Uppaal models to GAL specifications as in ITS-Tools).

LTSmin Approach. A bridge between Uppaal and LTSmin was devised, cf. Fig. 7b, which supports full multi-core LTL model checking of Uppaal networks of timed automata [46]. The advantage of this approach is that it maximizes code reuse. It uses opaal to generate C-code from Uppaal models, which was adapted to implement the Pins interface. Furthermore, the next-state function directly calls the DBM-library. For LTSmin's multi-core algorithms, a state vector just contains an opaque pointer to a DBM to represent a symbolic time zone. In



(a) Uppaal (b) LTSmin (c) ITS-tools

Fig. 7. Three architectures for TA model checkers.

this way, Uppaal users obtain a scalable multi-core LTL model checker in a transparent manner.

Two issues arise, however: First, timed automata based on timed zones have abstract states, which require subsumption for efficient state space reduction. This was solved by (again) extending the PINS interface with an extra function (to reuse the DBM-library for checking subsumption of symbolic states). Another issue is that time manipulation happens in every transition group, which leads to a dense dependency matrix. Hence symbolic model checking and partial-order reduction are not effective on timed automata.

ITS-Tools. The support for TA uses discrete time assumptions to be able to model the semantics using GAL, as in Fig. 7c. Fortunately, analysis in the discrete setting has been shown to be equivalent to analysis in a dense time setting provided all constraints in the automata are of the form $x \leq k$ but not $x < k$ [37]. We thus can build a transition that represents a one time unit delay and updates clocks appropriately. This transition is in fact a sequence of tests for each clock, checking if an urgent time constraint is reached (time cannot elapse), if the clock is active (increment its counter) or if it is inactive either because it will be reset before being read again, or because it has reached a value greater than any it could be tested against before a reset (do nothing). This test for inactive clocks corresponds to an abstraction that preserves observable behaviors, but prevents clock values from growing indefinitely, yielding an infinite state space.

Strengths. This discrete time approach is very effective to deal with systems where the number of concurrently enabled locations or clocks grows, since in such cases the classic explicit state with zones represented as DBM does not scale well. However, when the maximum bounds on clocks grow, even decision diagrams have trouble dealing with the state space explosion in the discrete setting. The two approaches thus have good complementarity, allowing to address different kinds of systems.

Weaknesses. Overall the main difficulty when developing support for timed automata is that the classical dense time semantics of TA cannot be feasibly encoded just using GAL which have discrete semantics. The correctness of switching to discrete semantics was fortunately already established [37], but in general mapping of arbitrary semantics to GAL is not always possible. It is much easier to map arbitrary semantics to a language such as Prolog (see Sect. 3.2) but this comes at the cost of verification power and efficiency. The discrete time models have a very large state space, and cannot feasibly be analyzed using non symbolic solution engines, so despite the pivot language approach, the choice of this path limits the choice of the solution engine. However explicit state approaches are of course still available on TA using the Uppaal verifier or LTSmin.

7 Discussion

This paper summarizes the evolution of modern model checking tools in terms of their architecture and usage to solve typical industrial-like problems, which

are more and more stated using high-level, domain specific languages instead of the "traditional" specification languages. Moreover, complementary techniques are often used to solve particular situations. For example, explicit techniques may scale less but algorithms to compute counter-examples are simpler. On the contrary, symbolic techniques usually scale better but computation of a counter example is not trivial.

To cope with such situations, an intermediate level has been introduced, the pivot representation, which provides a modular architecture to link high-level specifications with a backend for verification. This pivot representation can be either a library offering an API, or a language itself. Both approaches co-exist today and show their own advantages and drawbacks. This is of interest to enable transparent activation of a given technique. This can be seen as a configuration issue (choice of a technique/algorithm at a given stage of the verification process) or to some preprocessing phase. It is thus particularly important to benefit from a large portfolio of techniques available in a given environment, and linked to the pivot representation. Such a situation is observed in the Model Checking Contest [43] where some tools concurrently operate several techniques and retrieve results from the first that finishes. Let us also refer to CIL that is used as an entry in the software competition to operate various tools and techniques (or PNML that has a similar role in the model checking contest).

In both approaches, the problem of translating counter examples back to the user level exists. Due to their tight integration, monolithic architectures can also offer an integrated user experience, which can be viewed as an advantage. For modular approaches, it takes some effort to link between transformations (this is typically supported by MDE-based approaches) but this is more difficult when there are several translations (*e.g.* optimization phases).

Embedding external code is possible within the API-approach, as long as one respects the absence of side effects. This is an attractive feature for verifying systems using complex data structures or libraries. The API approach also allows to reuse existing implementations of the operational semantics of specification languages. As a consequence, the transition relation is more opaque, isolating verification algorithms from the actual representation. This possibly disables some abstraction opportunities. We demonstrated how greybox API solutions (PINS) disclose sufficient structural information to enable some important optimizations, like state compression, partial-order reduction and decision-diagram representations.

The use of a reified intermediate verification language as a pivot preserves the semantics completely. This means that more solution engines remain available, such as SMT solvers for encoding data abstractions. The translation needs to be complete and true to the original semantics. This requires more effort, which may be hard or impossible in some cases due to the semantic gaps in expressivity when source and target languages differ too much (*i.e.*, the translation of B to SMV in fact generates the full state space during the translation). Potential loss of the modular structure of specifications during the translation could also be

a drawback, since this structure contains useful information that can in general be exploited by model checking heuristics.

The field of model checking is moving fast: new algorithms, improved data structures and high-performance implementations are emerging all the time. This also leads to new application domains, with their own domain-specific, high-level modeling languages. Within these dynamics, intermediate representations provide some stability, by decoupling verification algorithms from high-level specifications. This paper presents a decade of research in sophisticated intermediate representations, either as intermediate verification languages, or as on-the-fly greybox interfaces. Despite an improved understanding of the relationship between verification capabilities and features of the intermediate representation, a "golden standard API", or a "holy grail IVL" has not yet emerged. On the contrary, the building blocks of model checking architectures are also still in development. Fortunately, the approaches that we have reported combine very well from a methodological point of view. Several language translations and optimisations can be composed; the resulting "flattened" specification is easily adapted to an API; and several building blocks can be combined through plug-and-play with the API. We demonstrated that through these successful combinations one can obtain very efficient model checkers for high-level specification languages.

Acknowledgements. We would like to thank the many people who have worked on the various verification tools such as LTSMin and ProB. In particular, we want to thank Jens Bendisposto, Philipp Körner, Jeroen Meijer, Helen Treharne, Jorden Whitefield for their work on the ProB to LTSmin API described in Sect. 6.1.

We also thank Stefan Blom, Michael Weber, Elwin Pater for setting up the architecture of LTSmin and Alfons Laarman, Tom van Dijk, Jeroen Meijer for recent developments on multicore and symbolic LTSmin.

On the ProB side we are grateful to many researchers and developers who have contributed to the tool or its underlying techniques, notably Michael Butler, Joy Clark, Ivaylo Dobrikov, Marc Fontaine, Fabian Fritz, Dominik Hansen, Sebastian Krings, Thierry Massart, Daniel Plagge, David Schneider, Joshua Schmidt, Corinna Spermann.

We finally thank the many colleagues who contributed to the development and algorithms for ITS-Tools, in particular, Béatrice Bérard, Denis Poitrenaud, Maximilien Colange, Yann Ben Maïssa, and many master students.

The third author has been partially funded from the 4TU NIRICT.BSR project on Big Software on the Run.

References

1. Abrial, J.R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst. Des. **15**(1), 7–48 (1999)
4. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: modularity in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028774

5. Arnold, A., Point, G., Griffault, A., Rauzy, A.: The altarica formalism for describing concurrent systems. Fundam. Inform. **40**(2–3), 109–124 (1999)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
7. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th IW on Satisfiability Modulo Theories, Edinburgh, UK (2010)
9. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST, pp. 125–126. IEEE Computer Society (2006)
10. Bendisposto, J., Körner, P., Leuschel, M., Meijer, J., van de Pol, J., Treharne, H., Whitefield, J.: Symbolic reachability analysis of B through ProB and LTSmin. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 275–291. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_18
11. Berthomieux, B., Bodeveix, J.P., Filali, M., Lang, F., Le Botland, D., Vernadat, F.: The syntax and semantic of fiacre. Technical report 7264, CNRS-LAAS (2007)
12. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
13. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A database approach to distributed state-space generation. J. Log. Comput. **21**(1), 45–62 (2011)
14. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_31
15. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Auton. Agent. Multi-Agent Syst. **12**(2), 239–256 (2006)
16. Borges, R.M., Mota, A.C.: Integrating UML and formal methods. Electron. Notes Theor. Comput. Sci. **184**, 97–112 (2007). 2nd Brazilian Symposium on Formal Methods (SBMF 2005)
17. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_16
18. Butler, M.J., Colley, J., Edmunds, A., Snook, C.F., Evans, N., Grant, N., Marshall, H.: Modelling and refinement in CODA. In: Derrick, J., Boiten, E.A., Reeves, S. (eds.) Proceedings 16th International Refinement Workshop, Refine@IFM 2013, Turku, Finland, 11 June 2013. EPTCS, vol. 115, pp. 36–51 (2013)
19. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
20. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging (turing award 2007). Commun. ACM **52**(11), 74–84 (2009)

21. Correa, T., Becker, L.B., Farines, J., Bodeveix, J., Filali, M., Vernadat, F.: Supporting the design of safety critical systems using AADL. In: 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, pp. 331–336. IEEE Computer Society (2010)
22. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3
23. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_16
24. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. STTT **3**(3), 250–270 (2001)
25. van Dijk, T., van de Pol, J.: Sylvan: multi-core decision diagrams. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 677–691. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_60
26. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
27. Eclipse Project: Model-to-Model Transformation MMT, subproject of Eclipse Modeling (2017). https://projects.eclipse.org/projects/modeling.mmt
28. Efftinge, S., et al.: XText (2017). http://www.eclipse.org/Xtext/
29. Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In: Proceedings of VCL 2001, Florence, Italy, September 2001
30. Garavel, H.: OPEN/CÆSAR: an open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054165
31. Garavel, H.: Nested-unit petri nets: a structural means to increase efficiency and scalability of verification on elementary nets. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 179–199. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19488-2_9
32. Garavel, H., Lang, F., Mateescu, R.: Compositional verification of asynchronous concurrent systems using CADP. Acta Inf. **52**(4–5), 337–392 (2015)
33. Girault, C., Valk, R.: Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-662-05324-9
34. Groote, J.F., Ponse, A., Usenko, Y.S.: Linearization in parallel pcrl. J. Log. Algebr. Program. **48**(1–2), 39–70 (2001)
35. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000). https://doi.org/10.1007/s100090050043
36. Hayes, I., Jones, C.B.: Specifications are not (necessarily) executable. Softw. Eng. J. **4**(6), 330–338 (1989)
37. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55719-9_103
38. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
39. Holzmann, G.: Spin Model Checker, The: Primer and Reference Manual. Addison-Wesley Professional, Boston (2003)

40. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Boston (1979)
41. Iliasov, A., Lopatkin, I., Romanovsky, A.: The SafeCap platform for modelling railway safety and capacity. In: Bitsch, F., Guiochet, J., Kaâniche, M. (eds.) SAFECOMP 2013. LNCS, vol. 8153, pp. 130–137. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40793-2_12
42. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
43. Kordon, F., Garavel, H., Hillah, L.M., Paviot-Adet, E., Jezequel, L., Rodríguez, C., Hulin-Hubard, F.: MCC'2015 – the fifth model checking contest. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 262–273. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_12
44. Körner, P.: An integration of ProB and LTSmin. Master's thesis, Universität Düsseldorf, February 2017
45. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
46. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 968–983. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
47. Laarman, A., Pater, E., van de Pol, J., Hansen, H.: Guard-based partial-order reduction. STTT **18**(4), 427–448 (2016)
48. Laarman, A., van de Pol, J., Weber, M.: Multi-core LTSmin: marrying modularity and scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_40
49. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
50. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
51. Leuschel, M., Fontaine, M.: Probing the depths of CSP-M: a new FDR-compliant validation tool. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88194-0_18
52. Leuschel, M., Massart, T.: Infinite state model checking by abstract interpretation and program specialisation. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 62–81. Springer, Heidelberg (2000). https://doi.org/10.1007/10720327_5
53. Liu, Y., Sun, J., Dong, J.S.: PAT 3: an extensible architecture for building multi-domain model checkers. In: IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, 29 November–2 December 2011, pp. 190–199 (2011)
54. Lowe, G.: Casper: a compiler for the analysis of security protocols. J. Comput. Secur. **6**(1–2), 53–84 (1998)
55. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC, pp. 137–151. ACM (1987)

56. Margaria, T., Nagel, R., Steffen, B.: jETI: a tool for remote tool integration. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 557–562. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_38
57. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 204–219. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_16
58. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: a practical approach. Formal Asp. Comput. **20**(4–5), 481–505 (2008)
59. Păsăreanu, C.S., Dwyer, M.B., Huth, M.: Assume-guarantee model checking of software: a comparative case study. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 168–183. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48234-2_14
60. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.N.: XMC: a logic-programming-based verification toolset. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 576–580. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_48
61. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1999)
62. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_7
63. Samia, M., Wiegard, H., Bendisposto, J., Leuschel, M.: High-level versus low-level specifications: comparing B with Promela and ProB with spin. In: Proceedings TFM-B 2009, pp. 49–61. APCB, June 2009
64. Schröter, C., Schwoon, S., Esparza, J.: The model-checking kit. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 463–472. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44919-1_29
65. Snook, C., Butler, M.: UML-B: a plug-in for the Event-B tool set. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, p. 344. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_32
66. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Upper Saddle River (1992)
67. Stefanescu, A., Wieczorek, S., Schur, M.: Message choreography modeling. Softw. Syst. Model. **13**(1), 9–33 (2014)
68. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 72–87. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60218-6_6
69. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: concepts and design. STTT **1**(1–2), 9–30 (1997)
70. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, Boston (2009)
71. Thierry-Mieg, Y.: Symbolic model-checking using ITS-tools. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 231–237. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_20
72. Valmari, A.: A stubborn attack on state explosion. Formal Methods Syst. Des. **1**(4), 297–322 (1992)
73. Voelter, M., et al.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages (2013). dslbook.org

74. Woodcock, J., Cavalcanti, A., Freitas, L.: Operational semantics for model checking Circus. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 237–252. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_17
75. Ye, K., Woodcock, J.: Model checking of state-rich formalism *Circus* by linking to CSP ‖ B. STTT **19**(1), 73–96 (2017)
76. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA^+ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6
77. Zhu, H., Sun, J., Dong, J.S., Lin, S.: From verified model to executable program: the PAT approach. ISSE **12**(1), 1–26 (2016)