Bridging the Gap between Enumerative and Symbolic Model Checkers

Stefan Blom, Jaco van de Pol and Michael Weber
Formal Methods and Tools, University of Twente, The Netherlands
{sccblom,vdpol,michaelw}@cs.utwente.nl

***Abstract*—We present a method to perform symbolic state space generation for languages with existing enumerative state generators. The method is largely independent from the chosen modelling language. We validated this on three different types of languages and tools: state-based languages (PROMELA), action-based process algebras (μCRL, mCRL2), and discrete abstractions of ODEs (Maple). Only little information about the combinatorial structure of the underlying model checking problem needs to be provided. The key enabling data structure is the "PINS" dependency matrix. Moreover, it can be provided gradually (more precise information yields better results). Second, in addition to symbolic reachability, the same PINS matrix contains enough information to enable new optimizations in state space generation (local transition caching), again independent from the chosen modelling language. We have also based existing optimizations, like (recursive) state collapsing, on top of PINS and hint at how to support partial order reduction techniques. Third, PINS allows interfacing of existing state generators to, e.g., distributed reachability tools. Thus, besides the stated novelties, the method we propose also significantly reduces the complexity of building modular yet still efficient model checking tools. Our experiments show that we can match or even outperform existing tools by reusing their own state generators, which we have linked into an implementation of our ideas.**

I. INTRODUCTION

In model checking, analysis algorithms are applied to large graphs, which model the behavior of (computer) systems. These models are typically generated from specifications in high-level languages. A major problem is that the generated models are much larger than their concise specification. For concurrent systems, model checking is a truly combinatorial problem.

Hence, over the last two decades considerable algorithm engineering effort has been invested in model checking tools. We mention on-the-fly, symbolic, compositional, and distributed model checking, partial-order and symmetry reduction, and also hashing and compression techniques. These techniques are in principle independent of the specification language. Rather, their effectiveness is based on the combinatorial structure present in concurrent systems. In particular, transitions are mostly local to one or a few parallel components.

This research has been partially funded by the EU under grant number FP6-NEST STREP 043235 (EC-MOAN).

This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.05N09.

Technical Report **TR-CTIT-09-30**, 05 June 2009, revised 15 July 2009, University of Twente, P.O.Box 217, 7500AE Enschede, The Netherlands

However, concrete realizations of those techniques are often tightly bound to specific languages and tools. In this work, we present a simple, but apparently overlooked, way to capture how transitions depend on specific parts of a system state. This proposed dependency matrix can be exploited through a small, but powerful, extension of the so-called *next-state* interface which is common in the world of enumerative model checking. For example, it has been underlying the success of the CADP toolkit: the OPEN/CÆSAR implementation [1] allows to connect a spectrum of specification languages to the functionality of various enumerative on-the-fly model checking and state space reduction algorithms.

Contribution. The information provided in the aforementioned dependency matrix allows us with little effort to retrofit methods for *symbolic* state space exploration onto existing *enumerative* state generators. Without having to change the internals of either tool, we were able to integrate them with existing *binary decision diagram* (BDD) packages.

We propose an extension of the traditional *next-state* interface to take advantage of the dependency matrix. We call it PINS, an **I**nterface based on a ***P**artitioned* **N**ext-**S**tate function. In a nutshell, a state for PINS is a vector of N slots, where a single slot can represent anything. The transition relation is split disjunctively into K groups. The $N \times K$ PINS dependency matrix then denotes which slots each group depends on.

We have implemented PINS[1] language modules for the state-based language PROMELA (via the NIPSVM), a process algebra with recursive data types (μCRL/mCRL2), and as a rather exotic application, for (abstractions of) *ordinary differential equations* (ODEs) in MAPLE, to analyse biological systems.

Moreover, on top of the PINS matrix we have implemented *tree compression* (which improves memory footprint and communication bandwidth) and *local transition caching* (which avoids redundant computations). These building blocks are implemented once, but all combinations of specification languages and analysis tools can benefit (Fig. 1).

We currently support both symbolic and distributed enumerative model checking. Although nothing prevents the use of full model checking, in this paper we have limited ourselves to reachability problems.

Related work is discussed in Sec. II. Subsequently, we introduce the PINS matrix and the interface to existing tools

[1] Available in the LTSMIN toolset v1.2, http://fmt.cs.utwente.nl/tools/ltsmin/

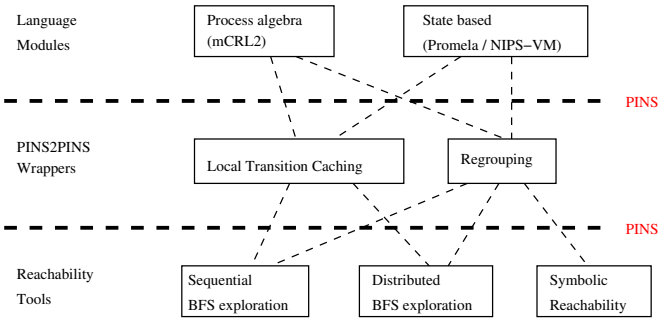

Fig. 1. Architectural Overview of PINS-Based Tools

(Sec. III). We discuss the language modules (Sec. IV) and analysis tools (Sec. V) that we have realized. Finally, we report on experiments (Sec. VI) that we carried out to measure the efficacy of our method and to demonstrate the benefits of having multiple analysis tools for multiple languages. As side effect, it becomes much easier to cross-compare model checking techniques on benchmarks from different communities without discriminating against specialized tools. We believe that this is of scientific interest.

The clear winners are the end users of model checking tools: they can keep using their preferred environment for modeling and validation. In practice, it is not always clear beforehand which analysis technique or language is best suited for the problem, because the effects of various heuristics are not predictable. A pleasant side-effect of the work presented here is that the choice of model checking method (enumerative, symbolic or distributed) becomes finally independent from the models.

II. Related Work

As discussed in the introduction, we view our work as a successor of the work of Garavel on Open/Cæsar [1], which provides a monolithic *next-state* interface. While keeping the separation between language modules and analysis tools, our partitioned next-state interface reveals more state and transition structure, which can be exploited effectively by symbolic and distributed analysis techniques.

The on-the-fly approach should be contrasted to other attempts to make model checking tools interoperable. Many of those approaches are based on language translations. A notable instance of this approach is the IF intermediate format [2], with tools like UML2IF and IF2PROMELA. Also the interoperability and remote integration provided by the jETI-platform [3] is ultimately based on linking various tools via translators between specifications or file formats. Another approach, directed towards cross-comparing tools, is BEEM [4]. This database of benchmark models is based on translations between Promela, DVE and mCRL2. We stress that our approach does not require language translations, but we link directly to the corresponding *native tools*.

We build upon previous work in symbolic and distributed model checking, reviewed below. An alternative approach is Exp.Open 2.0 [5], which combines partial-order reduction, compositional, and on-the-fly verification. Here networks of communicating automata are specified in the EXP specification language, while individual automata are treated as black boxes. Technically, EXP relies on *synchronization vectors*, which could be usefully connected with Pins.

In symbolic model checking [6], using sub-transitions is similar to disjunctive partitioning [7]. Symbolic model checkers, such as NuSMV [8], require Boolean encodings. This is complicated in the presence of recursive data types, or dynamically changing systems. We are inspired by the work on SMART by Ciardo et al. [9], where multi-way decision diagrams are used to handle growing data domains. In a sense, the actual encoding is only computed during reachability analysis. We have extended the SMART approach to non-deterministic transitions [10]. Pins improves on the interface introduced there by adding an implementation for Promela, incorporating tree compression for distributed model checking, and a new notion of transition caching.

Previous attempts to apply symbolic methods to model checking Promela specifications were described by Visser and Barringer [11] and Gregoire [12]. The representation of the visited states by a hash table was replaced by symbolic data structures (OBDDs and GETSs, respectively). Memory is conserved, but in general the time to perform a full reachability analysis increases because still all states are visited explicitly, one by one. In addition to the memory savings, our approach also *symbolically* computes the set of states reachable in the next level. As a consequence, we can report considerable speedups in some cases. Yet another approach is p2b [13], which translates Promela directly to SMV. Our approach does not involve any translations at the level of specification languages; we rely on an on-the-fly interface. Rather than using SPIN [14] for computing the operational semantics we have used NipsVM [15], because it was easier to interface with.

The scheme of our distributed tool closely follows the traditional approach [16], [17]. States are assigned to workers according to a static hash function. When a worker computes successor states, it must send them to their owning workers over the network. Extensions to this scheme which Blom et al. proposed in earlier work [18] are compatible with Pins. In fact, the proposed *tree compression* also depends on the extra state structure revealed by Pins.

III. Pins: Partitioning the Next-State Function

In this section we explain the partitioned interface for next-state functions (Pins) and its use with the pins matrix. Pins shares an important principle with the (monolithic) Open/Cæsar interface: it separates language specific aspects from generic model checking functionality. However, we expose more information with Pins in order to support both symbolic model checking and distributed model checking.

The underlying semantical model of both the monolithic and the partitioned interface is the labeled transition system. We elide state labels and edge labels (such as atomic propositions and actions labels) for presentational reasons.

```
int x=1;                          | int y=1;                          | int z=1;
process p1 (){                    | process p2 (){                    | process p3 (){
  do                              |   do                              |   do
  :: atomic{ x>0 -> x--; y++}     |   :: atomic{ y>0 -> y--; x++}     |   :: atomic{ z>0 -> z--; x++}
  :: atomic{ x>0 -> x--; z++}     |   :: atomic{ y>0 -> y--; z++}     |   :: atomic{ z>0 -> z--; y++}
  od }                            |   od }                            |   od }
```

Fig. 2. Example of a parallel system in PROMELA

Definition 1. *A transition system (TS) is a structure* $\langle S, \rightarrow, s^0 \rangle$*, where* S *is a set of states,* $\rightarrow \subseteq S \times S$ *is a transition relation and* $s^0 \in S$ *is the initial state.*

The monolithic *next-state* interface has two basic functions: INITIAL-STATE() returns the initial state s^0; NEXT-STATE(s) returns the successor states of state s. This interface allows explicit-state (enumerative) algorithms to be implemented for any language, but it has very limited use in combination with symbolic techniques due to the fact that we need to call the next-state function for every state.

To be more widely usable with symbolic techniques, we exploit what is known as *event locality*. For example, consider a system consisting of several (asynchronous) processes that communicate using a shared variable. The behaviour of such a system is determined by the behaviour of the processes. And although the behaviour of the system depends on the whole state, the behaviour of each process depends on the local state of the process and the shared variable only.

We can formalize this by assuming that the set of states is a *Cartesian product* and furthermore that the transition relation is the union of a few *transition groups*.

Definition 2. *A partitioned transition system (PTS) is a structure* $\mathcal{P} = \langle \langle S_1, \ldots, S_N \rangle, \langle \rightarrow_1, \ldots, \rightarrow_K \rangle, \langle s_1^0, \ldots, s_N^0 \rangle \rangle$*. The sets of elements* $S_1, \ldots, S_N$ *define the set of states* $S_{\mathcal{P}} = S_1 \times \cdots \times S_N$*. The transition groups* $\rightarrow_i \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$*,* $(1 \leq i \leq K)$ *define the transition relation* $\rightarrow = \bigcup_{i=1}^{K} \rightarrow_i$*. The initial state is* $s^0 := \langle s_1^0, \ldots, s_N^0 \rangle \in S_{\mathcal{P}}$*. The defined TS of* $\mathcal{P}$ *is* $\langle S_{\mathcal{P}}, \rightarrow, s^0 \rangle$*.*

Thus, a state $s \in \langle S_1, \ldots, S_N \rangle$ is really a vector consisting of N *slots*. In a partitioned TS, a transition group is independent of slot j if none of the transitions in the group can change the value of slot j and any transition in the group is enabled or disabled, regardless of the value of slot j. Formally, this can be stated as follows:

Definition 3. *Given a PTS* $\mathcal{P} = \langle \langle S_1, \ldots, S_N \rangle, \langle \rightarrow_1, \ldots, \rightarrow_K \rangle, \langle s_1^0, \ldots, s_N^0 \rangle \rangle$*. Transition group* i *is independent of state slot* j *if for all* $\langle s_1, \ldots, s_N \rangle$ *and* $\langle t_1, \ldots, t_N \rangle \in S_{\mathcal{P}}$*, whenever* $\langle s_1, \ldots, s_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots t_N \rangle$*, then*

1) $s_j = t_j$ *(i.e., state slot* j *is not modified in transition* i*)*
2) *for all* $r_j \in S_j$*, we also have* $\langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, r_j, \ldots, t_N \rangle$*. (I.e., the value of state slot* j *is not relevant in transition* i*.)*

If the dependencies are known they can be put in a matrix. In general they will not be known in advance. However, we can approximate them with static analysis. Thus, we define a dependency matrix as an approximation of the dependency relation. Given a transition group, the matrix allows us to *project* any state on a subvector that it may depend on.

$$\begin{array}{c} \\ p1 \\ p2 \\ p3 \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

(a)

$$\begin{array}{c} \\ p1.1 \\ p1.2 \\ p2.1 \\ p2.2 \\ p3.1 \\ p3.2 \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \end{array}$$

(b)

Fig. 3. Two valid dependency matrices for the example in Fig. 2: (a) "worst-case" matrix, and (b) a more precise matrix obtained by splitting transition groups.

Definition 4 (PINS Matrix)**.** *A dependency matrix* $D_{K \times N} = DM(\mathcal{P})$ *for PTS* $\mathcal{P}$ *is a matrix with* K *rows and* N *columns containing* $\{0, 1\}$ *such that if* $D_{i,j} = 0$ *then group* i *is independent of element* j*.*
For any transition group $1 \leq i \leq K$*, we define* π_i *as the projection* $\pi_i : S \rightarrow \Pi_{\{1 \leq j \leq N | D_{i,j}=1\}} S_j$*.*

The PINS interface exposes the dependency matrix to both, language modules and analysis algorithms. Thus, we add function GET-MATRIX() := $D_{K \times N}$, and adapt the signature of NEXT-STATE to take advantage of it:

$$\text{NEXT-STATE}(i, s) = \{s' \mid s \rightarrow_i s'\}$$

A small example of a parallel system is given in Fig. 2. The state consists of three variables x, y and z. There are three processes. Each process tries to decrease one of the variables and increase one of the others. A natural way of partitioning the state is to partition it as a vector of length three: $\langle x, y, z \rangle$. There are two natural ways of partitioning the transitions into groups. The first way is to introduce a group for every process. This would lead to three groups. The second option is to introduce a group for every atomic statement. This would result in six groups. The resulting dependency matrices are shown in Fig. 3. Matrix 3(b) will yield better results, for instance, because reachability tools can exploit the fact that p1.1 does not depend on z in any way. In other words, $\pi_{p1.1}(\langle x, y, z \rangle) = \langle x, y \rangle$ according to matrix 3(b).

Besides the exploration functions, PINS also contains functions that allow language modules to make data values known to the analysis algorithms. This is described in the next section from the language module's point of view.

IV. Interfacing to Language Implementations

A. From State Vectors to Index Vectors

In previous sections, we elaborated on how state vectors can have their structure exposed. In order to decouple state storage from the language module in the monolithic next-state interface, it is sufficient to define equality relations for state vectors and means to calculate their hash values. However, with the PINS interface, not state vectors but individual *slots* are accessed. Hence, we need to provide a more fine-grained solution. State slots contain any kind of data value that the underlying specification language can produce. Instead of providing comparison functions for all values, we opted for an inverse solution: a state vector $s_i = \langle s_{i_0}, s_{i_1}, \ldots, s_{i_\ell} \rangle_{\mathrm{SV}}$ consisting of values s_{i_j} is converted into its corresponding *index vector* $\bar{s}_i = \langle i_0, i_1, \ldots, i_\ell \rangle_{\mathrm{IV}}$ (and vice versa) by a PINS language module. State exploration algorithms deal primarily with index vectors, but they can query a language module for the data value corresponding to an index.

Beyond providing a simpler interface, the usage of index vectors has other advantages. State compression naturally falls out of index vectors; in effect, this is SPIN's (non-recursive) COLLAPSE method [19], [11]. We have shown earlier that *tree compression* yields even higher compression ratios for certain models, and how to benefit from index vectors in distributed state exploration [18].

B. Static Analysis for an Efficient NIPS-PINS Connection

For this paper, we restricted ourselves to a setting where the structure of state vectors is fixed and identical for all states, and hence also the number of state slots. On the downside, this would rule out languages which allow dynamic process or channel creation, like SPIN's PROMELA or other languages for the NIPSVM [15].

Even though PROMELA allows process creation at any point during "execution", many PROMELA models exist which only create a statically inferable number of processes and channels, and thus are perfectly suited for the techniques proposed in this paper. Such models can often be distinguished syntactically. However, for languages targeting the NIPSVM, parts of the process structure is not available any more at byte-code level. Still, the byte-code contains enough information for a static analysis to work: only two instructions can modify the structure of state vectors: process creation and channel creation byte-codes. Both cause a state vector to grow and change the number of state slots. In addition, *dead processes*, i.e., processes which reach the final state of their execution, need to be kept around. Unless they can influence[2] the rest of the system, they are normally removed from a state vector to save storage space. Channels are unproblematic in this respect because they cannot be destroyed.

We want to identify models which exhibit the above described behavior: for all possible execution paths from the initial state, after a small path prefix (the *setup phase*) no further operations are executed which modify the state group partitioning. In the remainder, we focus on NIPSVM byte-code programs because PROMELA models can also be compiled for execution on the NIPSVM [15]. Note that the analysis is generic, and also works, e.g., for NIPS byte-code derived from C programs for microcontrollers [20].

[2] e.g., if process-local variables are read by a monitor process

Algorithm 1 FIND-INITIALS(s)

```
{V: set of visited states}
if s ∉ V then
    V := V ∪ {s}
    if Creative(s) then
        for each s → s' do
            FIND-INITIALS(s')
    else
        I := I ∪ {s}
```

Ensure: I contains non-creative "initial" states

We split the analysis into two parts: a static analysis which annotates the control-flow graph (CFG) of a byte-code program, and a "run-time" analysis which explores a prefix of the state space guided by the annotated information. Our choice is pragmatic: while a more involved analysis could be performed to identify further models as usable, we do not expect their numbers to be large enough to outweigh the costs. We note that the run-time analysis will explore the whole state space already in this phase if it continues to find so-called *creative* states. In practice, an implementation could bail out after a pre-defined number of states has been visited to avoid this scenario.

We assume that the control-flow graph $\mathrm{CFG} = \langle \mathrm{PC}, \rightarrow_{\mathrm{CFG}} \rangle$ of a byte-code program consists of a number of nodes labeled with instructions. An edge $pc \rightarrow_{\mathrm{CFG}} pc'$ from one node to another denotes a possible transfer of control (e.g., via a conditional `JMPZ` byte-code instruction).

Definition 5 (Creativity)**.** *We define* Creative *as the (smallest) set of creative nodes in the control-flow graph* CFG *of a program M, and straightforwardly lift it to states s of the state space of M:*

$$\mathrm{Creative}(pc) = \text{"}\textit{instruction at } pc \textit{ can modify the state vector structure}\text{"} \vee \exists pc' \in \mathrm{CFG} : pc \rightarrow_{\mathrm{CFG}} pc' \wedge \mathrm{Creative}(pc')$$

$$\mathrm{Creative}(s) := \exists p \in \mathrm{Procs}(s) : \mathrm{Creative}(\mathrm{pc}(p))$$

$\mathrm{Procs}(s)$ *denotes the set of active processes in state s, and* $\mathrm{pc}(p)$ *the program counter of a process p (i.e., the associated control-flow node).*

The set I of non-creative initial states of M can then be computed by a standard depth-first search on the state space (Alg. 1).

A PINS *Matrix for* PROMELA*:* We first consider models without communication on synchronous channels. This is reasonable because our tools are targeted at exploiting the combinatorial structure of such models. In consequence, for any given transition of the state space only one process is

influenced. We can then define a simple dependency matrix for PROMELA which reflects this situation:

$$\begin{array}{c} \\ g_1 \\ g_2 \\ \vdots \\ g_k \end{array} \begin{array}{c} \begin{array}{cccccccccc} G_1 & \cdots & G_j & P_1 & P_2 & \cdots & P_k & C_1 & \cdots & C_\ell \end{array} \\ \left[\begin{array}{cccccccccc} 1 & \cdots & 1 & 1 & 0 & \cdots & 0 & 1 & \cdots & 1 \\ 1 & \cdots & 1 & 0 & 1 & 0 \cdots & 0 & 1 & \cdots & 1 \\ \vdots & \ddots & \vdots & \vdots & & \ddots & & \vdots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & 0 & \cdots 0 & 1 & 1 & \cdots & 1 \end{array}\right] \end{array}$$

We define one group per process in this scheme. All transitions of a process P_i are part of its transition group g_i. Without further information we can merely assume that all global resources (variables G_x and channels C_y) are referenced by every process. The matrix allows already all PINS tools to be applied successfully. We have implemented this option for our experiments in Sec. VI.

However, it is clear that with a better analysis the matrix can easily be refined to more accurately reflect the combinatorial nature of a model, thus increasing the efficacy of the PINS tools. For example, usually not all processes reference all global variables or channels. Replacing 1-entries with 0 in those cases will make the matrix less dense, and it is likely that symbolic tools can be sped up, and also use less memory for internal bookkeeping.

Another optimization is to introduce more groups (that is, more rows in the matrix): instead of lumping all transitions of a process into one group, we can also consider the other extreme: one group (row in the matrix) per *atomically executing block* in a byte-code program.

The "group per atomic block" scheme will likely result in rows which are sparsely populated with 1-entries because typically only few global resources are accessed per atomic block. On the other hand, it might turn out that several rows are identical. This will result in unnecessary work for the PINS tools. The corresponding groups can be merged, thus reducing the number of rows in the matrix. Again, this type of *regrouping* operations can be done purely on the PINS matrix, irrespective of the input language (cf. Fig. 1, block "Regrouping").

We can lift the restriction on synchronous communication among processes by declaring that a transition group depends on all processes that can possibly take part in a multi-way synchronous communication. We then obtain also rows with multiple 1-entries in the P_i columns of the matrix.

C. A PINS Matrix for μCRL and mCRL2

We have implemented a PINS interface for the μCRL and mCRL2 tool sets [21], [22], [23]. The concept of both toolsets is to take a process algebra specification and compile it into a linear process equation (LPE). An LPE is a process given as an initial state and a recursive equation:

$$X(x_1,\ldots,x_N) = \sum_{i=1}^{K} \underbrace{\sum_{e_i \in E_i} C_i \Rightarrow a(t_{i,0}).X(t_{i,1},\ldots,t_{i,N})}_{\text{summand } i}$$

where C_i and $t_{i,j}$ are expressions over $e_i, x_1, \ldots, x_N$. The intended meaning of this equation is that to perform a step, we first non-deterministically select $1 \le i \le K$ (determining a summand), then non-deterministically select some $e \in E_i$, evaluate the condition C_i to see if the transition is enabled and if it is enabled then the label of the transition is the result of the expression $a(t_{i,0})$ and the next state is $t_{i,1}, \ldots, t_{i,N}$.

Hence, an LPE admits a natural partitioning by assigning each summand its own group. Selecting this partitioning, we can define the contents of a PINS matrix $\mathrm{DM}(X) = [d^X_{i,j}]_{K\times N}$ as follows:

$$d^X_{i,j} = \begin{cases} 1 & \text{if } t_{i,j} \neq x_j \ \vee\ \exists 0 \le k \le N, k \neq j : \\ & \quad x_j \text{ occurs in } C_i \text{ or } t_{i,k} \\ 0 & \text{otherwise} \end{cases}$$

We have implemented this natural partitioning and we use it for the experiments presented in this paper.

D. Linking PINS to ODEs in MAPLE

In order to work with biological systems in the EC-MOAN project,[3] we have also established a PINS link to abstractions of *ordinary differential equations* (ODE) in MAPLE. Up to variations, the abstraction scheme for a system of N equations is to divide each of the N axes into several intervals, which leads to dividing the whole space into a grid of N-dimensional *boxes*. These boxes form the states of the abstraction and there is an edge from one box to a neighboring box if there exists a trajectory from the first box to the second. Thus, any real trajectory can be matched by a path, but not vice versa. This is a simplified version of the abstraction described by Batt et al. [24].

The simplified abstraction is easy to partition. Each of the variables is assigned to its own state slot and each of the derivatives is assigned to its own transition group. In this partition, a transition group depends on a state slot if the variable (state slot) occurs in the differential equation (transition group).

For example:

$$\mathrm{DM}\begin{pmatrix} \dot{x} &=& 1-x \\ \dot{y} &=& x-\frac{1}{2}y \\ \dot{z} &=& y-2z \end{pmatrix} = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \end{array}$$

$$\mathrm{DM}\begin{pmatrix} \dot{x} &=& y \\ \dot{y} &=& -x \end{pmatrix} = \begin{array}{c} \\ e_1 \\ e_2 \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \end{array}$$

In this way, we are able to apply symbolic and distributed state space generation for discrete abstractions of ODEs from Systems Biology.

V. CONNECTING TOOLS TO PINS

We have implemented three reachability tools on top of PINS: a symbolic tool, and enumerative sequential and distributed tools. The enumerative tools support a new feature: *transition caching*. We will not explain the enumerative tools

[3] http://www.ec-moan.org/

Algorithm 2 NEXT-STATE-ALL-CACHE(s)

```
succ := ∅
for all 1 ≤ i ≤ K do
    succ := succ ∪ NEXT-STATE-CACHE(i, s)
return succ
```

Algorithm 3 MAINTAIN-CACHE(i, s)

```
if cache[i](π_i(s)) = ⊥ then
    succ_i := ∅
for all s' ∈ NEXT-STATE(i, s) do
    ADD π_i(s') TO succ_i
cache[i](π_i(s)) := succ_i
```

Algorithm 4 NEXT-STATE-CACHE(i, s)

```
MAINTAIN-CACHE(i, s)
succ := ∅
for all t ∈ cache[i](π_i(s)) do
    k := 1
    for all 1 ≤ j ≤ N do
        if D_{i,j} = 1 then
            s'[j] := t[k]
            k := k + 1
        else
            s'[j] := s[j]
    ADD s' TO succ
return succ
```

because they are mostly standard (Sec. II). Our tools are available as part of the LTSMIN toolset.[4]

1) Local Transition Caching: Event locality can be used to speed up next-state computations. Within a transition group, it is not necessary to recompute the successors for every state from scratch. If the successors of a state s are known and we need to compute the successors of a state s', such that all slots which the transition group depends on have the same values in s and s' then the successors of s' can be obtained from the successors of s simply by replacing the values in the independent slots of the successors with the values of the corresponding slots in s'.

This property can be exploited by maintaining a cache for each transition group separately. With Alg. 2, we present an algorithm for computing successors that uses such caches. The code is straightforward, except for the use of the projections π_i (Def. 4). We have implemented this caching technique as a PINS2PINS-transformer. Thus it can be used with any tool that builds on PINS.

Using caching is not free. First, there is a small overhead in time, for instance due to cache lookups. More importantly, we need extra memory to store the cache. For sparse dependency matrices one expects that each transition group generates only a limited number of local transitions. On the other hand, if one row in the matrix has many 1-entries, the memory allocated for the corresponding cache may grow prohibitively.

2) Symbolic Reachability: The symbolic tool uses a kind of Multi-way Decision Diagrams (MDD) to store sets of states and transition relations. It builds a symbolic transition relation for each transition group and the set of reachable states in parallel. It cannot build the transition relation in advance because the size of the domain of a state slot is often infinite. The reachability analysis is needed to compute the finite set of values that actually occur.

The transition relations are built by calling the NEXT-STATE function once for every unique combination of relevant state slots and adding the transition with the identity relation for all irrelevant state slots. Recalling the example in Sec. III, we obtain as successor $\langle 0, 2, 1\rangle_{\text{SV}}$ when considering the state $\langle 1, 1, 1\rangle_{\text{SV}}$ and transition $p1.1$. Therefore, the (projected) transition relation $T_{p1.1}(x, y, x', y')$ for group $p1.1$ is changed to $T_{p1.1}(x, y, x', y') \vee (x = 1 \wedge x' = 0 \wedge y = 1 \wedge y' = 2)$. For a more detailed explanation of the symbolic reachability algorithm we refer to Blom et al. [10].

One factor in the performance of the tool is the number of calls to NEXT-STATE. The worst possible case is if a row in the dependency matrix contains only 1-entries because then NEXT-STATE has to be called for every reachable state. If the row contains one or more 0-entries then the number of calls to NEXT-STATE can be dramatically less than the number of states. Thus, the "one group for every statement" partitioning should be used for our example. If we assume an initial state $\langle N-1, 0, 0\rangle$ then for the process partitioning, we get three transition relations built with N^3 calls each and for the statement partitioning we get six relations built with N^2 calls each.

Another factor in the performance of symbolic reachability is the exploration order. We have implemented two orders so far, both work in iterations. The first strategy is *breadth-first* (BFS). In each iteration, the tool extends all transition relations and then extends the set of reachable states by those reachable from the original set of reachable states. The other strategy is a form of *chaining*, see also [25], and will cycle through the transition groups. It extends the transition relation for the current group, adds new reachable states and then continues with the next transition group. We have not yet implemented the saturation strategy of [9].

VI. BENCHMARKS

Using the tools described in Sec. V, we ran a series of benchmarking experiments. For these tests, we used a cluster with Quad-core Dual Intel Xeon 5335 CPUs and 8 GB memory each. Each test had exclusive access to the machine. The distributed tests were performed using 6 cores per machine on 1, 2 and 4 machines. In Tab. I, we present the results of a number of representative experiments.[5] In each case, we ran one or two tests using existing tools and several variants of our own tools, which we linked via PINS directly to the otherwise *unmodified* state generators of other toolsets. The *enum* column describes a sequential explicit-state tool, the *sym* column the symbolic tool with BFS exploration order;

[4] http://fmt.cs.utwente.nl/tools/ltsmin/

[5] For space reasons, a table containing more than 500 experiments can be found at http://fmt.cs.utwente.nl/tools/ltsmin/benchmarks/

6W, 12W and 24W describe the distributed experiments. The columns with *+cache* contain results from adding transition caching, the *+chain* column contains results for symbolic exploration in chaining order (Sec. V). The memory for the distributed tools is the total for all workers in bytes.

First, we will describe the case studies selected and the chosen existing tools. Afterwards, we will use the data to make the point that having both symbolic and explicit-state reachability with and without caching, can be crucial.

ccp33 This model has $9.7 \cdot 10^7$ states and $100 \cdot 10^7$ transitions. It describes an instance of the cache coherence protocol *Jackal* for Java programs with 3 processes and 3 threads [26].

msmie This model has $7.1 \cdot 10^6$ states and $11 \cdot 10^6$ transitions. It was taken from the BEEM repository.[6] It is a variant of Multiprocessor Shared-Memory Information Exchange [27] with 10 slaves, 10 masters and buffer size 5.

sw3 This model has $1.7 \cdot 10^8$ states and $20 \cdot 10^8$ transitions. It is a simplified model of the "schedule world" example from the AIPS 2000 contest, taken from the BEEM repository.

leader9 This model has $2.2 \cdot 10^8$ states and $16 \cdot 10^8$ transitions. It is a model of the DKR leader election algorithm [28], written in PROMELA and compiled to NIPSVM byte-code.

The native run in the lpo row (N2) is μCRL's INSTANTIATOR. The native runs in the DVE row refer to the DIVINE.GENERATOR with DVE (N1) and NIPSVM (N2) input, respectively. The native runs in the lps rows refer to MCRL2.LPS2LTS with `jittyc` rewriter, with the default `-fvector` (N1) and the `-ftree` (N2) state representation, respectively. The native tool for PROMELA is SPIN, with the options `-DNOREDUCE -DCOLLAPSE -DSAFETY` (N1) and `-DMA` added (N2).

The cache coherence model was used to study the symbolic tool before. So far we have not been able to get good symbolic performance for it. The sequential tool is roughly 10% slower than the native tool, but its memory footprint is only a quarter. By using transition caching, we could beat the native state space generator in time as well. The speedup of the distributed tool with caching disabled is reasonable but not perfect. It is also clearly visible that the effectiveness of caching decreases if the worker count grows. This is to be expected due to the fact that caches are local: instead of computing a cache entry once, it may be (re)computed by every worker. We also ran the mCRL distributed state space generator [18]. This tool took 14090s, 8003s and 3909s with 6, 12 and 24 workers.

The MSMIE model was originally specified in DVE, and compiled to NIPS and mCRL2. Out of the native tools, the DIVINE generator is the best. Of the translations the NIPS translation works best for the enumerative approaches. However, the best performance overall is achieved with the symbolic tools on the mCRL2 model. The performance of the symbolic tools on the NIPS model are an example of an anomaly in the tools. Usually, the symbolic tools are much faster if we use our own multi-way decision diagram library than when we use BuDDy. In this case, the numbers for our own library are much worse (165s/2495M and 1121s/5171M), even though we used the same fixed variable ordering.

The schedule world example shows that DVE isn't always better than mCRL2. In this case the DIVINE and the NIPS experiments (not shown in the table) all exceeded the time limit (24 h sequential, 4 h distributed). The mCRL2 experiments ran out of memory, as did the MPI version with 24 workers on 4 nodes. This experiment confirms the usefulness of caching at a modest memory penalty. Chaining is much better here than BFS, and the distributed version shows some (but not ideal) speedup.

For the leader election example, the symbolic case with chaining uses 50 % of the time and 5 % of the memory of the best enumerative solution. In all previous cases the numbers of states and transitions in all tools match, but SPIN reduces the model to $3.4 \cdot 10^7$ states and $22 \cdot 10^7$ transitions, which skews the comparison. SPIN and our symbolic methods are mostly incomparable: Symbolic BFS has just another time vs. memory trade-off than SPIN (it is actually in between N1 and N2). Chaining is incomparable to using minimized automata (N2), but is more attractive than N1: with slightly more time it achieves a considerable reduction in time.

In all of these examples, the memory cost of transition caching is visible. In some cases the cost is negligible (e.g., sw3) in other cases it is high (e.g., ccp33). Regarding time, all examples presented here show a benefit with transition caching. This is typical, but we know of cases where there is actually a penalty in time.

VII. CONCLUSION

We presented a solution to link explicit-state generators for different languages to both symbolic and distributed explicit-state model checking engines. We discovered that a simple interface, based on a partitioned next-state function and a dependency matrix, provides abstract information, yet sufficient for gaining efficiency. Adapting our tools to this interface enabled us to study how several techniques compare for a series of case studies in various languages, even exotics like ODEs with MAPLE. Moreover, we presented the results of a caching technique which can be combined with PINS tools in a generic way.

Our experiments show that symbolic techniques can be successfully applied to case studies written for enumerative tools. At the same time, they show that no technique is a clear winner for all models, and thus users are served best by having access to all of them. The PINS interface offers easy access to a range of tools for several languages.

Future Work. The tools can be improved in several ways. For example, we can use the dependency matrix to improve the initial variable ordering. We can also cluster transition groups with (nearly) identical rows in the dependency graph to reduce the number of symbolic next-state steps or cache lookups. Implementing these optimizations as another PINS2PINS wrapper, as in Figure 1, makes them available for several tools and languages.

[6] http://anna.fi.muni.cz/models/

TABLE I
SELECTED BENCHMARK RESULTS

Problem	N1	N2	enum	+cache	sym	+chain	6W	+cache	12W	+cache	24W	+cache
ccp33		67543s	73136s	28678s			14098s	12439s	7732s	7133s	4165s	4047s
(lpo)		7339M	1775M	1861M	>8G	>8G	2573M	3057M	2880M	3914M	3463M	5400M
msmie.4	169s	249s	139s	42s	17s*	37 s*	56s	43s	37s	34s	23s	24s
(dve/nips)	591M	1438M	163M	174M	28M*	28M*	292M	357M	325M	467M	454M	761M
msmie.4	481s	517s	539s	430s	11s	16s	130s	157s	80s	111s	50s	78s
(lps)	1780M	663M	250M	279M	20M	23M	412M	594M	603M	930M	865M	1546M
sw3			16076s	6536s	2220s	206s	7083s	5256s	4861s	4053s	2940s	1712s**
(lps)	>8G	>8G	3025M	3028M	7348M	1174M	4486M	4518M	5376M	5545M	6152 M	6566M**
leader9	490s	1467s	11391s	3407s	698s	504s	5565s		3880s	3322s	2356s	1494s**
(PROMELA)	2179M	16M	5508M	5622M	1667M	162M	8315M	>8G	10971M	12180M	14631M	18661M**

*: results obtained with BuDDy/fdd rather than ATermDD, see main text.
**: results for 24 workers on 8 nodes because the 24 workers on 4 nodes had load balancing problems.

Reachability analysis is useful, but not enough. We intend to produce output that can be *certified* by third-party tools (SMV, CADP, SMART, SPIN, etc.), and implement more powerful analyzers of our own.

Several partial order methods exist which rely on independence of assignments only. The dependency matrix contains enough information to implement those as yet another PINS2PINS wrapper.

Acknowledgement. The implementation of the mCRL2 language module was joint work with Jeroen van der Wulp.

REFERENCES

[1] Garavel, H.: OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In Steffen, B., ed.: TACAS. Volume 1384 of LNCS., Springer (1998) 68–84

[2] Bozga, M., Graf, S., Mounier, L.: IF-2.0: A validation environment for component-based real-time systems. [29] 343–348

[3] Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in jETI. In: ECBS, IEEE Computer Society (2005) 431–436

[4] Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In Bosnacki, D., Edelkamp, S., eds.: SPIN. Volume 4595 of LNCS., Springer (2007) 263–267

[5] Lang, F.: Exp.Open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In Romijn, J., Smith, G., van de Pol, J., eds.: IFM. Volume 3771 of LNCS., Springer (2005) 70–88

[6] Clarke, E.M., McMillan, K.L., Campos, S.V.A., Hartonas-Garmhausen, V.: Symbolic model checking. In Alur, R., Henzinger, T.A., eds.: CAV. Volume 1102 of LNCS., Springer (1996) 419–427

[7] Cabodi, G., Camurati, P., Lavagno, L., Quer, S.: Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In: DAC. (1997) 728–733

[8] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. [29] 359–364

[9] Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. STTT **8**(1) (2006) 4–25

[10] Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In Fitzgerald, J.S., Haxthausen, A.E., Yenigün, H., eds.: ICTAC. Volume 5160 of LNCS., Springer (2008) 81–95

[11] Visser, W., Barringer, H.: Memory efficient state storage in SPIN. In Grégoire, J.C., Holzmann, G.J., Peled, D., eds.: The Spin Verification System. Volume 32 of DIMACS., AMS (1997)

[12] Grégoire, J.C.: State space compression in Spin with GETSs. In: Proc. Second SPIN Workshop, Rutgers Univ, American Mathematical Society (1996)

[13] Baldamus, M., Schröder-Babo, J.: p2b: A translation utility for linking promela and symbolic model checking (tool paper). In Dwyer, M.B., ed.: SPIN. Volume 2057 of LNCS., Springer (2001) 183–191

[14] Holzmann, G.J.: The model checker Spin. IEEE Trans. Software Eng. **23**(5) (1997) 279–295

[15] Weber, M.: An embeddable virtual machine for state space generation. In Bošnački, D., Edelkamp, S., eds.: Proceedings of SPIN. Volume 4595 of LNCS., Springer (2007) 168–185

[16] Ciardo, G., Gluckman, J., Nicol, D.: Distributed state-space generation of discrete-state stochastic models. INFORMS Journal on Comp. **10**(1) (1998) 82–93

[17] Stern, U., Dill, D.L.: Parallelizing the Murφ verifier. In Grumberg, O., ed.: Computer-Aided Verification, 9th International Conference. Volume 1254 of LNCS., Springer (1997) 256–267

[18] Blom, S., Lisser, B., van de Pol, J., Weber, M.: A database approach to distributed state space generation. ENTCS **198**(1) (2008) 17–32

[19] Holzmann, G.J.: State compression in Spin: Recursive indexing and compression training runs. In: In Proceedings of Third International SPIN Workshop. (1997)

[20] Schlich, B., Rohrbach, M., Weber, M., Kowalewski, S.: Model checking software for microcontrollers. Technical Report AIB-2006-11, RWTH Aachen (2006)

[21] Blom, S., Fokkink, W., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: μCRL: A toolset for analysing algebraic specifications. In Berry, G., Comon, H., Finkel, A., eds.: CAV. Volume 2102 of LNCS., Springer (2001) 250–254

[22] Blom, S., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: New developments around the mCRL tool set. ENTCS **80** (2003)

[23] Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., Weerdenburg, M.V.: The formal specification language mcrl2. In: Proc. Methods for Modelling Software Systems. Dagstuhl Seminar Proceedings 06351. (2007)

[24] Batt, G., de Jong, H., Page, M., Geiselmann, J.: Symbolic reachability analysis of genetic regulatory networks using discrete abstractions. Automatica **44**(4) (2008) 982–989

[25] Thomas, D., Chakraborty, S., Pandya, P.K.: Efficient guided symbolic reachability using reachability expressions. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 120–134

[26] Pang, J., Fokkink, W.J., Hofman, R.F., Veldema, R.: Model checking a cache coherence protocol of a Java DSM implementation. JLAP **71** (2007) 1–43

[27] Bruns, A.: Gaining Assurance with Formal Methods. In Hinchey, M.G., Bowen, J.P., eds.: Applications of Formal Methods, Prentice Hall (1995)

[28] Dolev, D., Klawe, M.M., Rodeh, M.: An o(n log n) unidirectional distributed algorithm for extrema finding in a circle. J. Algorithms **3**(3) (1982) 245–260

[29] Brinksma, E., Larsen, K.G., eds.: Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. In Brinksma, E., Larsen, K.G., eds.: CAV. Volume 2404 of LNCS., Springer (2002)