High-Level Petri Net Model Checking with AlPiNA

HOSTETTLER, Steve Patrick, et al.

Abstract

Although model checking is heavily used in the hardware domain, it did not take off in software engineering yet. One of the possible reasons is that software models are very complex. They integrate many dimensions such as data types and concurrency, leading to the infamous state space explosion problem. This article introduces the Algebraic Petri Nets Analyzer (AlPiNA), a symbolic model checker for High-level Petri net. It is comprised of two independent modules: a GUI plug-in for Eclipse and an underlying model checking engine. AlPiNA is a step towards performing efficient and user-friendly model checking of large software systems. This is achieved by separating the model and its properties from the optimisation artifacts. This article describes the features that AlPiNA provides to the user for designing models and verifying properties. It also presents the techniques and artifacts used for tuning verification performance, along with some theoretical background.

Reference


DOI : 10.3233/FI-2011-608
HIGH-LEVEL PETRI NET MODEL CHECKING WITH ALPINA

STEVE HOSTETTLER AND ALEXIS MARECHAL AND ALBAN LINARD
AND MATTEO RISOLDI AND DIDIER BUCHS

Centre Universitaire d’Informatique, Université de Genève
Route de Drize 7, CH-1227 Carouge, Switzerland

Abstract. Although model checking is heavily used in the hardware domain, it did not take off in software engineering yet. One of the possible reasons is that software models are very complex. They integrate many dimensions such as data types and concurrency, leading to the infamous state space explosion problem. This article introduces the Algebraic Petri Nets Analyzer (AlPiNA), a symbolic model checker for High-level Petri nets. It is comprised of two independent modules: a GUI plug-in for Eclipse and an underlying model checking engine. AlPiNA’s goal is to perform efficient and user-friendly model checking of large software systems. This is achieved by separating the model and its properties from the optimisation artifacts. This article describes the features that AlPiNA provides to the user for designing models and validating properties. It also presents the techniques and artifacts used for tuning validation performance, along with some theoretical background.

1. INTRODUCTION

Model checking consists in verifying whether a model satisfies a given property, often expressed in temporal or modal logic. Model checking implies fully automated property proving. When a property does not hold on a model, the user gets a counterexample. Model checking requires expressing models using formalisms. Several approaches are possible for this. Using high-level formalisms (e.g., [Jen97a, Vau87, Rei91, BG00]) users can specify complex models in a compact and flexible way. The model checking benefits from the richness of information

Key words and phrases. System design and verification, Higher-level Nets Models, Algebraic Petri Nets, State Space Generation, Computer Tools for Nets, Model Checking.

S. Hostettler and A. Marechal were partially supported by the COMEDIA project funded by the Hasler foundation, project #2107.

A. Linard was partially supported by the BRINTA project funded by the Fonds National Suisse de la Recherche Scientifique, #200021-130159.

This article was published in Steve Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, and Didier Buchs. High-Level Petri Net Model Checking with AlPiNA. Fundamenta Informaticae, 113(3-4):229–264, 2011.
of such models. This article describes the Algebraic Petri Nets Analyzer (AlPiNA), a tool for model checking a particular high-level formalism called Algebraic Petri nets (APNs) [Vau87, Rei91]. In APNs, the model is composed of a Petri net (PN) and of Algebraic Abstract Data Types (AADTs). PNs express aspects related to causality, non-determinism and concurrency. AADTs describe data and their manipulation. AlPiNA is an integrated environment for modelling and checking systems. It pursues two main goals. The first is to perform reachability analysis on finite models in a user-friendly and efficient way. The second is to scale up to large models without requiring the end user to understand the underlying model checking techniques. AlPiNA provides a suite of graphical and textual editors integrated in the Eclipse environment. Its modular architecture includes an engine that makes use of several optimisations to improve model checking performance. In the current release, these optimisations are based on information about the model provided by the user via a Domain Specific Language (DSL). AlPiNA is available under the GPL license at http://alpina.unige.ch. Fig. 1 gives an overview of

AlPiNA’s approach to the model checking activity using Business Process Modeling Notation (BPMN). First ("Designing the model") the user designs a model of the system to analyse, containing also a set of properties that are expected to hold on the system. Usually the first versions of the model are rather small. Then ("Performing the validation") AlPiNA either confirms that the properties hold, or reports a violation. In the latter case, it provides a counterexample, i.e., a state that violates the property. The user may use the counterexample to refine the model of the system or to correct the property. Once these small versions of the model are correct, bigger models are considered. In order to handle the increase in the state space size, the user may provide optimisation information to improve the validation performance ("Scaling up the model"). The advantages of AlPiNA include its ability to manage large state spaces and the separation of concerns between modelling and optimisation. The former is achieved via the use of a particular approach to model checking, called Symbolic Model Checking (SMC). The latter is the result of the separation of the modelling and optimisation languages. As a matter of fact, modelling and optimisation are two independent activities:

![Figure 1. Model checking process using AlPiNA](image)
AlPiNA can perform model checking even without optimisation, although this limits the performance and the complexity of the models that can be handled. This article begins by illustrating the steps of APN model checking in AlPiNA shown in Fig. 1. For each activity of the process, details about the underlying theory and techniques will be given. Section 2 will describe the model design activity. Section 3 will talk about the validation. Section 4 will talk about optimisation. In Section 5, we will compare the performance of AlPiNA with other model checkers that use a variety of approaches. Section 6 will describe AlPiNA’s architecture. Finally, Section 7 will wrap up the article with conclusions and perspectives for future developments. Related work will be discussed throughout the article as the subject unfolds, instead of in a dedicated section.

2. Designing the model

AlPiNA covers both activities of designing models and performing validation, shown in Fig. 1. This section gives an informal overview of the model design activity, which is outlined in Fig. 2. The modelling task is twofold:

- first, users design the system to verify. This requires the definition of both the control flow and the data types of the system. In AlPiNA, APNs are used to express this information;
- second, users write properties that are expected to hold on the system. In AlPiNA, properties are expressed using an extension of first order logic [BE93], which allows verifying invariants.

![Designing the model](image)

**Figure 2.** Expanded view of the activity “Designing the model” of Fig. 1

This section will describe the design process with the help of a toy example, which is included in AlPiNA’s example repository along with other standard models. We first focus on the modelling formalism used in AlPiNA: Algebraic Petri nets (APNs).

---

1The example can be loaded in AlPiNA by following the menu:

`File > New > Example > AlPiNA > SimpleNet`.

All editors, UI elements and menus mentioned in the remainder of this section are available when the AlPiNA perspective is enabled via the menu

`Window > Open Perspective > Other > AlPiNA`.
APNs [Vau87, Rei91] are a combination of Petri nets (PNs), modelling concurrency and non-determinism, and Algebraic Abstract Data Types (AADTs) [EM85] modelling the data types of a model. PNs are used to model and analyse discrete systems where notions such as concurrency, synchronisation and interaction among components play a central role. The state of a system is modelled using places (i.e., system variables). The system’s evolution is represented by transitions.

In the original formulation of PNs, called Place/Transition Petri nets (PTs), only one type of token with a single value exists, called the black token. This restriction limits the expressiveness of PTs. Modelling complex systems using PTs leads to huge models that may be difficult to read and maintain. In other variants of PNs, called High-level Petri nets (HLPNs), tokens have an internal structure. This greatly increases the modelling power compared with PTs. APNs are a member of the HLPN family. Other examples of HLPNs are Coloured Petri nets (CPNs) [Jen97a] and Symmetric Petri nets (SNs) [DIPVM02] (formerly known as “Well-Formed Petri nets”). APNs in a way can be compared to CPNs, but they replace colour sets with algebras defined using AADTs.

2.1. Defining Data Types. AADTs describe data types in an abstract way, i.e., independently from a particular implementation. They can be seen as contracts that specify types and their values. AADTs define sorts, i.e., classes of objects, via their signature. They also define the observable behaviour of functions operating on the sorts. Functions are divided into two categories: generators and operations. Generators are the minimal set of functions on the sort that are necessary to construct any distinct element of that sort.

The observable behaviour (i.e., the semantics) of the functions is given by a finite set of axioms in the form of conditional equations. The equations and conditions may contain variables that are universally quantified. There are several ways to derive semantics from the axioms: denotationally (by defining a morphism to another algebra, e.g., the natural numbers), axiomatically (i.e., by equational logic) and operationally (i.e., as sequences of computational steps). AlPiNA takes the latter approach by using Term Rewriting (TR) [Ter03]. Term Rewriting uses a set of directed rewriting rules. These are derived from axioms specified in the data part of the model, by interpreting them from left to right. The derived rewriting rules must constitute an orthogonal (thus confluent) and terminating rewriting system, i.e., rules must be left-linear and non-overlapping, and any term must have a unique normal form. The normal form of a term is reached when no further rewriting rules can be applied on that term.

Example 2.1. The AADT for natural numbers is classically defined as a sort called nat with two generators, zero and suc (the successor). It also declares operations on naturals; for the sake of the example, let us declare just one plus operation for the addition:
The axiomatisation \( a_1, a_2 \) shown below gives the semantics of \( \text{plus} \) following the Peano arithmetic. Two directed rewriting rules \( r_1 \) and \( r_2 \) can be derived from \( a_1 \) and \( a_2 \):

\[
\begin{align*}
\text{a1} & : \text{plus}(x, \text{suc}(y)) = \text{suc}(\text{plus}(x, y)) \\
\text{r1} & : \text{plus}(x, \text{suc}(y)) \rightarrow \text{suc}(\text{plus}(x, y)) \\
\text{a2} & : \text{plus}(x, \text{zero}) = x \\
\text{r2} & : \text{plus}(x, \text{zero}) \rightarrow x
\end{align*}
\]

Let us suppose that one wants to represent the \( 2+1 \) operation on naturals. The value 2 is represented as \( \text{suc}(\text{suc}(\text{zero})) \) (i.e., twice the successor of \( \text{zero} \)). Likewise, \( \text{suc}(\text{zero}) \) represents the value 1. The whole operation is represented by the term \( \text{plus}(\text{suc}(\text{suc}(\text{zero})), \text{suc}(\text{zero})) \), which is rewritten as \( \text{suc}(\text{suc}(\text{suc}(\text{zero}))) \) (i.e., the natural value 3) by application of \( r_1 \) followed by \( r_2 \).

Iter theory [CDE+02] is supported in order to enable shorthand notation for repeated application of the same operation on a term. Thus, in order to represent the natural number 100, one can use the term \( \text{suc}^{100}(\text{zero}) \) instead of writing a long sequence of \( \text{suc}(\text{suc}(\text{suc}(\ldots))) \).

\( \square \)

\( \text{AlPiNA} \) also supports order-sorting [GM92]. A subsort is a sub-set of the elements of the parent sort. The functions defined in the parent sort are preserved for the elements of the sub-sort. Subsorting is mainly used to avoid partial functions. For instance, the axiomatisation of the division of natural numbers \( \text{div} : \text{nat}, \text{nat} \rightarrow \text{nat} \) requires a non-zero condition on the second operand. An elegant way to solve this problem is to make \( \text{div} \) a total function. For that, we require the second operand to be of sort \( \text{nznat} \), which is a subsort of \( \text{nat} \) that does not contain \( \text{zero} \). The generators and rewriting rule become:

\[
\begin{align*}
\text{zero} : & \rightarrow \text{nat} \\
\text{suc} : \text{nat} & \rightarrow \text{nznat} \\
\text{div} : \text{nat}, \text{nznat} & \rightarrow \text{nat}
\end{align*}
\]

Using AADTs brings several benefits to the end-user, especially when compared to more ad hoc approaches (e.g., native types). Among other things, AADTs allow users to reason on data types in a general and abstract way. For example, axiomatisation enables theorem proving. Moreover, the inductive nature of AADT is well-suited to model complex structures such as lists, stacks, sets or trees.

Inductive data structures are indeed used in \( \text{AlPiNA} \) for optimisation purposes as will be shown in Section 4. Fig. 3 shows an excerpt from the naturals AADT using \( \text{AlPiNA} \)’s syntax. The complete file is named \( \text{Naturals.adt} \) and can be found in the \( \text{SimpleNet} \) example (as well as in any new \( \text{AlPiNA} \) project). The naturals AADT begins by importing the definition of other AADTs, which are used in some of the operation definitions. Then it defines a sort called \( \text{nat} \) and two generators \( \text{zero} \) and \( \text{suc} \). All values in the algebra can be represented by combining these generators.
The third section declares the operations. The behaviour of operations is defined in the Axioms section. The AADT ends with the declaration of the variables used in the axioms. AlPiNA’s data types provide more features than this example shows, like sub-sorting, polymorphism or data type modularity similar to that of [BG00]. The AADT editor provides syntax highlighting and auto-completion. An interactive term rewriter is included where one can specify a term and ask for its normal form. This is useful while debugging a data type definition.

2.2. Defining the control flow. While AADTs define the data types of the system, its control flow is described by APN places, transitions and arcs. Places usually represent process states or system resources. Each place contains a multiset of values of a single sort, i.e., a set allowing multiple occurrences of the same value. For instance, the multiset $[\text{succ(zero), 2\cdot \text{zero}}]$ contains once the value 1 and twice the value 0. An empty multiset is denoted by $[]$. For the user, defining a multiset by explicit enumeration of its contents may be tedious. Thus, in addition to the standard way of defining multisets, AlPiNA supports multiset definition by intension. For instance, let us suppose one wants to represent the multiset of naturals that contains every number between 0 and 100 twice. The standard definition would be an enumeration like $[2\cdot \text{zero}, 2\cdot \text{succ(zero)}, \ldots, 2\cdot \text{succ}^{100}(\text{zero})]$. The definition by intension is much more compact: $[2\cdot (n : \text{le} (n, \text{succ}^{100}(\text{zero})))$ where le is the less than or equal to operation.

The multiset contained in a place is called the marking of that place. It represents the state of the corresponding resource. The set of place markings is called the marking of the APN and it represents the state of the system. Note that there is no way in PNs to distinguish a process state from a resource.

Like places, arcs are labelled with multisets. A transition may consume resources (i.e., tokens) according to the labelling of its input arcs. It may also generate resources according to the labelling of its output arcs. A predicate called guard may also be attached to each transition. A guard is a conjunction of term equations.

In transition firing, the variables on the input arcs are bound to the values
consumed from the input places. This is called the binding of the transition. Note that we do not consider bindings that do not satisfy the input arcs. A transition can be fired if its binding satisfies the guard. The binding may be used to compute the output values of the transition according to the terms attached to the output arcs.

To define the control flow, AlPiNA offers a graphical interface, where places, transitions and arcs are annotated with algebraic terms. Fig. 4 is a screenshot of the graphical interface. A tool palette on the right allows the user to create Petri net elements (i.e., places, transitions, arcs). The properties of the elements can be edited using the standard Eclipse Properties view (bottom part of the screenshot). For instance, in Fig. 4, the place $P_1$ is selected, and the Properties panel shows its name, sort and the contained multiset.

![Figure 4. AlPiNA's GUI showing an example APN](image)

**Example 2.2.** Fig. 4 shows the net of a toy example. This APN is comprised of two places $P_1$ and $P_2$ and a transition $T$. $P_1$ initially contains two tokens: $\text{suc}^3(\text{zero})$ and $\text{suc}^4(\text{zero})$. $P_2$ is initially empty. $T$ consumes a value from place $P_1$ binding it to $\$x$. It also produces one token $\text{suc}(\$x)$ in the place $P_2$. There are two possible bindings for firing $T$: $\$x := \text{suc}^3(\text{zero})$, which will produce $\text{suc}^4(\text{zero})$ in $P_2$; and $\$x := \text{suc}^4(\text{zero})$, which will produce $\text{suc}^5(\text{zero})$ in $P_2$. In this example, $T$ can only be fired twice, after which there will be no available tokens in $P_1$ to obtain a binding for $\$x$.

2.3. **Property definition.** To perform verification, it is necessary to specify the properties that the design must satisfy. There are two main families of properties:
invariant/reachability properties:: they are evaluated on each state of the system. They are usually expressed with first-order logic. In this case, the predicates are defined over the individual states of the system, using a tool-specific language.

**temporal properties::** they assert how the system evolves over time. Therefore, they are evaluated on sequences of states. They require to extend the properties defined over single states by using temporal logic, such as Computation Tree Logic (CTL) [CE82] or Linear Temporal Logic (LTL) [Pnu77].

The added expressiveness of temporal properties comes at a cost. First, it requires expertise for the user to write and understand properties. Second, it requires more advanced model checking techniques and is more time and memory consuming.

In **AlPiNA**, we have chosen to implement a property language that is roughly equivalent to first-order logic with deadlock detection. This language was inspired by that of **Helena** [PE09], a well-known model checker for HLPNs, but was adapted to benefit from the flexibility of AADTs. This section presents an example of property definition, which shows some of the features of the language. A complete description of the syntax and semantics of the property language is out of the scope of this article and can be found in [MB10].

Like all text-based editors in **AlPiNA**, the property editor provides syntax highlighting and automatic completion. A property is a Boolean expression that must be evaluated to *true* for every state of the model. If there is a reachable state where the property does not hold, a textual representation of this state is returned as a counterexample\(^2\).

\(^2\)As of version 1.0, only one counterexample is returned and the tool does not provide a trace of the transitions fired to reach it.
Example 2.3. Fig. 5 shows a screenshot of the property editor containing property definitions for our toy example. The property file starts with the import of the used AADTs and APNs. The Expressions section defines Boolean predicates that can be used to form the property. This is done in the Check section by combining the predicates via Boolean operations. In Fig. 5, the empty predicate states that the tokens in places P1 and P2 must have different values. The atMost predicate states that there are at most two tokens with value 4 or 5 in P1 (the card operator yields the cardinality of a multiset). The notEmpty predicate states that P1 is never empty. The property @empty & @atMost checks that both expressions hold on every state of the system. The property definition does not have to use all predicates defined in the file: in this example, notEmpty is defined but not used in property check. The file ends with variable declarations.

Property checking in AlPiNA is launched via a Check property button on the tool palette (seen in Fig. 4). Checking the property defined in Fig. 5 will produce a counterexample, i.e., a state where the property does not hold: < P1: [1\*suc^4\{zero\}]; P2: [1\*suc^4\{zero\}] > This state indeed violates the empty predicate. It can be reached from the initial state by firing T with a binding of $x := suc^3(zero)$.

3. Performing the validation

Once a model and its properties have been defined, the next logical step is to check whether the model satisfies the properties. Unlike model design, validation is an activity that is typically automated. Fig. 6 gives an overview of the validation process in AlPiNA. First, a pre-processing step prepares the model for state space computation. During this step, static checks (i.e., static type checking) are performed on the model. Also, the pre-processor unfolds the domain of the free variables and the sets of terms declared by intension. Namely, it instantiates free variables with each value of their domain (either completely or up to a given bound). After that, the model checker computes the reachable state space, i.e., the set of the possible states the system can reach from the initial state. Finally, the properties are checked on the state space. If the property does not hold for all states, then AlPiNA returns one of the states that violate it.

This section will now go into the details of AlPiNA’s approach to model checking. We will see the theoretical foundations as well as the state space encoding techniques. Finally, we will describe how property checking is performed.

3.1. Managing complexity. One of the major issues of model checking is the State Space Explosion (SSE) problem [Val98]. A naive approach to model checking is to compute sequentially all the states that are reachable from the initial state, and to check whether a property holds for each of them. To know that a state has already been visited, the model checker has to keep track of the whole state space. Usually, this is done using some sort of map of states. This requires
3.2. Symbolic Model Checking. One way to avoid SSE is to use a state space encoding with a lower complexity than the explicit enumeration of states. McMillan [BCM+92] realised that using Bryant’s Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bry86] — a variant of Decision Diagrams (DDs) — it was possible to encode the transition relation symbolically. This approach is called Symbolic Model Checking (SMC). Because ROBDDs are well-suited to express regularity in the state space of circuits and protocols, they provide an encoding that is usually logarithmic to the number of states in the best case and linear in the worst. Many authors improved the original idea: Ciardo et al. [CLS00], Couvreur et al. [CEPA+02, CTM05] have checked models having up to $10^{2500}$ states. The main drawback of these approaches is that model design is influenced by considerations about the model checking performance. Because of this, the model designer must have a deep knowledge of the model checking techniques.

**AlPiNA** extends these works to APNs. While doing so, it clearly separates the model and the information for model checking optimisation. For instance, previous works such as [HKPAE10] require to statically unfold an HLPN (i.e., transform it into a PT with an isomorphic transition relation, as in [Mur89]) before computing the state space. **AlPiNA** handles a particular kind of unfolding (see 4.1) but it is not mandatory. This allows **AlPiNA** to handle models that cannot be completely unfolded (e.g., models with infinite data types). The commutative diagram in Fig. 7 explains the symbolic approach and its correctness principle. Let $S$ be the set of all possible markings of an APN, and $S_0 \in \mathcal{P}(S)$ the set of its initial states. The set $\tau(S_0)$ of states reachable from $S_0$ can be calculated by applying $\tau$, the transitive
closure of the application of the set of transitions \( T \) on \( S_0 \) (top-left part of Fig. 7). The calculated sets of states must be encoded in some domain \( \mathbb{D} \) (\( \text{enc}_S: \mathcal{P}(S) \to \mathbb{D} \)), link between the top and bottom part of Fig. 7). The transitive closure is encoded as well (\( \text{enc}_T(\tau): \mathbb{D} \to \mathbb{D} \)), bottom-left part of Fig. 7). The central point of SMC is the choice of a \( \mathbb{D} \) that can encode sets of states with a sub-linear complexity. McMillan proposes ROBDDs as the co-domain for \( \text{enc}_S \) and \( \text{enc}_T \). Couvreur et al. propose to use Data Decision Diagrams (DDDs) for the same purpose, as well as DD-Homomorphisms (DDHoms) to encode \( \text{enc}_T \). As computing the state space is not enough, it is also necessary to encode the check of the properties. The right part of Fig. 7 illustrates the verification of a property \( \Phi \) on the state space and its encoding \( \text{enc}_P(\models \phi) \). Symbolic Model Checking approach proposes to apply \( \text{enc}_P(\models \phi) \) on the symbolic state space \( \text{enc}_T(\tau)(\text{enc}_S(S_0)) \). A DDD encodes a set of sequences of variable assignments. Each sequence encodes a state. Some variables assignments may be shared by several sequences, therefore modifying one assignment may impact a set of states. To manipulate DDDs, one can use set operations and user-defined functions. These functions are homomorphic with respect to the union operation.

DDDs have several relevant properties. For instance, their size — i.e., the number of arcs and nodes — does not depend directly on the quantity of stored data. This size can be exponentially smaller w.r.t. the actual size of stored data. Because a DDD is a canonical representation of the set of data its represents, implementations can represent each set of data uniquely in memory. This is usual in DDs since ROBDDs, using the flyweight design pattern [GHJV95] (also known as hash-consing [Got74]). Because of the canonicity and unicity of their representation, the equality of two DDDs is checked in constant time. This property enables memoization [Mic68], which provides efficient fixed point computation. DDDs and their extensions — e.g., Hierarchical Set Decision Diagrams (SDDs) [CTM05], Multi-Set Decision Diagrams (MSDDs) [LH09] and \( \Sigma \) Decision Diagrams (\( \Sigma \)DDs) [BH09b] — are not limited to embedding Boolean values. Their arcs are labelled with values (DDDs), sets of values (SDDs and MSDDs) or sets of terms (\( \Sigma \)DDs). Unlike Binary Decision Diagrams (BDDs), they represent sets of sequences of variable assignments. Thus, the same variable can be repeated along a path. The only constraint is that all paths are compatible as defined in [CTM05], i.e., there is only one possible variable for each DD node. Henceforth, the term DD will refer to DDDs, SDDs, MSDDs and \( \Sigma \)DDs alike. An extensive classification of DDs is beyond the scope of this article and can be found in [LPAK+10].

3.3. An example. Fig. 8 along with the AADTs of Appendix A shows an example
producer-consumer model. It originates from [Jen97b] (page 58) with some slight adjustments to use AADTs instead of colour sets. We will use this as a running example in the remainder of the article. The model describes producers (place \texttt{Prod}) who produce (transition \texttt{produce}) and send (transition \texttt{send}) a packet — identified by a producer and a consumer — via a buffer (place \texttt{Buff}). Consumers (place \texttt{Cons}) receive (transition \texttt{receive}) and consume (transition \texttt{consume}) packets. Please note that we chose an extremely reduced version of this model, with only one consumer, to ease the explanation in the following sections. Some elements of the model should be highlighted:

- in Fig. 8, and in the rest of the examples, we used a shorthand for terms — e.g., $c_0$, $c_1$, ... instead of $c_0$, $c(c_0)$, ..., and $p_0$, $p_1$, ... instead of $p_0$, $p(p_0)$, ... — for increased readability;
- the output arc of transition \texttt{produce} contains variable $c$ of sort \texttt{consumer}. This is a free variable, \textit{i.e.}, an output variable that is not bound to an input variable. Therefore the term $pk(p_{\text{prod}}, c)$ will be unified to the input variable $p_{\text{prod}}$ and to \textit{any} consumer. For the sake of the presentation, this particular example has only one consumer (defined in the \texttt{Consumers} AADT);
- the transition \texttt{send} has a guard: $\texttt{isFull(b)} = \texttt{false}$. This enables the firing of \texttt{send} only if the input variable $b$ — which represents the current state of the buffer — is not full.

The function $\texttt{push(p, b)}$ on the output arc queues packet $p$ in buffer $b$. On the other output arc, $\texttt{getProd(p)}$ extracts the producer from packet $p$.

Figure 8. Algebraic Petri net of a producers/consumers model
the guard of transition receive is a conjunction of two equations: isEmpty($b$) = false that is true when the buffer is not empty and getCons(head($b$)) = $c$ that checks that the current consumer is the actual recipient of the packet extracted from $b$ (using the function head);

- the signature and semantics of functions isFull, push, head, tail, isEmpty are defined in an AADT called Buffers.adt. The functions pk, getProd and getCons are defined in an AADT called Packets.adt.

3.4. Encoding the State Space. The system’s state is modelled using a vector of places representing system variables. These places contain a multiset of values, which in APNs are terms of a sort. Let $[x]_p$ be the multiset $[x]$ contained by the place P. The initial state of the APN in Fig. 8 (i.e., the contents of all the multisets in all the places) is noted as: $\langle [p_0, p_1]_{\text{Prod}}, [c_0]_{\text{Cons}}, [\text{empty}]_{\text{Buff}} \rangle$.

Fig. 9 gives an intuition of how to encode this state using a hierarchy of DDs:

- at the lower level, a DDD encodes the vector of places representing the initial marking of Fig. 8. The DDD variables (e.g., Prod, Cons) represent the places of the APN. Their arcs contain pointers to the representation of their respective multisets. We say that a variable embeds the pointed multiset;

- at the middle level, multisets are encoded by MSDDs embedded in the DDD variables. The key characteristic of MSDDs is their support of multi-terms. Unlike DDDs or SDDs, MSDDs’s terminals are not limited to 0 and 1. They can take any natural number that represents the cardinality of the value they embed. The variables in MSDDs represent the sort of the multiset. We note $\text{enc}_M([x])$ the function encoding the multiset $[x]$ as a MSDD. Notice that for readability reasons, we replace the encoding of the multisets by the $\text{enc}_M$ function in Fig. 12, 13, 20 and 24;

- at the upper level, $\Sigma$DDs encode sets of terms. These are described later in this section.

Remark 3.1. The figures in this section that represent DDs have been slightly adjusted for readability. First, pointers to terminal structures (like $\{p_0, p_1\}$ or $\{\text{empty}\}$ in Fig. 9) have been omitted. Second, the terminal 1 has been represented.
once for each branch of the DD, whereas in an actual DD all branches would point to the same shared terminal. The same stands for the empty multiset $\emptyset$.

Figure 10. The $\Sigma$DD that encodes the term $\text{push}(\text{pk}(p0, c0), \text{empty})$

3.4.1. Encoding terms. In APNs the pre- and post-conditions of transitions are described by terms that may contain variables. To compute transition firing, all the possible bindings for these variables must be considered. The number of terms obtained is often close to the size of the Cartesian product of the variable domains. This is the motivation for pursuing efficiency in encoding and rewriting huge sets of terms using DDs. SDDs are a good solution for this problem, because of their inductive nature and the possibility to encode rewriting procedures using DDHoms. However, even higher efficiency can be achieved by introducing stronger typing constraints and specialising set operations to support order-sorting. To obtain this, we developed an extension of the SDDs, called $\Sigma$DDs [BH09b]. Without getting too deep in the formalisation of the $\Sigma$DDs, we will give some intuitive examples.

Fig. 10 shows the $\Sigma$DD that encodes the singleton set $\{\text{push}(\text{pk}(p0, c0), \text{empty})\}$. Like in MSDDs, $\Sigma$DD nodes represent the sort of the encoded terms: $B$ is the sort in Buffers.adt, $K$ is the sort in Packets.adt, $P$ is the sort in Producers.adt and $C$ is the sort in Consumers.adt. Arcs either contain operations and generators — push, pk, $p0$, $c0$ and empty — or point to other $\Sigma$DDs, i.e., subterms. The structure of the main $\Sigma$DD follows the signature of the generator $\text{push} : K, B \rightarrow B$. Similarly, the $\Sigma$DD representing the subterm $\text{pk}(p0, c0)$, embedded as the first argument, follows the signature $\text{pk} : P, C \rightarrow K$ of the generator pk. Finally, the innermost subterms of Fig. 10 represent the constants $p0$, $c0$ and empty.

Fig. 11 presents a more complex example encoding a set of nine terms, with two producers and three consumers. Please note that in order to build such terms the Consumers AADT should be modified to allow the definition of more than one consumer.

$$
\begin{align*}
(1) \ & \text{push}(\text{pk}(p0, c0), \text{push}(\text{pk}(p1, c0), \text{empty})) \\
(2) \ & \text{push}(\text{pk}(p0, c0), \text{push}(\text{pk}(p1, c1), \text{empty})) \\
(3) \ & \text{push}(\text{pk}(p0, c0), \text{push}(\text{pk}(p1, c2), \text{empty})) \\
(4) \ & \text{push}(\text{pk}(p0, c1), \text{push}(\text{pk}(p1, c0), \text{empty})) \\
(5) \ & \text{push}(\text{pk}(p0, c1), \text{push}(\text{pk}(p1, c1), \text{empty})) \\
(6) \ & \text{push}(\text{pk}(p0, c1), \text{push}(\text{pk}(p1, c2), \text{empty}))
\end{align*}
$$
This example demonstrates the Cartesian product reduction achieved by DDs. Each argument of the main $\Sigma$ DD is a $\Sigma$ DD that represents three terms, the complete encoding is the Cartesian product of these two sets. Thus, nine terms are encoded in total. It also shows how common sets of terms are shared in the structure. The set of terms $\{c_0, c_1, c_2\}$ is encoded only once, but it is shared by both $\Sigma$ DDs at the second level. It is worth mentioning that $\Sigma$ DD rewriting allows to rewrite several terms in one rewriting step. This induces an excellent optimisation of the computational process in favourable cases.

3.5. Symbolic State Space Generation. Before going on, let us define the semantic function $\tau$ more precisely. Let $\text{fire}(t)$ be the function, homomorphic w.r.t. to the union of elements, that fires a transition $t$. Let $+$ be the union of two homomorphisms, $f^*$ be the fixed-point application of a function $f$, and $\text{Id}$ be the identity morphism. We define $\tau$ through homomorphisms on sets of states as 

$$\tau = (\text{fire}(t_1) + \ldots + \text{fire}(t_n) + \text{Id})^*$$

where $T = \{t_1, \ldots, t_n\}$.

As illustrated in Fig. 7, to calculate symbolically the state space $\tau(S_0)$ reachable from a set of initial states $S_0$, it is necessary to compute $\text{enc}_S(\tau(S_0)) = \text{enc}_T(\tau(\text{enc}_S(S_0)))$. Similarly to the overview of $\text{enc}_S$ given earlier, this section briefly introduces the encoding of $\tau$ via the $\text{enc}_T$ function. Additional details can be found in [BH09a, BH09c]. The firing of a transition is encoded by a composition of homomorphisms. Pre- and post-conditions based on closed terms, i.e., those that do not contain variables, are directly encoded as a composition of DD-Homomorphisms prior to runtime during static analysis. Pre/post-conditions with variables, on the other hand, must be bound at runtime.

Let us first encode a transition without variables with the DD framework. A transition can have multiple pre-conditions requesting values from different places. Each pre-condition consumes a multiset $m$ from a place $p$. Let $h_{m,p}$ be the DDHom that computes such a pre-condition. It walks through the graph and for each variable (i.e., DD node) and checks whether the variable corresponds to the place.
If so it also tests whether there are enough tokens in the place. When there are sufficient resources, it generates a new assignment in which it subtracts $m$ from the current multiset contained in $p$. Otherwise, $h_{m,p}^-$ discards the branch by returning the empty DD. The same idea applies to the post-conditions that are only composed of closed terms. The post-condition DDHom $h_{p,m}^+$ walks through the graph until it finds the place $p$ and then it adds the produced tokens $m$ to the current content of the place.

Pre-/post-conditions that contain variables that are still unbound after the pre-processing phase (see Fig. 6) are bound at runtime. For that purpose, a DDHom called $\text{genFiring}$ selects the relevant bindings from the state space and generates a composition of $h^-$ and $h^+$ accordingly. The idea is, at runtime, to boil pre-/post-conditions with variables down to their instantiated version using the selected bindings. Therefore, the general form of the encoding of the firing of a given transition $t$ is:

$$\text{enc}_T(\text{fire}(t)) = h_{m(t,p_i),p_i}^+ \circ \ldots \circ h_{m(t,p_j),p_j}^+ \circ \text{genFiring}_t \circ h_{m(p_k,t),p_k}^- \circ \ldots \circ h_{m(p_n,t),p_n}^-$$

where $p_i, \ldots, p_j$ and $p_k, \ldots, p_n$ are places and $m(p,t)$ (resp. $m(t,p)$) represents the pre-condition (resp. post-condition) of transition $t$ with place $p$.

Let $\text{genFiring}_t = \text{generatePrePost}_t \circ \text{checkGuard}_t \circ \text{selectBindings}_t$ be the DDHom that computes the bindings and generates a composition of $h^-$ and $h^+$:

1. $\text{selectBindings}_t$ walks through the DD and selects bindings candidates for a given pre-condition. Namely, it selects assignments in which the number of tokens is sufficient to satisfy the pre-condition. This produces a new DD containing binding candidates. Fig. 12a shows the bindings candidates for the transition $\text{produce}$ using the marking of Fig. 9. Note that $\text{selectBindings}_t$ does not only consider the bound variables (e.g., $\text{prod}$) but also the free variables (e.g., $\text{sc}$);
2. $\text{checkGuard}_t$ traverses the DD built previously and only keeps the sequence of assignments that do satisfy the guard. Otherwise, it discards the branch by returning the empty DD. As the transition $\text{produce}$ of Fig. 8 has no guard, $\text{checkGuard}_t$ confirms all binding candidates of Fig. 12a.
3. for each pair $\langle \text{variable}, \text{value} \rangle$ of the DD representation of the bindings, $\text{generatePrePost}_t$ generates a $h^-$ accordingly and replaces the value of the variable in the post-condition if necessary. Using the bindings of Fig. 12a, it produces the following composition of DDHoms:

$$\left(h_{\text{encM}([pK(p0,c0)])},\text{PK} \circ h_{\text{encM}([p0])},\text{Prod} \right) + \left(h_{\text{encM}([pK(p1,c0)])},\text{PK} \circ h_{\text{encM}([p1])},\text{Prod} \right)$$

The application of this composition of DDHoms to the state space of Fig. 9 yields a new set of states shown in Fig. 12.
Applying $\text{enc}_T(\text{fire}(\text{produce})) + \text{Id}_{\text{DD}}$ twice to the marking of Fig. 9 produces all states reachable from that marking by only firing the $\text{produce}$ transition. These are shown in Fig. 13. The places that are not impacted by the transition, i.e., $\text{Cons}$, $\text{Pk}_c$, $\text{Buf}$ are shared in the DD.

Instead of generating DDHoms at runtime, an alternative is to use “smarter” versions of $h^+$ and $h^-$ that are able to handle variables. This solution however requires more complex DDHoms that are less efficient and consume more memory. Indeed, a general rule of thumb is that the more complex the DDHom are, the lower the efficiency of the cache will be. Using lot of small and fast DDHom is usually much better than using few complex ones.

The encoding of the semantic function $\tau$ for $T = \{t_1, \ldots, t_n\}$ is $\text{enc}_T(\text{fire}(t_1) + \cdots + \text{fire}(t_n) + \text{Id})^\ast$ which is equivalent to $(\text{enc}_T(\text{fire}(t_1)) + \cdots + \text{enc}_T(\text{fire}(t_n)) + \text{enc}_T(\text{Id}))^\ast$.

For the model in Fig. 8, $\text{enc}_T(\text{fire}(\text{produce}) + \text{fire}(\text{send}) + \text{fire}(\text{receive}) + \text{fire}(\text{consume}) + \text{Id})^\ast = (\text{enc}_T(\text{fire}(\text{produce})) + \text{enc}_T(\text{fire}(\text{send})) + \text{enc}_T(\text{fire}(\text{receive})) + \text{enc}_T(\text{fire}(\text{consume})) + \text{Id}_{\text{DD}})^\ast$.

3.6. **Computing the state space in AlPiNA.** The concepts explained in this section have been implemented in AlPiNA. State space computation is performed by clicking a button in the tool palette (shown in Fig. 4). The tool computes the
state space and returns either the number of states of the system, or an out-of-memory error if the state space is too large. The model checker engine is separated from the interface. It can be started either locally, or on a separate server to take advantage of powerful architectures.

Fig. 14 presents what AlPiNA displays in its console after having generated the state space of Fig. 8. It displays the static analysis, i.e., pre-processing time, the state space generation time, the number of states of the system as well as the number of DD necessary to encode it and finally the amount of memory required for the process. This amount of memory can seem quite large for Decision Diagrams. It comprises memory required for pre-processing, memory to store the Decision Diagrams, as well as computation caches. Note that these caches contain a lot of entries, because hierarchical Decision Diagrams require much more unions and intersections of Decision Diagrams that flat ones. In bigger executions, useless entries of the operation cache are cleaned regularly (using weak references).

3.7. **Symbolic property verification.** Symbolic state space generation has an impact on property verification. Unlike explicit model checking, SMC computes the symbolic complete state space of the system before checking whether properties hold. Consequently, explicit model checkers tend to be more efficient in detecting failures that occur early during the state space exploration, as they immediately stop their computation and report the problem.

Another major difference resides in the symbolic encoding of the state space: SMC checks whether a property holds on a whole set of states instead of checking each state individually. This approach is very efficient if the property involves one or several boolean clauses whose variables are independent, e.g., $x < 3 \land y = 4$. However, it partially loses its efficiency if the variables in the property are not independent, e.g., $x < y$. In this case, the symbolic representation has to be “broken” to check all possible bindings. In other words, the Cartesian product encoded by the Decision Diagram is split to compare all the permutations. Usually such split is a very costly operation. This inefficiency can be partially avoided by

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlPiNA’s model checker started on port 12345.</td>
<td></td>
</tr>
<tr>
<td>Compute State Space...</td>
<td></td>
</tr>
<tr>
<td>Static Analysis Time : 15 ms</td>
<td></td>
</tr>
<tr>
<td>State space generated in : 373 ms</td>
<td></td>
</tr>
<tr>
<td>State Space has been fully generated.</td>
<td></td>
</tr>
<tr>
<td>Total Time : 388 ms</td>
<td></td>
</tr>
<tr>
<td>Number of states : 372</td>
<td></td>
</tr>
<tr>
<td>#DD : 4,343</td>
<td></td>
</tr>
<tr>
<td>Memory (KB) : 3,491</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 14.** State space generation output for the model of Fig. 8
unfolding the domains of the property variables. With this, mutually dependent
variables become independent, e.g., \( x < y \equiv (x = 1 \land y = 2) \lor (x = 1 \land y = 3) \lor \ldots \). Of course, total unfolding is not practical for large domains. This is one
of the optimisations explained in the next section — a pragmatic approach for
unfolding where the user specifies what to unfold.

Figure 15. Console output of property check of the model of Fig. 8
using the property defined in Fig. 16

In \textit{AlPiNA}, properties are expressed using a dedicated language. The satisfaction
of a property \( \phi \) is expressed by \( \models \phi \). The translation homomorphism \( \text{enc}_P(\models \phi) \) en-
codes the property satisfaction checking as a DDHom. This DD-Homomorphism
(DDHom), when applied to the DD representing the state space \( \text{enc}_S(\tau(S_0)) \), re-
turns the set of states that do not satisfy the property. Thus, if the property
holds on the whole state space, it returns an empty DD. This is illustrated in the
lower-right part of Fig. 7:

1. \( \text{enc}_P(\models \phi) \) translates the verification function to a DDHom. Logical con-
nectors in the property are translated to operations on homomorphisms.
   For example, disjunction is converted to the union of two homomorphisms.
2. According to the unfolding provided by the user, the formula checker
   DDHom is unfolded. As explained before, this can improve the property
   checking phase considerably.
The model checker projects the state space according to the variables involved in the property to check. This technique, called the “cone of influence”, may dramatically shrink the size of the DD.

The model checker applies the DDHom to the projection of the state space. This results in a DD that only encodes the bindings that do not satisfy the property.

Finally, the complete state space is explored again to select the states that contain the bindings found in the previous step. These are the states that do not satisfy the property.

In AlPiNA, property checking is performed by clicking a button in the tool palette (see Fig. 4). Fig. 16 presents a property for the Producer/Consumer model, expressed in AlPiNA’s language.

This property checks that there is no state in which the buffer is full. It is easy to verify that it does not hold, and Fig. 15 shows the console’s output from the property checker. It contains the same information as for state space generation, plus the property checking statistics and a counterexample. Note that currently, AlPiNA only returns one of the states where the property does not hold. The markings of places that violate the property are given as a counterexample. The counterexample below is an excerpt of Fig. 15. Only one faulty marking of place Buff is returned.

```
import "prod_consumer.apnmm"
import "Boolean.adt"
import "Buffers.adt"
Expressions
checksize : forall($x in Buff : (isNotFull($x) = true));
Check
@checksize;
Variables
  x : B;
```

**Figure 16.** A property of the Producer/Consumer model in AlPiNA

Property does not hold! Here’s a counterexample:
< Pk_p : []; Pk_c : []; Cons : [c0]; Prod : [p(p0) + p0]; Buff : [push(pk(p(p0), c0),
push(pk(p(p0), c0), push(pk(p(p0), c0), push(pk(p(p0), c0), empty)))] >

To get usable counterexamples, model checkers usually return a trace of the states traversed, from the initial state to a faulty one. Currently there is no easy way to return the trace as only the state space is memorised and not the full transition relation. To solve this problem, we plan to implement backward firing in AlPiNA. This operator has two benefits. First, it is usable to compute traces for counterexamples. Starting from the faulty states, we can build iteratively the set of predecessor states until one of the initial states is met. Iteration ensures that a
shortest path is returned. Moreover, backward firing would bring the computation of CTL properties to \textsc{AlPiNA}.

4. SCALING UP THE MODEL

Model checkers are created to check properties on industrial models. However, these are difficult to obtain: they usually require a joint work between the researchers and domain specialists. As these models can be huge, model checkers have to handle large state spaces. Therefore, before testing the limits of a model checker on an industrial model, one usually tests it with rather small academic models that are parameterised. Increasing their parameters leads to a larger state space, so it is easy to test the model checker by scaling up such models. This brings two requirements:

- the model should be easily configurable. It should be possible to adapt the bounds of the domains used by the model and to change the initial marking easily;
- as the number of the states grows exponentially with the size of the model, the state space generation and the property validation must handle the surge gracefully.

![Figure 17. Expanded view of process “Scaling up the model” of Fig. 1](image)

The first requirement is satisfied by the way systems are represented using HLPNs. Contrary to PTs, HLPNs do not require to add places and transitions to the model in order to increase the number of components. It is sufficient to modify the domain bounds and to change the initial state. \textsc{AlPiNA} provides functions such as definitions by intension to declare huge sets of terms symbolically (as seen in Section 2.2) and easy ways to set the limit of the domains.

The second requirement is related to how the model checker counters the SSE. In \textsc{AlPiNA}, users have a great control on how to optimise the validation. They can affect the state space generation and property validation phases by means of meta-data. Fig. 17 presents the different artifacts the user may produce to scale-up the validation. As shown in Fig. 1, these optimisation meta-data are injected in the model checker during validation. There are three types of information the user can provide:

- **unfolding:** this describes how the model checker handles data types, more specifically whether it should consider the complete domain or a subset of it;
variable ordering: this is a recurrent problem when dealing with DDs as it has a great impact on the efficiency of the symbolic encoding;

clustering: this is a kind of hierarchical ordering.

In ALPiNA, the user can define unfolding, variable ordering and clustering through a dedicated optimisations language. Note that the model checker can also work without optimisations. In this case, none are performed and big models are not handled. This approach enables the user to test and compare easily several optimisations.

The remainder of this section will describe unfolding, ordering and clustering in detail.

4.1. Unfolding. In APNs, arcs are labelled with terms that may contain variables. Hence, as said in Section 3.5, the functions that compute the successors of a set of states have to compute variable bindings. Computing bindings at runtime can be very costly. It implies creating a new DD for each binding, as presented in Fig. 12. The problem is that often the transitions in a Petri net have guards that invalidate some or even most of the available bindings. In this situation many useless DD are created an canonised.

We propose to unfold the domains and build specific homomorphisms for each binding during the static analysis. However, complete unfolding is obviously not viable for infinite (or even very large) domains. Thus, every unfolded domain has to be bound, either structurally or by the user:

- Some domains are intrinsically finite, e.g., an enumeration, or an AADT where a structural analysis proves that the domain is finite. This is called a structural bound.
- If the unfolded domain is not finite, the user must set a bound. This is called a presumed bound. Notice that, even in the case of finite domains, the user can overrule the default bound to set a presumed bound. The presumed bound is interpreted by the system as the number of applications of the non-constant generators on the set of constant generators. For instance, consider the natural numbers described in Example 2.1 and a presumed bound set to 4. In this case, the set of unfolded values has 5 elements: zero (constant generator), suc(zero), suc(suc(zero)), suc(suc(suc(zero))) and suc(suc(suc(suc(zero)))).

The unfolding of a transition encoding goes through three steps:

1. The domains of the input and output variables are unfolded — either completely if a structural bound exists, or up to the presumed bound.
2. The system computes the bindings and only keeps those that satisfy the guard.
3. The computed bindings are used to instantiate a binding-specific version of the transition encoding encr. This is used in the state space generation instead of the original homomorphism.
Using the axioms between generators \((c(c0) = c0\) and \(p^4(p0) = p0\)) of the definitions of Appendix A, the domains \(P\) and \(C\) of example of Fig. 18 are finite (one consumer and four producers).

Therefore, the transition encoding of \(fire\)\((produce)\) is unfolded to Equation (1) but without creating any bindings at runtime and thus without requiring \(genFiring_{produce}\).

To set presumed bounds, users must make a guess at a “good” value for the bound. This guess may have a dramatic impact on the model checking. If the presumed bound is too low, the tool may not explore the state space completely. This makes the results inconclusive. On the other hand, if the presumed bound is too high, then the system will waste CPU time and memory.

Moreover, a lot of the unfolded bindings will not be used during the state space generation:

- many will be discarded because they do not satisfy the transition guards;
- others will not be explored because their pre-conditions are never satisfied.

This means the bound has to be chosen with great care. AlPiNA can partially assist the users in avoiding undesirable unfoldings by using partial unfolding.

4.2. Partial unfolding. In some cases, it may be difficult or not desirable to set a bound for certain domains. This is for example the case when:

1. the maximal used value of a domain is not easily predictable;
2. the domain is a structured type such as lists or buffers. The unfolding of such highly inductive types quickly becomes intractable, especially when lists embed other structured types;
3. the domain is sparse, i.e., only few values are used and they are scattered.

An example could be the domain of the natural numbers when only the values \(\{1, 15257, 1154587\}\) are used.

To assist users, we introduce partial unfolding: an unfolding of the model in which not all domains have to be unfolded. For instance, in the model of Fig. 8, the domain of the buffers should not be unfolded. In such a case, the encoding of the transition function is a mix of unfolded and non-unfolded DDHoms. The choice of what to unfold is a trade-off between the reduction in complexity of the state space generation and the cost of the unfolding operation itself, which is polynomial with respect to the size of the algebras. From a user perspective, the

\[\text{unfolding complexity is } O(n^c)\] where \(n\) is the size of the largest domain and \(c\) the largest number of input arcs.
novel aspect of this feature resides in the fact that for each data type, users can choose whether they want the engine to unfold it (partial net unfolding) and, if so, whether the unfolding should be bound.

Fig. 19 presents the unfolding applied to the AADTs used in Fig. 8 and shown in the appendix. The P, C, K and bool domains are totally unfolded (using the modifier TOTAL). The domain nat of the natural numbers is bound to 5. Finally, the domain of the buffers B is not unfolded.

**ALPiNA** is able, to a certain extent, to prevent combinations that may lead to incomplete state space coverage or infinite computations:

- If a data type is infinite, it cannot be totally unfolded (e.g., B and nat).
- If the engine cannot determine whether an inductive data type is infinite, then a warning against total unfolding is displayed (e.g., P and C).

Although not implemented yet, it is possible to detect that a post-condition generates a value that is greater than its domain bound. In such case, it would be interesting to pause the model checking activity and to ask whether the user wants to raise the domain bound. If so, he could resume model checking with the new bound.

![Figure 19](image)

**Figure 19.** Unfolding of the domains for the model in Fig. 8

### Unfolding

<table>
<thead>
<tr>
<th>Domain</th>
<th>Unfolding</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>TOTAL;</td>
</tr>
<tr>
<td>C</td>
<td>TOTAL;</td>
</tr>
<tr>
<td>K</td>
<td>TOTAL;</td>
</tr>
<tr>
<td>bool</td>
<td>TOTAL;</td>
</tr>
<tr>
<td>nat</td>
<td>5;</td>
</tr>
<tr>
<td>B</td>
<td>NONE;</td>
</tr>
</tbody>
</table>

4.3. **Ordering.** The choice of a variable ordering is a common problem for DDs as it critically impacts the size of the graph. Ordering traditionally refers to the structural description of the DD. This includes the order in which the variables appear in the DD and also their hierarchical organisation. For example, Fig. 20 presents the same state as Fig. 12b but with a different variable ordering: Prod > Cons > Pk_c > Buff > Pk_p for Fig. 20, versus Prod > Pk_p > Cons > Pk_c > Buff for Fig. 12b. It is easy to see that the sharing in Fig. 12b is higher and thus the number of nodes is lower.

![Figure 20](image)

**Figure 20.** The state of Fig. 12b with a different ordering
In general, finding an optimal variable ordering is infeasible. Even checking whether a particular ordering is optimal is NP-Complete [CGP99]. Various heuristics have been developed to find a good variable ordering either statically — *i.e.*, prior to the state space generation — or at runtime — *i.e.*, by reorganising the DD encoding the states on the fly. Most of the efforts on static ordering are centered on the concept of minimum variable span. For a survey on static ordering see [RK08]. It can be observed that usually DDs are more compact when related variables are close to each other in the ordering. For instance, variables representing related places of the APN are usually put together in the DD. This limits the number of nodes impacted by the computation of the transitions' application.

In the example of Fig. 8, places Prod and Pk_p are clearly related as all the transitions that involve Prod also involve Pk_p. Because of this they are near in the ordering. The same can be said of Cons and Pk_c. Place Buff instead is not really linked either to the first or to the second group. It is thus put at the end. Fig. 21 shows how the ordering is defined using AlPiNA’s ordering language. If a place is not mentioned in the ordering, it is automatically put at the end of the DD.

As SDDs and their derivations support hierarchy, the concept of hierarchical ordering naturally comes to mind. This extension is treated in the next paragraph, and is called *clustering* as it organises and manipulates clusters of variables.

**4.4. Clustering.** The ordering discussed in Section 4.3 is flat: there is a total order on the variables. However, consider the case where a transition requires to update variables at the end of the DD. In this case, the algorithm must still walk through all the previous nodes of the DD. Moreover, each modification of a single variable induces the re-canonisation of the entire DD. To solve this particular problem, Couvreur and Thierry-Mieg proposed *clustering* [CTM05]. This, put simply, is a hierarchical ordering. Not only the variables are ordered, but clusters of variables are in their turn ordered hierarchically. There are two advantages to this approach:

- it reduces the size of the state space encoding, because it leverages the Cartesian product effect induced by DDs. Indeed, the state space of a model with independent components is often close to the Cartesian product of the state space of each component;
- it speeds up the state space computation, because actions that are local to a cluster do not impact other clusters.

**4.4.1. Topological clustering.** Clustering was initially developed for PTs. Clustering in a PT can only be based on the topology of places and transitions. Fig. 22
shows an example of topological clustering for the model of Fig. 8. Related parts of the net are clustered together. The Prod and Pk.p places are grouped in the clProd cluster. The Cons and Pk.c places are in the clCons cluster. Finally, Buff, which is not related to other places, is put in a default cluster called clDef.

![Topological clustering of the encoded marking of Fig. 9](image)

Using this clustering, a function that modifies Cons and Pk.c does not affect the other variables, as it only operates in the embedded DD of the clCons cluster. From an operational point of view, homomorphisms that act on clusters are embedded in a localisation homomorphism. The localisation homomorphism that applies a homomorphism $h$ on a cluster $c$ is noted $L_c(h)$. The projection of a homomorphism $\text{enc}_T(\text{fire}(t))$ on a cluster $c$ is noted $\text{proj}_c(\text{enc}_T(\text{fire}(t)))$.

### 4.4.2. Saturation

An inherent issue to SMC is the so-called “peak effect” problem. The state space generation produces many intermediate DDs that are not part of the final state space. For instance, this happens when computing pre-conditions or guards. This consumes a lot of memory and may hamper the state space generation. Saturation as introduced by Ciardo et al. in [CLS01] aims at reducing these intermediate structures by applying events in a more efficient manner. It is empirically several orders of magnitude better than the usual symbolic state space generation, because saturated nodes have better chances to be a part of the final state space. The idea is to compute the local fixed-points, starting from the leaves of the DD and going back to the root while re-saturating the nodes. Whenever a node is modified, the process starts over from the leaves. This dramatically reduces the number of costly re-canonisation operations.

Other saturation techniques have also been proposed. For instance, [GV01] computes iteratively subsets of the reachable states. A smart choice of the subsets leads to smaller intermediary DDs. Even if this technique requires more iterations, it gives good results, as the memory is usually the problem when using DDs. Moreover, memoization usually also improves computation time when sharing is increased.

Another adaptation of saturation to SDD has been defined by Couvreur & Thierry-Mieg [CTM05] and later improved by Hamez et al. [HTMK09]. In this case, because of the hierarchy, it is necessary to distinguish local transitions from synchronisation transitions. Local transitions only impact one cluster, i.e., all the variables used by that transition are within the same cluster. Synchronisation transitions modify several clusters, i.e., the variables are distributed over many
clusters. For instance, in the model of Fig. 8 with the clustering of Fig. 22, the produce transition is local to the \texttt{clProd} cluster. The send transition instead is a synchronisation transition between the \texttt{clDef} and \texttt{clProd} clusters. In essence, DD saturation is based on two principles:

- identifying local transitions;
- computing local fixed-points before the global fixed-point.

Based on these localities, the transition relation is rewritten to saturate the local components before firing the synchronisation transitions. Following [HTMK09], \texttt{AlPiNA} uses a set of rewriting rules based on the localisation homomorphism to rewrite the transition relation automatically. Moreover, if two or more bindings happen to have common parts (and thus to generate similar \(h^-\) or \(h^+\)) their application will be factorized thanks to automatic saturation. This enables to share the effect of the pre- or post-condition of different bindings among different transitions.

Example 4.1. Following the topological clustering of Fig. 22, the transitive closure is rewritten as:

\[
\text{enc}_T(\tau) = (L_{\text{clDef}}(\text{proj}_{\text{clDef}}(\text{enc}_T(\text{fire}(\text{send}) + \text{fire}(\text{receive})))) + L_{\text{clProd}}(\text{proj}_{\text{clProd}}(\text{enc}_T(\text{fire}(\text{produce}) + \text{fire}(\text{send})))) + L_{\text{clCons}}(\text{proj}_{\text{clCons}}(\text{enc}_T(\text{fire}(\text{consume}) + \text{fire}(\text{receive})))) + \text{Id})^* 
\]

4.4.3. Algebraic clustering. As stated previously, clustering for PTs is based on topological information: related places are clustered together. For instance, if a model represents a set of identical processes, each process’s places can be put in a separate cluster. The expressiveness of APNs allows to represent the same structure with fewer places, thus losing this explicit separation between the model components. The same place can contain values for different components. Because of this, clustering in APNs cannot be derived from topological information alone. To support this change of paradigm we propose an extension of the topological clustering approach that also considers the domains of the places, that is called Algebraic clustering.

This approach sees clustering as a function that associates a place and a value to a cluster. In order to exploit the inductive nature of values defined with AADTs, algebraic clustering can use an inductive definition itself. Criteria for choosing clusters are generally based on the structure of the model and the properties to be verified.

In general, the best results are obtained when independent elements are put in separate clusters. A heuristic is to put processes in the same cluster as their internal resources, while resources shared among several processes are put in a separate independent cluster. In \texttt{AlPiNA}, clusters are naturally ordered because they are inductively defined using the \texttt{next} operator. A clustering always contains at least a default cluster. Fig. 23 shows how to apply the previous heuristic on
the model of Fig. 8 using a domain-specific language for clustering. It inductively assigns each producer, starting from p0 to its own cluster along with the packets from that producer. Any combination of place and value that is not explicitly specified using the clustering language is automatically assigned to the default cluster. This is the case for any value in the places Cons, Pk_c and Buff. Fig. 24 shows the resulting encoding for the clProd cluster of Fig. 22.

**Figure 24.** The algebraic clustering of Fig. 23 applied to the clProd topological cluster of Fig. 22

| Clustering | Clusters 2 = cl0; |
| Rules      | cluster of p0 in Prod is cl0; |
|            | cluster of p($p$) in Prod is next(cluster of $p$); |
|            | cluster of pk($p$, $c$) in Pk_p is cluster of $p$; |
| Variables  | p : prod; |
|            | c : cons; |

**Figure 23.** Algebraic clustering for the model of Fig. 8

From a user perspective, choosing the granularity of algebraic clustering is a trade-off between the independence of the components and the size of the cluster. A too fine or too coarse clustering will lead to sub-optimal performance. Still, unlike unfolding, sub-optimal clustering will not hamper complete state space exploration. Fig. 25 details the clustering function defined in Fig. 23. Each line refers to a value and the columns refer to the two places. The light-grey cells represent the first cluster (cl0) and the dark-grey cells represent the second
cluster (cl1). The produce transition is unfolded into two local transitions, one for each cluster. Thus we obtain the following composition of homomorphisms corresponding to the firing of produce:

$$L_{cl0}(h^+_{p0,c0}, p0 \circ h^-_{p0,Prod}) + L_{cl1}(h^+_{p1,c0}, p1 \circ h^-_{p1,Prod})$$

In AlPiNA, optimisation meta-data are clearly separated from the model and specified using a Domain Specific Language (DSL). This separation of concerns helps the user to focus on the model first, and to tweak the validation in a second phase if necessary. This optimisation phase may be difficult for the system designer as it requires to understand the underlying techniques. However, the optimisation meta-data are described using a textual language and hence it is easy to use external tools, such as PNXDD [HKPAE10] to generate and inject them automatically in the engine. It is also possible to automatically infer APNs models and optimisation information from models expressed using DSLs. A DSL that integrates enough meta-data can be transformed to equivalent instances of APNs as well as instances of the optimisation DSL (e.g., clustering, variable ordering, unfolding). For instance, a DSL that integrates the notions of tasks and resources can leverage some of the heuristics presented above (group processes and their resources together). Therefore, the system designer can focus on the model and does not have to learn the arcane knowledge of formal verification.

5. Benchmarks

AlPiNA took part in the Model Checking Contest at the Petri Nets 2011 conference. This contest was organised inside the “Scalable and Usable Model Checking for Petri nets and other models of concurrency” workshop (SUMo2011). Participating tools were compared on several parameterised models, expressed in Place/Transition nets or Symmetric nets. The models are described in their respective articles, referenced in [KLB+11]: Flexible Manufacturing System (FMS), Kanban system, mitogen-activated protein kinase cascade (MAPK), Peterson’s mutual exclusion algorithm, the Dining Philosophers with deadlock, Shared Memory, and Token Ring. For each model, three examinations are performed, state space generation, deadlock detection, and reachability formulae. The Model Checking Contest report [KLB+11] analyses the evolution of the execution time and memory consumption when the parameter of each model increases. It also shows the highest parameter reached by the tools for each model.

In this section, we comment the results of this contest for state space generation. Deadlock detection in AlPiNA depends on the full state space generation. The maximum parameter reached and computation time for deadlocks are similar to the ones of state space generation. AlPiNA did not take part in the reachability formulae examination.

---

4AlPiNA models submitted to the Model Checking contest can be found at: http://alpina.unige.ch/misc/MCC-AlPiNA.zip
We argue the results of the contest are more reliable than home-made benchmarks. The drawback is that we do not compare to tools that did not take part to the contest, for instance **SMART** [CLS00]. During the contest, benchmarks were executed on the same environment for each tool. Each tool developer submitted its own tool, with the best settings for each model. Moreover, the tool developers could enrich models with information used to increase the performance of their tools. For instance, we provided only the models in APNs, together with their clustering and unfolding optimisations.


The compared tools are classified in Fig. 26 according to the model structures they handle and their state space representation. **ACTIVITY_LOCAL** appears as both explicit and DD-based, because it mixes both representations.

<table>
<thead>
<tr>
<th>Static structures</th>
<th>Explicit</th>
<th>DD-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVITY_LOCAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crocodile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITS-Tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PNXDD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YASPA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic structures</th>
<th>Explicit</th>
<th>DD-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helena</td>
<td></td>
<td>AlPiNA</td>
</tr>
</tbody>
</table>

**Figure 26. Classification of the compared tools**

(referenced in [KLB+11])

Static or Dynamic model structures. Some modelling formalisms, like High-level Petri nets and Algebraic Petri nets allow to use dynamic data structures, such as queues, lists, etc. Others, like PTs or Symmetric Petri nets (SNs), only handle static data structures, e.g., vectors, arrays, bounded integers, etc. As all presented tools only generate finite state spaces, dynamic data structures can be converted into static ones by manually setting bounds. However, this may be difficult for the user.

Explicit or DD-based state space representation. We compare here two main types of state space representation. Explicit state space representation stores a set of elements, each representing one state. The state encoding can be non-trivial. The set can also use non-trivial element indexing or ordering.

A DD-based state space is represented by a Decision Diagram. As seen in Section 3.5, operations to compute the state space from the initial state are adapted to the DD structure. They compute the successor states for a whole set of states in one step. Most tools in the benchmark use Decision Diagrams, as it seems to be the most interesting technique for state space computation.

5.1. **Maximum parameter reached.** Fig. 27 shows the highest parameter reached by the competing tools, for each model. It shows one diagram for each model. Every diagram is divided into sectors that represent tools. The size of the radius for a particular tool shows the maximum parameter reached by the tool when doing state space generation, before a time or memory limit is reached. Scale is logarithmic. Dashed circles represent parameter steps (10, 100, ...). The
dotted circles mark the results obtained by tools. For this benchmark, we exclude the Kanban and Shared Memory models. Kanban is very similar to Flexible Manufacturing System, and Shared Memory to the Dining Philosophers.

Figure 27. Highest parameter reached by the tools for state space generation (logarithmic scale)

Generality of AlPiNA. We can remark that PNXDD and AlPiNA are the only two tools that were able to generate the state space for all models. This includes both models that are not shown in Fig. 27. Even if models were restricted to Petri nets with no dynamic structures, the other tools could not handle all of them. YASPA only worked on Place/Transition Petri nets and did not unfold the models given as Symmetric Petri nets. ITS-Tools and ACTIVITY-LOCAL use different formalisms, thus requiring to rewrite the models. They did not provide the conversion for some models.

Efficiency on Place/Transition Petri nets. AlPiNA reaches rather low parameters for Flexible Manufacturing System. Two tools are far less effective, namely Crocodile and helena. Crocodile uses Decision Diagrams, but is not intended for PTs. helena is explicit and its abstractions are disabled for the state space generation, thus the decreased efficiency.

Efficiency on Symmetric Petri nets. On Symmetric Petri nets models, AlPiNA is much more efficient. It has results similar to those of PNXDD on all models. The major difference in the results of these two tools is for Peterson’s algorithm. In this model, AlPiNA reaches parameter 3, whereas PNXDD handles 5. In the other cases, AlPiNA is slightly behind by PNXDD because the latter uses a DD technique that is
still not implemented in \texttt{AlPiNA}, namely anonymisation of the DD variables. Using this technique, all DD nodes are labelled with the same variable. This consistently provides a reduction of the size of the DDs.

\texttt{ITS-Tools} is much more efficient than the other tools on state space generation for SN. It applies a technique called “recursive folding” where the DD hierarchy is combined with anonymisation to get a better sharing in the DD representations of the components in the modelled system. The drawback of this technique is that \texttt{ITS-Tools} requires to completely rewrite the model. The modeller has to design himself a recursively folded model.

The performance of \texttt{helena} vary. It behaves very poorly for highly symmetric systems such as the Dining Philosophers. But it is also as good as \texttt{AlPiNA} for the Token Ring. This model has few states compared to the others. For instance, it has $\sim 10^4$ states for the maximum parameter reached by \texttt{AlPiNA} and \texttt{helena}, whereas the Dining Philosophers have $\sim 10^{238}$ states for the maximum parameter reached by \texttt{AlPiNA}. An explicit model checker cannot represent a state space of this size without reduction techniques, that were disabled for \texttt{helena}.

Comparison between Place/Transition Petri nets and Symmetric Petri nets. All PT models are nonsafe, whereas all SN ones are safe. When compared to \texttt{PNXDD}, this benchmark shows that \texttt{AlPiNA} is currently less efficient than expected on non-safe PNs. Both tools rely on rather similar techniques, so their results should be closer for PT models. The explanation of the poor performance of \texttt{AlPiNA} on PTs is that the MSDDs are only well-adapted to small cardinalities in markings. When cardinalities grow, the MSDDs can do less sharing, and thus generate more DD nodes and more operations on them. We are currently working on their replacement.

From these benchmarks, we see that \texttt{AlPiNA} lacks several interesting DD techniques. Still, \texttt{AlPiNA} is already an efficient state space generator. We are currently working on the integration of the missing techniques with APNs through enhancements of the optimisation language.

5.2. Computation time. Fig. 28 presents the computation time for some models. \texttt{AlPiNA} is executed on the Java Virtual Machine (JVM), with a fixed amount of memory. This is a good configuration for model checkers, which are typically run on a dedicated computer. When the available memory is exhausted, the JVM does a lot of garbage collections and the process does not terminate. Thus, we do not provide memory measurements, and only time is measured with a fixed amount of memory. Note that the Model Checking Contest ran tools with less than 2GB of memory. We also provide time measurements for \texttt{AlPiNA} with 6GB of memory. These measurements have been taken outside of the Model Checking Contest to provide more observations on the behaviour of \texttt{AlPiNA}.

In Fig. 28 we can observe that \texttt{AlPiNA} follows more or less the time curve of \texttt{PNXDD}. On high parameters for the Dining Philosophers, \texttt{AlPiNA} has a greater slope, because is does not implement anonymisation contrary to \texttt{PNXDD}. The difference
for Shared Memory is very high, because \texttt{AlPiNA} uses a bad clustering compared to the one of \texttt{PNXDD}. With more memory, \texttt{AlPiNA} has a smaller slope than \texttt{PNXDD} on the Token Ring. As \texttt{AlPiNA} uses a subset of the techniques used in \texttt{PNXDD}, this difference in slope is caused by a better clustering in \texttt{AlPiNA}. In Shared Memory and Dining Philosophers, \texttt{ITS-Tools} performs far better than the other tools, because it uses recursive folding, a technique the other tools do not implement, but that requires a special encoding of the model.

\texttt{AlPiNA}, as a standalone model checker, has an initial cost of almost 30 seconds (on a rather slow computer). This time is spent in loading all the required libraries, reading the model, validating it and converting it to the internal representation of the model checker. When the model checker is used within the modelling environment, this initial cost is greatly reduced to only a few seconds.

5.3. Conclusion of benchmarks. From these benchmarks, it can be argued that \texttt{AlPiNA} is an efficient state space generator. It handles very well models of distributed systems where most operations are local to each process, i.e., most resources are consumed and produced within the same process. This is not the case in Peterson’s mutual exclusion algorithm.

As for the DD-based tools in this comparison, \texttt{AlPiNA} is comparable to \texttt{PNXDD} when using optimisations. The performance difference with \texttt{PNXDD} for the Dining Philosophers and the Token Ring is due to \texttt{AlPiNA} lacking some optimisations, such as DD anonymisation. Besides, it has a high initial memory footprint ($\approx 400$MB) because of the Eclipse Eclipse Modeling Project (EMP) infrastructure. Future work will bring such improvements to our tool.

\texttt{AlPiNA} is not optimised for Place/Transition Petri nets or Symmetric Petri nets (as \texttt{PNXDD}). On the contrary, it is based on Algebraic Petri nets, which are more expressive than the Petri net classes used in the Model Checking Contest. This expressiveness, that enables \texttt{AlPiNA} to handle structured types that are not bounded
at modelling time, has an overhead during pre-computation (for the unfolding) and during computation. Thus, the rather good results obtained by AlPiNA on SNs are promising.

6. Architecture

One of AlPiNA’s goals is providing a synergy between usability and performance. To provide a good user interface we leverage the Eclipse platform. It provides a well-known user interface model as well as very efficient tools to develop new software. The most natural way to create a model is using a language that is specifically tailored to the domain of the model — a DSL. The Eclipse platform provides several tools to develop DSLs, one of the most relevant being the EMP [Ecl], which follows a meta-modelling approach and provides tools for both the graphical and the textual editors.

The meta-modelling approach allows the integration with other projects that use the same technology. As an example, we are currently working on the integration of the Petri Net Markup Language (PNML) [WK03]. PNML’s goal is to become a standard for defining different types of Petri nets. It can be used as an exchange platform between different PN tools.

As a research project, AlPiNA tries to be as modular as possible in order to support the rapid evolution of technologies as well as new ideas. Fig. 29 describes the layered architecture of AlPiNA. The left block represents the structure of the graphical user interface (GUI). The first layer manages the user interface itself. It leverages the code that has been produced by the tools on the second layer, which presents the meta-modelling tools used in AlPiNA. The concrete syntaxes are defined with EMP.

The right block of Fig. 29 represents AlPiNA’s model checking engine, which performs the actual computations. The first two layers are the property checker and the APN engine. These two layers act as an interface to the engine: the input is an APN and some properties to be checked; the output is the result of the property checks and some information about the APN’s state space, such as the

![AlPiNA’s architecture](image-url)
computation time and the number of states. These two layers are based on the DDs, used to calculate and represent the state space and properties check.

Communication between the blocks of Fig. 29 is done through Java Remote Method Invocation (RMI). This ensures a strong independence between the GUI and the engine, and allows experienced users to extend the tool easily. Both the interface and the engine can be substituted by different components or re-used in other projects.

7. Conclusion and perspectives

This article provided an overview of AlPiNA [BHMR10], an Algebraic Petri nets model checker. We showed its user-friendly interface and its modelling and optimisation languages. We also explored the main model checking techniques used in AlPiNA:

- Symbolic techniques based on $\Sigma$DDs, a new variant of Decision Diagrams. We explained their power especially with respect to sharing of information;
- Advanced optimisations, namely ordering, unfolding (partial or total) and clustering (topological and algebraic). They are specified in a clear and abstract way by means of dedicated languages. This also accomplishes a clear separation between the model and the optimisations.

Benchmarks from the Model Checking Contest [KLB+11] at the Petri Nets 2011 conference show that AlPiNA clearly outperforms explicit model checkers for state space generation. Its performance is comparable to other symbolic model checkers. Even if there is ongoing work on AlPiNA’s optimisations, its execution time curves are similar to the ones of tools that use DDs in state space generation of Petri nets. This is a good result, as AlPiNA manages more expressive models, expressed in Algebraic Petri net.

AlPiNA is under active development. There is ongoing work on the following improvements:

- support for modular, object-oriented and hierarchical [BG00] models, as they simplify the modelling activity. Moreover, optimisation meta-data will be automatically extracted from such richer models. Symmetries inherent to object-oriented models can be automatically discovered and used for further optimisation;
- support for CTL. Currently, AlPiNA verifies invariant properties, which are enough for many industrial applications. CTL will enable to check properties on execution trees rather than states;
- improvement of the property checking phase. Encoding the state space using DDs presents new challenges for checking properties. We will improve this using techniques such as parallelisation;
- application of the Domain Specific Language approach to model checking. We propose in [SBH+11] a language dedicated to the description of gene
regulatory networks. This DSL covers both the network model and properties to check, for instance the presence of a given molecule. Work on this language continues to extract optimisations from the gene regulatory network.

We actively pursue the use of AlPiNA in a concrete environment. For this, we will combine it with our previous work on DSML definition [Ris10], testing [BBP96] and prototyping [ASBB+03] to achieve automatic test case generation [BLC09] and verification of DSML languages [PRBA10].

AlPiNA, its source code and meta-models are available under the GPL license as an Eclipse plug-in or a complete Eclipse package at http://alpina.unige.ch.

REFERENCES


Appendix A. The AADTs used by the model of Fig. 8

import "Boolean.adt"
import "Naturals.adt"
import "Packets.adt"

Adt Buffers
Sorts B;

Generators
empty : B;
push : K, B -> B;

Operations
size : B -> nat;
head : B -> K;
tail : B -> B;
isFull : B -> bool;
isEmpty : B -> bool;

Axioms
size(empty) = zero;
size(push($p, $b)) = suc(size($b));
//head(empty) and tail(empty) undefined.
head(push($p, $b)) = $p;
tail(push($p, $b)) = $b;
//Buffer with capacity 4, full if it has strictly more than 3 elements
isFull($b) = gt(size($b),suc^3(zero));
isEmpty(empty) = true;
isEmpty(push($p, $b)) = false;

Variables
p : K;
b : B;

import "Consumers.adt"
import "Producers.adt"

Adt Packets
Sorts K;

Generators
pk : P, C -> K;

Operations
getProd : K -> P;
getCons : K -> C;

Axioms
getProd(pk($p, $c)) = $p;
getCons(pk($p, $c)) = $c;

Variables
p : P;
c : C;

Adt Consumers
Sorts C;

Generators
c0 : C;
c : C -> C;

Axioms
c(c0) = c0; //1 consumer

Adt Producers
Sorts P;

Generators
p0 : P;
p : P -> P;

Axioms
p^4(p0) = p0; //4 producers