Predicate diagrams for the verification of reactive systems *

Dominique Cansell[1], Dominique Méry[2], and Stephan Merz[3]

[1] Université de Metz & LORIA, cansell@loria.fr
[2] Université Henri Poincaré & LORIA, mery@loria.fr
[3] Institut für Informatik, Universität München, merz@informatik.uni-muenchen.de

Abstract. We define a class of diagrams that represent abstractions of—possibly infinite-state—reactive systems described by specifications written in temporal logic. Our diagrams are intended as the basis for the verification of both safety and liveness properties of such systems. Non-temporal proof obligations establish the correspondence between the original specification and the diagram, whereas model checking can be used to verify properties over finite-state abstractions. We describe the use of abstract interpretation techniques to generate proof diagrams from a given specification and user-defined predicates that represent sets of states.

1 Introduction

Verification techniques for reactive systems are traditionally classified as either deductive or algorithmic. Whereas deductive verification can in principle establish properties of arbitrarily complex systems, algorithmic verification such as model checking is usually restricted to finite-state systems, although some classes of infinite-state systems have been identified for which certain properties are decidable. On the other hand, algorithmic verification is (essentially) automatic and, hence, easy to use by system engineers, whereas computer support for deductive verification in the form of interactive proof assistants requires careful guidance by experienced users. Combinations of both kinds of technique such as deductive model checking have also been suggested, with the aim of providing a unified framework that combines the advantages of deductive and algorithmic verification. More generally, abstraction and composition have been recognized as the key concepts that bridge the two approaches [13, 9]. Our paper proposes the concept of *predicate diagrams* to integrate algorithmic and deductive verification techniques; it allows to relate *specifications* and *properties* by constructing an abstract finite model. We consider safety as well as liveness properties by annotating diagrams with information relating to fairness and ordering conditions. We also show how predicate diagrams can be constructed semi-automatically, based on techniques of abstract interpretation.
Predicate diagrams intend to help users to understand how a reactive system (or a distributed system) is working. We have focused on defining a framework that is as powerful as traditional deductive verification and can integrate safety, fairness, and general

* This work has been partially supported by a PROCOPE grant from EGIDE and DAAD.

liveness properties, yet can abstractly represent the given system at a level that is intuitive to understand for the system designer. We have applied predicate diagrams to some case studies such as a reader-writer algorithm developed at Siemens Corporate Research [11] and a self-stabilizing algorithm due to Dijkstra [26]. On the other hand, an *abstract animator* [4] based on techniques of abstract interpretation has been shown to allow models for non-trivial systems to be built automatically once the user has defined the relevant abstract predicates. Our method can therefore be seen as a natural generalization of traditional model checking, except that it is applied to an abstract state space. In particular, there is no need for the user to provide inductive invariants as in deductive verification.

We assume the system to be modeled as a specification written in a variant of TLA [17] and we use the temporal language of linear-time temporal logic for stating required assertions or properties. Nevertheless, the basic ideas that underly our approach apply more widely. We define verification conditions that ensure that a predicate diagram adequately represents a system specification; these verification conditions must in general be discharged using an automatic or interactive theorem prover. The second task is to show that every run of the diagram satisfies the asserted properties. Because predicate diagrams are finite, this can be established using standard model checkers such as Spin [12], obviating the need for tedious temporal reasoning that plagues standard deductive verification. We have shown that the constructions of the diagram can often be automated using abstract interpretation. The user need then only choose the predicates that define the abstraction function. Besides their use as a formal basis for verification, predicate diagrams can also serve as support for explaining how systems are working and for documenting them.

Using diagrams to simplify, to visualize or to structure a proof is not new. There are many works like *proof diagrams* proposed by Manna et al [7, 22, 23], *predicate-action diagrams* from Lamport [18], *proof lattices* proposed by Lamport and Owicki [27] and *proof charts* proposed by R. Cousot [5].

The contributions of our work are the following:

- We emphasize a clear separation of concerns and identify those subtasks that are within the realm of, respectively, theorem proving and model checking.
- Following ideas present in [25, 14], edges of a predicate diagram can be labelled by well-founded orderings to exclude runs where such an edge is taken infinitely often; we can thus prove liveness properties (these annotations simulate the lattice rule of temporal logic).
- Fairness assumptions about transitions can be stated on the level of diagrams; these assumptions are taken into account when proving liveness properties using model checking.
- We propose an automatic construction of a predicate diagram by using abstract interpretation. This construction may require approximations introduced via *maybe* edges, and we propose solutions to eliminate such *maybe* edges via a reconsideration of the corresponding transitions at the concrete level.

The paper is structured as follows. Section 2 presents predicate diagrams. Section 3 describes the relationship between specifications and predicate diagrams. Sections 4 and 5 explain how we use abstract interpretation to construct a predicate diagram for a given specification. Sections 6 and 7 compare with related works and conclude the paper.

2 Predicate Diagrams

We express system specifications and properties in a variant of linear-time temporal logic whose formulas are built from *state predicates* and *actions*, which may contain primed state variables. For example, $x > 3$ is a state predicate, and $x \leq y'+1$ is an action. For an action A, we denote by ENABLED A the state predicate obtained from A by existential quantification over the primed state variables. For a state predicate P, we denote by P' the action obtained from P by replacing all flexible variables v by v'. Temporal formulas are formed using boolean connectives, the *always* operator $\Box$, and quantification over rigid (state-independent) variables.
The semantics of state formulas is defined with respect to a *state*, i.e. an assignment of values to state variables, and a valuation of the rigid variables. Actions are interpreted relative to a pair (s, t) of states, where s and t interpret respectively the unprimed and primed state variables. Temporal formulas are evaluated over *behaviors*, which are ω-sequences $\sigma = s_0 s_1 \ldots$ of states [15,24]. We write $s \models P$, $(s, t) \models A$, and $\sigma \models F$ to denote that a state predicate P, an action A or a temporal formula F hold of a state, a pair of states or of a behavior. Validity of a temporal formula F (over the class of first-order interpretations of interest) is denoted by $\models F$, and similarly for state predicates and actions.
Derived connectives include the *eventually* operator defined by $\Diamond F \equiv \neg\Box\neg F$, the *leads-to* operator $F \leadsto G \equiv \Box(F \Rightarrow \Diamond G)$ and, for an action A, the formulas

$$\mathrm{WF}(A) \equiv (\Diamond\Box\, \textsc{enabled}\ A) \Rightarrow \Box\Diamond A$$
$$\mathrm{SF}(A) \equiv (\Box\Diamond\, \textsc{enabled}\ A) \Rightarrow \Box\Diamond A$$

that assert weak and strong fairness conditions for A. This logic is similar to Lamport's Temporal Logic of Actions [17], but it is not invariant under stuttering.[1] As in TLA, specifications of reactive systems can be expressed as temporal formulas; these are usually written in the form $Init \land \Box Next \land L$ where $Init$ is a state predicate characterizing the system's initial state, $Next$ is an action that represents the next-state relation, and L is a conjunction of formulas $\mathrm{WF}(A)$ or $\mathrm{SF}(A)$.
We assume the underlying assertion language to contain a finite set $\mathcal{O}$ of binary relation symbols $\prec$ that are interpreted by well-founded orderings. For $\prec \in \mathcal{O}$, we denote by $\preceq$ its reflexive closure. We write $\mathcal{O}^=$ to denote the set of relation symbols $\prec$ and $\preceq$, for $\prec$ in $\mathcal{O}$.
The definition of predicate diagrams is relative to finite sets $\mathcal{P}$ and $\mathcal{A}$ that contain the state predicates and actions of interest. We denote by $\overline{\mathcal{P}}$ the set containing the predicates in $\mathcal{P}$ and their negations.

Definition 1. *A* predicate diagram $G = (N, I, \delta, o, \zeta)$ *over* $\mathcal{P}$ *and* $\mathcal{A}$ *consists of*

[1] Stuttering invariance is unimportant for the verification of some fixed specification. It becomes essential when predicate diagrams are used for the top-down development of reactive systems by stepwise refinement, a topic to be described in a separate paper.

- *a finite set* $N \subseteq 2^{\overline{\mathcal{P}}}$ *of* nodes,
- *a finite set* $I \subseteq N$ *of* initial nodes,
- *a family* $\delta = (\delta_A)_{A \in \mathcal{A}}$ *of relations* $\delta_A \subseteq N \times N$,
- *a family* $o = (o_A)_{A \in \mathcal{A}}$ *of edge labellings* o_A *that associate finite sets* $\{(t_1, \prec_1), \ldots, (t_k, \prec_k)\}$ *of terms* t_i *paired with a relation* $\prec_i \in \mathcal{O}^=$ *with the edges* $(n, m) \in \delta_A$,
- *a mapping* $\zeta : \mathcal{A} \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ *that associates a fairness condition with every action in* $\mathcal{A}$*; the possible values represent no fairness, weak fairness, and strong fairness.*

We say that the action $A \in \mathcal{A}$ can be taken *at node* $n \in N$ *iff* $(n, m) \in \delta_A$ *holds for some* $m \in N$.

A predicate diagram G is a finite labelled transition system whose nodes are sets of state predicates from $\mathcal{P}$ or their negations. Intuitively, a node represents the set of system states that satisfy the formulas contained in the node. (In the following, we indifferently write n for the set and the conjunction of its elements.) Transitions are labelled by actions and by ordering annotations that indicate which terms decrease with respect to relations from $\mathcal{O}^=$. An action A may have an associated fairness condition; it applies to all transitions in δ_A rather than to individual edges.
We now define traces through a diagram as behaviors that correspond to fair runs satisfying the node and edge labels. To evaluate the fairness conditions, we identify the enabling condition of an action $A \in \mathcal{A}$ with the existence of A-labelled edges at a given node.

Definition 2. *Let* $G = (N, I, \delta, o, \zeta)$ *be a predicate diagram over* $\mathcal{P}$ *and* $\mathcal{A}$*. The set* $tr(G)$ *of* traces through G *consists of all* ω*-sequences* $\sigma = s_0 s_1 \ldots$ *of states such that there exist sequences* $n_0 n_1 \ldots$ *and* $A_0 A_1 \ldots$ *of nodes* $n_i \in N$ *and actions* $A_i \in \mathcal{A}$ *such that all of the following conditions hold:*

- $n_0 \in I$ *is an initial node,*
- $(n_i, n_{i+1}) \in \delta_{A_i}$ *holds for all* $i \in \mathbb{N}$,
- $s_i \models n_i$ *holds for all* $i \in \mathbb{N}$,
- $(s_i, s_{i+1}) \models A_i$ *holds for all* $i \in \mathbb{N}$,
- $(s_i, s_{i+1}) \models t' \prec t$ *holds for all* $i \in \mathbb{N}$ *and* $(t, \prec) \in o_{A_i}(n_i, n_{i+1})$,
- *for every action* $A \in \mathcal{A}$ *such that* $\zeta(A) = \mathrm{WF}$ *there are infinitely many* $i \in \mathbb{N}$ *such that either* $A_i = A$ *or* A *cannot be taken at* n_i*, and*
- *for every action* $A \in \mathcal{A}$ *such that* $\zeta(A) = \mathrm{SF}$*, either* $A_i = A$ *holds for infinitely many* $i \in \mathbb{N}$ *or there are only finitely many* $i \in \mathbb{N}$ *such that* A *can be taken at* n_i.

3 Reasoning about systems via predicate diagrams

3.1 Relating specifications and predicate diagrams

We say that a predicate diagram G *conforms* to a specification *Spec*, written $Spec \trianglelefteq G$, if every behavior that satisfies *Spec* is a trace through G. In general, proving $Spec \trianglelefteq G$ requires reasoning about entire behaviors. The following theorem allows us to (essentially) reduce this temporal reasoning to "local" proof obligations that concern single states or pairs of states.

Theorem 3. *Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram over $\mathcal{P}$ and $\mathcal{A}$, and $Spec \equiv Init \wedge \Box Next \wedge L$ be a system specification. If all of the following conditions hold, then $\sigma \in tr(G)$ holds for all models σ of Spec.*

1. $\models Init \Rightarrow \bigvee_{n \in I} n$
2. $\models n \wedge Next \Rightarrow \bigvee_{\{(A,m):(n,m)\in\delta_A\}} A \wedge m'$ *holds for every node $n \in N$.*
3. $\models n \wedge A \wedge m' \Rightarrow t' \prec t$ *holds for all $A \in \mathcal{A}$, $(n, m) \in \delta_A$, and $(t, \prec) \in o_A(n, m)$.*
4. *For every action $A \in \mathcal{A}$ such that $\zeta(A) \neq$ NF:*
   (a) *If $\zeta(A) =$ WF then $\models Spec \Rightarrow$ WF(A).*
   (b) *If $\zeta(A) =$ SF then $\models Spec \Rightarrow$ SF(A).*
   (c) $\models n \Rightarrow$ ENABLED A *holds whenever A can be taken at node n.*
   (d) $\models n \wedge A \Rightarrow \neg m'$ *holds for all $n, m \in N$ such that $(n, m) \notin \delta_A$.*

Proof. Assume that *Spec* and G are such that all the conditions hold, and assume that $\sigma = s_0 s_1 \ldots$ is a behavior that satisfies *Spec*. In order to show that $\sigma \in tr(G)$, we inductively define a sequence $n_0 n_1 \ldots$ of nodes $n_i \in N$ and a sequence $\mathcal{A}_0 \mathcal{A}_1 \ldots$ of sets of actions $\emptyset \neq \mathcal{A}_i \subseteq \mathcal{A}$ such that for all $i \in \mathbb{N}$ the following conditions hold:

(i) $s_i \models n_i$,
(ii) $n_0 \in I$,
(iii) $\mathcal{A}_i = \{A \in \mathcal{A} : (n_i, n_{i+1}) \in \delta_A$ and $(s_i, s_{i+1}) \models A\}$, and
(iv) $(s_i, s_{i+1}) \models t' \prec t$ for all $A \in \mathcal{A}_i$ and all $(t, \prec) \in o_A(n_i, n_{i+1})$.

Obviously, conditions (i)–(iv) ensure the conditions stated in definition 2 for any choice of $A_i \in \mathcal{A}_i$, except for the conditions related to fairness requirements.
For the induction base, we choose some node $n_0 \in I$ such that $s_0 \models n_0$ holds; the existence of some such node is ensured by condition (1), since $s_0 \models Init$ holds by assumption.
For the induction step, we assume that $n_0 \ldots n_i$ and $\mathcal{A}_0 \ldots \mathcal{A}_{i-1}$ have already been defined such that conditions (i) and (ii) hold for all $j \leq i$ and conditions (iii) and (iv) hold for all $j < i$. In particular, we have $s_i \models n_i$. Moreover, since $\sigma \models Spec$, we know that $(s_i, s_{i+1}) \models Next$, and condition (2) ensures that $(s_i, s_{i+1}) \models A \wedge m'$ holds for some action A and some transition $(n_i, m) \in \delta_A$. Define $n_{i+1} \triangleq m$ for some such transition, and choose $\mathcal{A}_i$ as given in the right-hand side of condition (iii). These choices imply that $s_{i+1} \models n_{i+1}$ and that $\mathcal{A}_i \neq \emptyset$. To prove condition (iv), assume that $A \in \mathcal{A}_i$ and that $(t, \prec) \in o_A(n_i, n_{i+1})$. By induction hypothesis and the choices of n_{i+1} and $\mathcal{A}_i$ we have $(s_i, s_{i+1}) \models n_i \wedge A \wedge n'_{i+1}$ and $(n_i, n_{i+1}) \in \delta_A$, hence condition (3) implies $(s_i, s_{i+1}) \models t' \prec t$ as required.
It remains to pick a sequence $A_0 A_1 \ldots$ of actions $A_i \in \mathcal{A}_i$ such that the last two conditions from definition 2, which concern the fairness annotations in G, are satisfied. Choose a sequence such that every action $A \in \mathcal{A}$ that appears in infinitely many $\mathcal{A}_i$ is chosen infinitely often. (Such a choice is possible because $\mathcal{A}$ is finite.)
Now assume that $\zeta(A) =$ WF for some action $A \in \mathcal{A}$ and that for all but finitely many $i \in \mathbb{N}$, $(n_i, m) \in \delta_A$ holds for some $m \in N$ (otherwise the condition is obviously satisfied). Because we already know that $s_i \models n_i$ holds for all $i \in \mathbb{N}$, condition (4c)

implies that $s_i \models \text{ENABLED } A$ holds for all but finitely many $i \in \mathbb{N}$. Moreover, since $\sigma \models \mathit{Spec}$ and $\models \mathit{Spec} \Rightarrow \text{WF}(A)$ (by condition (4a)), it follows that $(s_i, s_{i+1}) \models A$ holds for infinitely many $i \in \mathbb{N}$. For every such $i \in \mathbb{N}$, we have $(n_i, n_{i+1}) \in \delta_A$: otherwise, we would obtain $s_{i+1} \models \neg n_{i+1}$ from condition (4d) and have a contradiction. But then, condition (iii) above implies that $A \in \mathcal{A}_i$ holds for infinitely many $i \in \mathbb{N}$, hence $A_i = A$ holds for infinitely many $i \in \mathbb{N}$, which completes the proof.
For actions $A \in \mathcal{A}$ such that $\zeta(A) = \text{SF}$, the proof is analogous, replacing "all but finitely many" by "infinitely many" and using (4b) instead of (4a). □

3.2 Model checking predicate diagrams

Since predicate diagrams are finite transition systems, it is straightforward to encode their runs in the input language of standard LTL model checkers such as Spin [12] or STeP [21]. We briefly sketch our encoding of predicate diagrams in Promela, the modelling language of Spin. Two variables are used to indicate the current node and the last action taken. The predicates in $\mathcal{P}$ are represented by boolean variables, which are updated according to the label of the current node (nondeterministically if that label contains neither P nor $\neg P$). We also add variables $b_{(t,\prec)}$, for every term t and relation $\prec \in \mathcal{O}$ such that $(t, \prec)$ appears in some ordering annotation o_A. These variables are set to 2 if the last transition taken is labelled by $(t, \prec)$, to 1 if it is labelled by $(t, \preceq)$, and to 0 otherwise.
The fairness conditions can be stated as temporal logic assumptions when properties are verified of the Promela model. In order to do so, we assume that action A is enabled whenever the currently active node has an outgoing edge in δ_A, as asserted by condition (4c) of theorem 3. To capture the effect of the ordering annotations, we add the assumptions

$$\Box\Diamond(b_{(t,\prec)} = 2) \Rightarrow \Box\Diamond(b_{(t,\prec)} = 0)$$

for every variable $b_{(t,\prec)}$. These assumptions ensure that transitions known to strictly decrease t with respect to $\prec$ cannot be taken infinitely often unless infinitely often some transitions are taken that may increase the value of t.
It is easy to see that whenever $\mathit{Spec} \trianglelefteq G$ holds, then any property verified for the Promela encoding of G also holds of Spec. We can therefore use model checking to exhaustively consider the traces of G in order to establish temporal logic properties of Spec. In combination with the non-temporal proof obligations stated in theorem 3 to establish the correctness of the abstraction, we thus achieve the desired separation of concerns for the entire verification of Spec. Obviously, the size of diagrams for which model checking is feasible is mostly limited by the number of fairness conditions and ordering annotations that appear in the diagram.

3.3 Two examples

We illustrate the use of predicate diagrams at the hand of two examples: a two-process version of Lamport's Bakery algorithm and the "dining mathematicians" mutual-exclusion protocol. We present the specifications as they are input to the generator of predicate diagrams described in section 4, using the syntax of the specification language

TLA$^+$ [19, 20]. Specifications can consist of several modules that contain variable declarations and operator definitions. Formula *Spec* in the top-level module is the main specification.

The Bakery algorithm. We consider a two-process version of Lamport's well-known protocol [16], shown in figure 1. Mutual exclusion between two processes is ensured via "ticket" variables y_1 and y_2 that processes draw when they wish to enter the critical section. The variables pc_1 and pc_2 represent the control states of the processes (UNCHANGED x abbreviates $x' = x$).

MODULE *Bakery*

VARIABLES $y_1,\ y_2,\ pc_1,\ pc_2$

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & y_1 = 0 \land y_2 = 0 \land pc_1 = \text{“l1”} \land pc_2 = \text{“m1”} \\
L1 & \triangleq & pc_1 = \text{“l1”} \land pc_1' = \text{“l2”} \land \text{UNCHANGED } \langle y_1, y_2, pc_2 \rangle \\
L2 & \triangleq & pc_1 = \text{“l2”} \land pc_1' = \text{“l3”} \land y_1' = y_2 + 1 \land \text{UNCHANGED } \langle y_2, pc_2 \rangle \\
L3 & \triangleq & \land\ pc_1 = \text{“l3”} \land (y_2 = 0 \lor (y_1 \leq y_2)) \\
 & & \land\ pc_1' = \text{“l4”} \land \text{UNCHANGED } \langle y_1, y_2, pc_2 \rangle \\
L4 & \triangleq & pc_1 = \text{“l4”} \land pc_1' = \text{“l5”} \land \text{UNCHANGED } \langle y_1, y_2, pc_2 \rangle \\
L5 & \triangleq & pc_1 = \text{“l5”} \land pc_1' = \text{“l1”} \land y_1' = 0 \land \text{UNCHANGED } \langle y_2, pc_2 \rangle \\
M1 & \triangleq & pc_2 = \text{“m1”} \land pc_2' = \text{“m2”} \land \text{UNCHANGED } \langle y_1, y_2, pc_1 \rangle \\
M2 & \triangleq & pc_2 = \text{“m2”} \land pc_2' = \text{“m3”} \land y_2' = y_1 + 1 \land \text{UNCHANGED } \langle y_1, pc_1 \rangle \\
M3 & \triangleq & \land\ pc_2 = \text{“m3”} \land (y_1 = 0 \lor \neg(y_1 \leq y_2)) \\
 & & \land\ pc_2' = \text{“m4”} \land \text{UNCHANGED } \langle y_1, y_2, pc_1 \rangle \\
M4 & \triangleq & pc_2 = \text{“m4”} \land pc_2' = \text{“m5”} \land \text{UNCHANGED } \langle y_1, y_2, pc_1 \rangle \\
M5 & \triangleq & pc_2 = \text{“m5”} \land pc_2' = \text{“m1”} \land y_2' = 0 \land \text{UNCHANGED } \langle y_1, pc_1 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Next} & \triangleq & \lor\ L_1\ \ \lor L_2\ \ \lor L_3\ \ \lor L_4\ \ \lor L_5 \\
 & & \lor\ M_1\ \lor M_2\ \lor M_3\ \lor M_4\ \lor M_5
\end{array}
$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box\, \mathit{Next}$$

Fig. 1. System specification of the Bakery algorithm.

Consider the predicate diagram G shown in figure 2. We cannot show $\mathit{Spec} \trianglelefteq G$ for the Bakery specification of figure 1 using theorem 3 because the attempt to prove obligation (2) would require proving

$$
\begin{array}{l}
\land\ \neg(pc_1 = \text{“l4”} \land pc_2 = \text{“m4”}) \\
\land\ pc_2 = \text{“m3”} \land (y_1 = 0 \lor y_2 < y_1) \\
\land\ pc_2' = \text{“m4”} \land \text{UNCHANGED } \langle y_1, y_2, pc_1 \rangle \\
\Rightarrow \neg(pc_1 = \text{“l4”} \land pc_2 = \text{“m4”})
\end{array}
$$

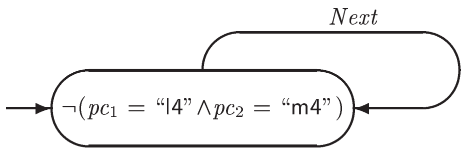


Fig. 2. Predicate diagram for mutual exclusion.

In section 4 we explain how we can construct semi-automatically a predicate diagram for the Bakery specification from which one can prove mutual exclusion as well as the desired liveness properties of the Bakery algorithm.

The “dining mathematicians” example. This is another two-process mutual exclusion protocol introduced in [6]. Processes alternate between “thinking” and “eating” states; they may eat only if n is even or odd, respectively. The temporal logic specification of the protocol appears in figure 3.

MODULE *DM*

EXTENDS *Natural*
VARIABLES $n,\ c_0,\ c_1$

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & n \in \mathit{Nat} \wedge n > 0 \wedge c_0 = \text{“t”} \wedge c_1 = \text{“t”} \\
\mathit{Eat}_0 & \triangleq & c_0 = \text{“t”} \wedge \mathrm{even}(n) \wedge c_0' = \text{“e”} \wedge n' = n \wedge c_1' = c_1 \\
\mathit{Think}_0 & \triangleq & c_0 = \text{“e”} \wedge c_0' = \text{“t”} \wedge n' = n \text{ div } 2 \wedge c_1' = c_1 \\
\mathit{Eat}_1 & \triangleq & c_1 = \text{“t”} \wedge \neg\mathrm{even}(n) \wedge c_1' = \text{“e”} \wedge n' = n \wedge c_0' = c_0 \\
\mathit{Think}_1 & \triangleq & c_1 = \text{“e”} \wedge c_1' = \text{“t”} \wedge n\ = 3{\cdot}n{+}1 \wedge c_0' = c_0 \\
\mathit{Next} & \triangleq & \mathit{Eat}_0 \vee \mathit{Think}_0 \vee \mathit{Eat}_1 \vee \mathit{Think}_1 \\
\mathit{var} & \triangleq & \langle n,\ c_0,\ c_1 \rangle \\
\mathit{Spec} & \triangleq & \mathit{Init} \wedge \Box\, \mathit{Next}
\end{array}
$$

Fig. 3. System specification of the dining mathematicians protocol.

Let $\mathit{DiningGraph}$ be the predicate diagram shown in figure 4. It is an easy exercise to show that $DM.\mathit{Spec} \trianglelefteq \mathit{DiningGraph}$ by an application of theorem 3. Using the encoding of predicate diagrams for Spin described in section 3.2, we can establish the following properties:

$$
\begin{array}{llll}
(\mathit{Pos}) & \Box(n \in \mathit{Nat} \wedge n > 0) & (\mathit{Excl}) & \Box\neg(c_0 = \text{“e”} \wedge c_1 = \text{“e”}) \\
(\mathit{Live}_0) & \Box\Diamond(c_0 = \text{“e”}) & (\mathit{Live}_1) & \Box\Diamond(c_1 = \text{“e”})
\end{array}
$$

The invariants assert that n remains a positive natural number throughout any run of specification DM, and that mutual exclusion is ensured. The other properties are liveness properties and show starvation-freedom for both processes. In particular, property

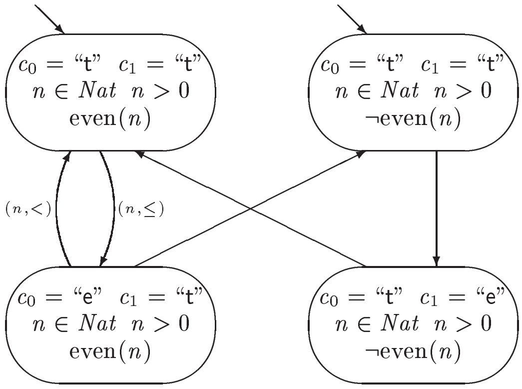


Fig. 4. A predicate diagram for the dining mathematicians.

$(Live_1)$ can be established despite the cycle between the two leftmost states in the diagram, because the ordering annotations forbid that cycle to be followed indefinitely.

4 Using abstract interpretation to construct predicate diagrams

Theorem 3 can be used to justify that a given predicate diagram conforms to a given specification, but it may generate a number of proof obligations that is quadratic in the number of nodes. It is therefore desirable to construct predicate diagrams semi-automatically from a specification whenever this is possible.

In [4] we have defined an "abstract animator" of temporal logic specifications that constructs behaviors of system specifications where concrete values may be abstracted via a user-defined abstraction function. Successor states are computed by abstractly evaluating the system's next-state relation w.r.t. the abstract values at the previous state. If the abstract evaluation succeeds for all reachable states, we obtain a finite approximation of the system's behaviors that can be used to infer system invariants. The implementation of our prototype was based on the rewrite engine Logic-Solver that underlies Atelier B [29], an environment supporting the B method [1]. We have since extended the functionality of our tool in order to produce predicate diagrams that conform to the given system specification.

Traditionally, the relationship between concrete and abstract states that underlies abstract interpretation is described by a Galois connection. We recall that a Galois connection between two partially ordered sets $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ is given by two mappings

$$(L_1, \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L_2, \sqsubseteq_2)$$

such that $x_1 \sqsubseteq_1 \gamma(x_2)$ holds iff $\alpha(x_1) \sqsubseteq_2 x_2$, for all $x_1 \in L_1$ and $x_2 \in L_2$. In our setting, elements of L_1 are sets of states and L_2 is the Boolean algebra $\mathcal{B}(\mathcal{P})$ whose atoms are the predicates in $\mathcal{P}$. The abstraction function α returns the set of predicates true or false of a set of states, and the concretization function γ produces the set of states that are models of a set of (negated) predicates.

Given an abstract state $n \in \mathcal{B}(\mathcal{P})$, the main problem is to compute an abstract representation of the set of successor states of the states in $\gamma(n)$ with respect to some action A (including the next-state relation $Next$) of the given specification.

Definition 4. *For $m, n \in \mathcal{B}(\mathcal{P})$ and an action $A \in \mathcal{A}$, we say that n is an* abstract successor *of m iff for all states s, t, if $s \in \gamma(m)$ and $(s,t) \models A$, then $t \in \gamma(n)$.*

Because we may alternately interpret abstract states as elements of $\mathcal{B}(\mathcal{P})$ and as predicates over concrete states, the above definition can be restated as requiring

$$\models m \wedge A \Rightarrow n'.$$

Since nodes of predicate diagrams are sets of $\overline{\mathcal{P}}$, interpreted conjunctively, we may assume m to be a conjunction of literals, and try to compute some successor n in disjunctive normal form.

As the first step of the abstract evaluation of A from m, we try to evaluate those subformulas of A that contain only unprimed variables to either true or false in order to simplify the action formula. This amounts to abstractly evaluating the guards of the action. In the second step, we try to evaluate as many formulas P', for $P \in \mathcal{P}$, as possible in order to assemble information about the predicates that are true or false after the action has been executed. Throughout, we maintain A and the results of simplification in disjunctive normal form.

Let us first describe the abstract evaluation procedure for monadic predicates. If $\mathcal{P}$ contains k monadic predicates $P_1(x), \ldots, P_k(x)$ that contain the same (concrete) state variable x, we let the abstract states contain k state variables $\overline{x_1}, \ldots, \overline{x_k}$ such that $\overline{x_i}$ takes values P_i or $not\text{-}P_i$. For example, the predicate $y_1 = 0$ might be represented by the variable $\overline{y_1} \in \{zero, not\text{-}zero\}$. Now, every unprimed occurrence of variable x in A replaced by the value assigned to $\overline{x_i}$ by the abstract source state m, for every $\overline{x_i}$ that appears in m. User-supplied rewrite rules of the form are used to simplify the resulting expressions. The evaluation is successful if the resulting formula is of the form

$$res \;\equiv\; \bigvee_p \bigwedge_q x' = e_{pq}$$

for some expressions e_{pq} that contain the "abstract values" P_i and $not\text{-}P_i$. The set of abstract successor states is extracted from res by picking those subformulas for which e_{pq} is either P_i or $not\text{-}P_i$.

Example 5. In the context of the Bakery specification considered in section 3.3, consider the evaluation of the next-state action $Next$ from the state

$$m = \{pc_1 = \text{"l1"}, pc_2 = \text{"m2"}, \overline{y_1} = zero, \overline{y_2} = zero\}$$

(Note that our tool allows concrete state variables to be left unabstracted; this is useful when variables take values from a finite domain.)

The underlying set of rewrite rules include

$$\begin{aligned}
\mathit{zero}{+}1 &\rightarrow \mathit{not\text{-}zero}\\
\mathit{not\text{-}zero}{+}1 &\rightarrow \mathit{not\text{-}zero}\\
\mathit{zero} = \mathit{zero} &\rightarrow \mathit{true}\\
\mathit{zero} = \mathit{not\text{-}zero} &\rightarrow \mathit{false}\\
\mathit{zero} \leq \mathit{zero} &\rightarrow \mathit{true}\\
\mathit{zero} \leq \mathit{not\text{-}zero} &\rightarrow \mathit{true}\\
\mathit{not\text{-}zero} \leq \mathit{zero} &\rightarrow \mathit{false}
\end{aligned}$$

Note that expressions such as $\mathit{not\text{-}zero} = \mathit{not\text{-}zero}$ or $\mathit{not\text{-}zero} \leq \mathit{not\text{-}zero}$ cannot be simplified.
Application of the abstract evaluation procedure outlined above produces the two successor states

$$\begin{aligned}
&\{pc_1 = \text{“l2”}, pc_2 = \text{“m2”}, \overline{y_1} = \mathit{zero}, \overline{y_2} = \mathit{zero}\}\\
&\{pc_1 = \text{“l1”}, pc_2 = \text{“m3”}, \overline{y_1} = \mathit{zero}, \overline{y_2} = \mathit{not\text{-}zero}\}
\end{aligned}$$

that correspond to either process taking a step in the protocol. □

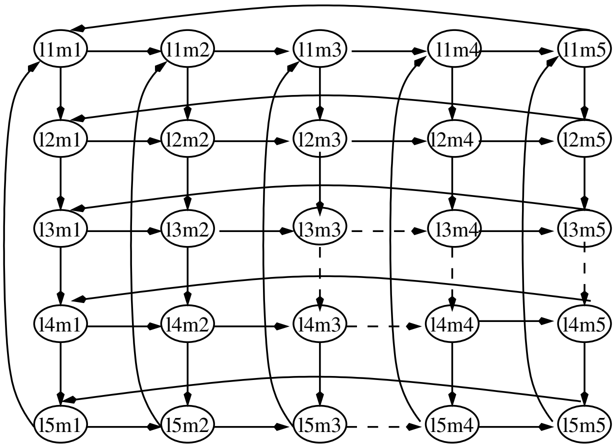
Given a specification $\mathit{Spec} \equiv \mathit{Init} \wedge \Box \mathit{Next} \wedge L$, we construct a predicate diagram G as follows: first Init is rewritten to disjunctive normal form, producing a set I of abstract initial states. Starting from I, we repeatedly apply abstract evaluation for Next and for any action such that WF(A) or SF(A) appears as a conjunct in L in order to arrive at the set N of abstract states and the relations δ_A for the chosen actions. Assuming that all rewrite rules are correct, the resulting predicate diagram is guaranteed to satisfy the conditions of theorem 3.
The procedure described above can be extended to sets $\mathcal{P}$ that contain other than just monadic predicates, but it becomes somewhat more tedious to describe and implement because it can no longer simply be based on syntactic substitution. We prefer to rely on the strengthening technique described in section 5.2.

5 Improving abstract evaluation

If the abstraction is not fine enough, it is not always possible to successfully evaluate an action A with respect to a source state m, because some of the guards cannot be reduced to true or false. In such a situation, we can try to compute an approximation, introducing *maybe* edges. Any property proved from the resulting over-approximation is still valid of the original specification Spec. On the other hand, we may be able to eliminate *maybe* edges introducing an “on-the-fly” refinement of the abstraction with the help of an automatic prover. Alternatively, we may introduce non-determinism as allowed in condition (2) of theorem 3 by relying on a lattice of abstractions.

5.1 Maybe edges

Suppose we choose $\{\mathit{zero}, \mathit{not\text{-}zero}\}$ as the abstract domain for the variables $\overline{y_1}$ and $\overline{y_2}$ of the Bakery algorithm. Then the abstract animation based on the evaluation rules



state	pc1	pc2	y1	y2	state	pc1	pc2	y1	y2
l1m1	"l1"	"m1"	*zero*	*zero*	l1m2	"l1"	"m2"	*zero*	*zero*
l1m3	"l1"	"m3"	*zero*	*not-zero*	l1m4	"l1"	"m4"	*zero*	*not-zero*
l1m5	"l1"	"m5"	*zero*	*not-zero*	l2m1	"l2"	"m1"	*zero*	*zero*
l2m2	"l2"	"m2"	*zero*	*zero*	l2m3	"l2"	"m3"	*zero*	*not-zero*
l2m4	"l2"	"m4"	*zero*	*not-zero*	l2m5	"l2"	"m5"	*zero*	*not-zero*
l3m1	"l3"	"m1"	*not-zero*	*zero*	l3m2	"l3"	"m2"	*not-zero*	*zero*
l3m3	"l3"	"m3"	*not-zero*	*not-zero*	l3m4	"l3"	"m4"	*not-zero*	*not-zero*
l3m5	"l3"	"m5"	*not-zero*	*not-zero*	l4m1	"l4"	"m1"	*not-zero*	*zero*
l4m2	"l4"	"m2"	*not-zero*	*zero*	l4m3	"l4"	"m3"	*not-zero*	*not-zero*
l4m4	"l4"	"m4"	*not-zero*	*not-zero*	l4m5	"l4"	"m5"	*not-zero*	*not-zero*
l5m1	"l5"	"m1"	*not-zero*	*zero*	l5m2	"l5"	"m2"	*not-zero*	*zero*
l5m3	"l5"	"m3"	*not-zero*	*zero*	l5m4	"l5"	"m4"	*not-zero*	*not-zero*
l5m5	"l5"	"m5"	*not-zero*	*not-zero*					

Fig. 5. Predicate diagram with *maybe* edges.

of section 4 will fail because the expressions $not\text{-}zero \leq not\text{-}zero$ that appear in the guards of actions $L3$ and $M3$ cannot be evaluated. Our first approximation is to assign the value $maybe$ to such uninterpreted expressions:

$$not\text{-}zero \leq not\text{-}zero \quad \rightarrow \quad maybe$$

"Maybe" values are propagated using rewrite rules such as

$$\begin{aligned} maybe \wedge maybe &\rightarrow maybe \\ false \wedge maybe &\rightarrow false \\ \neg maybe &\rightarrow maybe \end{aligned}$$

Successor states can be extracted as described above even in the presence of *maybe* conjuncts, but we remember such situations and indicate edges obtained in this way using dashed edges. For the Bakery specification, we obtain the graph shown in figure 5

(the values assigned to the abstract variables are shown in a separate table). The diagram gives a clear indication of the progress of the two processes with the first process moving in the vertical and the second in the horizontal direction.
The predicate diagram corresponds to an over-approximation of the Bakery specification and abstractly represents every behavior of the Bakery algorithm. It cannot be used to prove the invariant

$$Bakery.Spec \Rightarrow \Box\neg(pc_1 = \text{“l4”} \wedge pc_2 = \text{“m4”})$$

that asserts mutual exclusion for the two processes. However, every path leading to the abstract state l4m4 contains a dashed edge, and these edges indicate opportunities for refining the approximation by reconsidering the transitions in view of the concrete specification. We now describe a technique that can perform such a refinement during the computation of the diagram.

5.2 "On-the-fly" refinement of diagrams

Maybe edges are introduced when predicates that appear in the concrete-level next-state relation (such as $y1 \leq y2$ in the Bakery example) cannot be interpreted in the abstract states. We suggest to reconsider the offending transitions based on the predecessors of the abstract source state. More precisely, suppose that we cannot interpret a sub-formula Q in an abstract state m during the evaluation of action A. We may then try to evaluate Q' with respect to the formula $n \wedge A \wedge m'$, for every predecessor n of m. In other words, we try to prove either of

$$n \wedge A \wedge m' \;\Rightarrow\; Q' \qquad \text{or} \qquad n \wedge A \wedge m' \;\Rightarrow\; \neg Q'$$

By construction, it is never the case that both formulas are provable, and so we consider the three following cases:

	(1)	(2)	(3)
$n \wedge A \wedge m' \Rightarrow Q'$	Proved	Unproved	Unproved
$n \wedge A \wedge m' \Rightarrow \neg Q'$	Unproved	Proved	Unproved

In the first and second case, we may add Q resp. $\neg Q$ to the label of m (if m has more than one predecessor, the node is split into two copies). If all subformulas that originally evaluated to $maybe$ can be decided one way or another, the *maybe* edge can be replaced by a standard transition.
In the case of the Bakery example we can prove

$$\text{l3m2} \wedge Next \wedge \text{l3m3}' \Rightarrow \neg(y_1' \leq y_2')$$
$$\text{l2m3} \wedge Next \wedge \text{l3m3}' \Rightarrow y_1' \leq y_2'$$

Consequently, state l3m3 is split into two states l3m3$\wedge y_1 \leq y_2$ and l3m3$\wedge\neg(y_1 \leq y_2)$ as indicated in figure 6. Continuing the abstract evaluation, both *maybe* edges leading to node l4m4 disappear, and mutual exclusion can be proven.
In our implementation, we remember the subformulas of the concrete-level action that evaluated to $maybe$. If such expressions remain after rewriting completes, the tool calls

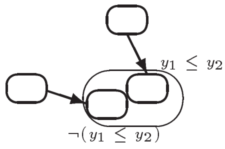


Fig. 6. Splitting state l3m3.

the automatic prover Simplify [8] in order to try and solve the resulting proof obligations, and splits the node as necessary. Applying this modified generation procedure to the Bakery specification, we arrive at the predicate diagram schematically shown in figure 7. In this diagram, the node at which both $pc_1 = $ "l3" and $pc_2 = $ "m3" are true has been split into two nodes that differ in the value assigned to the predicate $y_1 \leq y_2$, and the node l4m4 has been eliminated. In particular, mutual exclusion can now be established, although the diagram still contains two *maybe* edges.

Because of these edges, we can still not prove liveness properties about the Bakery algorithm. However, the edges can be eliminated by forward propagation of the inferred properties in a similar manner. In this way we obtain the predicate diagram shown in figure 8 for the Bakery example which precisely represents the behaviors of the specification modulo the chosen abstraction. Similar graphs have been produced manually, for example, in [14], but to our knowledge our tool is the first that is able to obtain this diagram semi-automatically, with the user only indicating that zero and non-zero values for the "ticket" variables should be distinguished.

Running Spin on the Promela encoding of the diagram of figure 8, we can prove liveness properties such as $Bakery.Spec \Rightarrow (pc_1 = \text{"l2"} \leadsto pc_1 = \text{"l4"})$ as well as precedence properties such as one-bounded overtaking for either process.

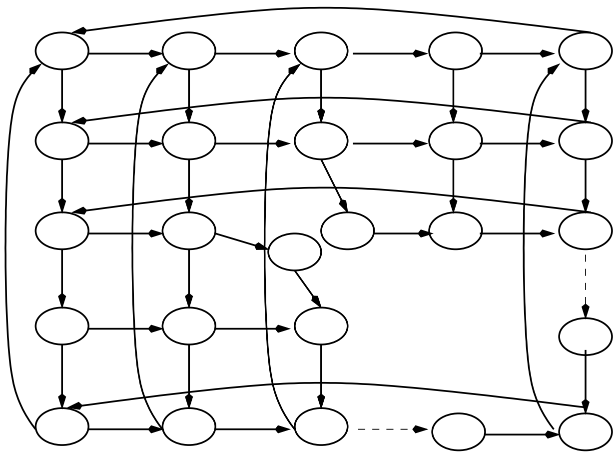


Fig. 7. Predicate diagram for $Bakery$ with some *maybe* edges removed.

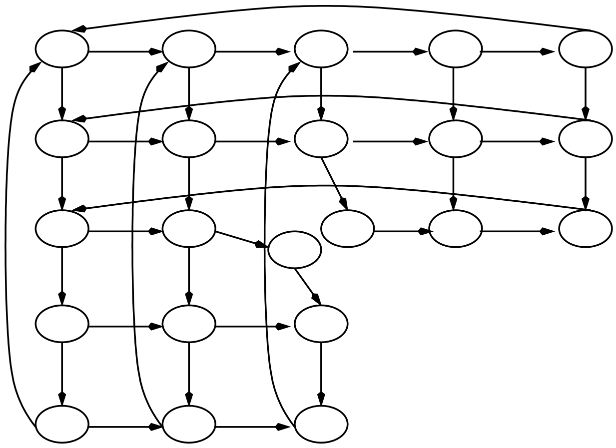


Fig. 8. Final predicate diagram for $Bakery$.

5.3 Using a lattice of abstractions

Sometimes when we cannot evaluate an expression in the abstract world we can change the abstract world using a lattice abstraction. For example, with the classical calculation rules [10] on $\{odd, even\}$ we cannot evaluate $even$ div 2 and we cannot prove that n is always greater than 0 for the "dining mathematicians" specification. Instead, we can use an abstraction using the three abstract values $zero$, $even^{>0}$ and odd; this amounts to abstracting on both predicates $\mathrm{even}(n)$ and $n = 0$. However, the expression $even^{>0}$ div 2 is yet uninterpreted. We can replace this abstract expression by another abstract value $not\text{-}zero$ which represents every natural greater than 0. After this evaluation we transform $x = not\text{-}zero$ in $x = even^{>0} \vee x = odd$. Formally, this transformation can be explained as a change of the degree of abstraction, cf. figure 9. When successful evaluation of the predicates at the more abstract level has established that $even^{>0}$ div 2 is a positive natural number, we can re-descend to the first abstraction based on $zero$, $even^{>0}$ and odd using the following calculation rules (where the variable x that appears in the last rule can be instantiated arbitrarily).

$$
\begin{array}{rclrcl}
even^{>0}+even^{>0} & \to & even^{>0} & even^{>0}+odd & \to & odd \\
odd+even^{>0} & \to & odd & odd+odd & \to & even^{>0} \\
even^{>0}\cdot even^{>0} & \to & even^{>0} & even^{>0}\cdot odd & \to & even^{>0} \\
odd\cdot even^{>0} & \to & even^{>0} & odd\cdot odd & \to & odd \\
even^{>0} \text{ div } 2 & \to & not\text{-}zero & \mathrm{even}(even^{>0}) & \to & TRUE \\
\mathrm{even}(odd) & \to & FALSE & x = not\text{-}zero & \to & x = even^{>0} \vee x = odd
\end{array}
$$

In this way, we can automatically construct the predicate diagram $DiningGraph$ of figure 4, except for the ordering annotation that must still be justified interactively. In principle, similar methods of abstract evaluation could be used to compute ordering annotations, but we believe that they should be used only judiciously as they greatly affect the effectiveness of model checking.

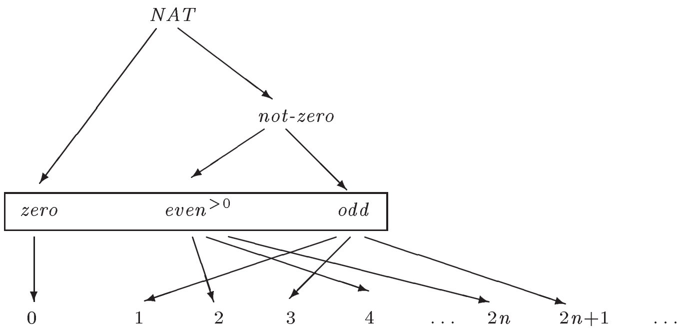


Fig. 9. A lattice of abstractions.

6 Related work

The first work using graphs to visualize and structure temporal proofs for concurrent programs is due to Lamport and Owicki [27]. There, proof lattices are used to better explain logic rules and to "see" and so verify what a program is supposed to do. Building on these ideas, proof charts were introduced in Radhia Cousot's thesis [5]. A proof chart for a transition system is a finite graph with a unique start and final state. Proof obligations are associated with every sub-chart, and "return" edges can be labelled by well-founded orderings to guarantee termination of the system. In contrast to our diagrams, these approaches concentrate on illustrating the structure of the proof rather than of the system under analysis.

Predicate-action diagrams proposed by Lamport [18] represent the safety part of specifications. An interesting point is that different, complementary views about a specification can be illustrated by different diagrams (even for the same set of predicates), and that the proof of refinement relations via diagrams is considered. Manna et al [7, 22, 23] have also advocated the diagrammatic verification of temporal logic properties. The main difference is in the representation of fairness conditions, which we assert on the level of entire diagrams rather than for individual edges. This simplifies the verification conditions related to fairness assumptions, and allows to take them into account during abstract evaluation. We also allow an arbitrary number of ordering annotations, which reduces the number of proof obligations, and should be more intuitive for a system designer. We have prototypically implemented refinement rules discussed in [22] that allow to split nodes or remove edges in our tool. Probably closest in spirit, although different in detail, to our work is the work by Bensalem, Saïdi et al on the generation of invariants that is incorporated in the InVest and SAL tools [3, 2, 28]. While our approach focuses more on abstract evaluation, they use the theorem prover PVS as their primary workhorse. Another conceptual difference is our emphasis on liveness properties; indeed, we believe that it is here that abstraction-based techniques can be used to full advantage because the deductive verification of liveness properties is inherently tedious and difficult.

7 Conclusion and Future Work

In this paper we have proposed predicate diagrams as a general framework for presenting Boolean abstractions of reactive systems and as a adequate theoretical basis for the integration of deductive and algorithmic verification techniques. We have shown that both safety and liveness properties can be verified from predicate diagrams thanks to the presence of fairness assumptions and of annotations related to well-founded orderings that can be associated with state transitions. While the focus of this paper has been a bottom-up approach to verification of given specifications, which could be characterized as *abstract model checking*, we have found predicate diagrams to be equally useful for proving refinement relations between systems in a top-down development method.
For the acceptance of formal methods in practice it is essential to achieve as much automation as possible, and to be able to supply relevant feedback to the developer. While obviously the process of computing abstractions for infinite-state systems can never be fully automatic, we have shown that a substantial degree of automation is possible. We also believe that predicate diagrams provide an adequate level of abstraction for system engineers, and that the abstract counter-examples that are produced when model checking a predicate diagram indicate how to refine the chosen abstraction. The generation of *maybe* edges should also help clarify missing predicates or evaluation rules for the computation of predicate diagrams.
Some small-scale case studies have convinced us that the method is powerful and really helps in verifying reactive systems. Our current prototype implementation is based on the rewriting engine *Logic Solver* from Atelier B [29], the automatic prover Simplify [8], and the model checker Spin [12]. Besides carrying out further case studies, we intend to concentrate on the representation of hierarchy and refinement in the formal development of reactive systems via predicate diagrams.

References

1. J.-R. Abrial. *The B book - Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *5th Langley Formal Methods Workshop*, 2000. to appear.
3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In *Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.
4. D. Cansell and D. Méry. Abstract animator for temporal specifications Application to TLA. In Gilberto File' and Agostino Cortesi, editors, *SAS'99 Static Analysis Symposium, Venise, Italie*, LNCS 1694, Heidelberg, September 1999. Springer Verlag.
5. R. Cousot. *Fondements des mé thodes de preuve d'invariance et de fatalité de programmes parallè les*. PhD thesis, INPL, novembre 1985. Doctorat-ès-Sciences.
6. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstraction preserving $\forall ctl^{\cdot}$, $\exists ctl^{\cdot}$ and $ctl^{\cdot}$. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 561–581. IFIP Transaction, 1994.
7. L. de Alfaro and Z. Manna. Temporal verification by diagram transformation. In R. Alur and T.A. Henzinger, editors, *CAV'96*, LNCS 1102, pages 287–299. Springer-Verlag, jul. 1996.

8. D. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC theorem prover. Technical report, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, November 1996.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
10. P. Granger. Static analysis of arithmetical congruences. *IJCM*, 30:165–190, 1989.
11. Hermann Hellwagner. Scalable readers/writers synchronization on shared-memory machines. Technical report, Siemens AG, ZFE ST SN 2, 1993.
12. G. Holzmann. The SPIN model checker. *IEEE Trans. on software engineering*, 16(5):1512–1542, May 1997.
13. Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. In *23rd Intl. Symp. Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 1998.
14. Y. Kesten and A. Pnueli. Verifying liveness by augmented abstraction. In A. Flum and M. Rodríguez-Artalejo, editors, *CSL'99*, LNCS 1683, pages 141–156. Springer-Verlag, sep. 1999.
15. F. Kröger. *Temporal Logics of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987. ISNB 3-540-17030-8.
16. L. Lamport. A new solution of dijkstra's concurent programming problem. *Communications of the ACM*, 17(8):435–455, May 1974.
17. L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3):872–923, May 1994.
18. L. Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, September 1995.
19. L. Lamport. The operators of TLA+. Technical report-WEB TLA, Digital Equipment Corporation, 1996. http://www.research.digital.com/SRC/tla/papers.html.
20. L. Lamport. The module structure of tla+. Technical report-WEB TLA, Digital Equipment Corporation, 1998. http://www.research.digital.com/SRC/tla/papers.html.
21. Z. Manna, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. Step : the stanford theorem prover. Technical report, Department of Computer Science, Stanford University, June 1994.
22. Z. Manna, A. Browne, H.B. Spima, and T.E. Uribe. Tools and algorithms for construction and analysis of systems. In Brinksma, editor, *TACAS '97*, LNCS 1217, pages 334–350. Springer-Verlag, apr. 1997.
23. Z. Manna, A. Browne, H.B. Spima, and T.E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *AMAST'98*, LNCS 1548, pages 28–41. Springer-Verlag, dec. 1998.
24. Z. Manna and A. Pnueli. *The temporal logics of reactive and concurrent systems - Specification*. Springer-Verlag, 1992.
25. S. Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, Digital Equipment Corporation 1997. Springer-Verlag.
26. S. Merz. On the verification of a self-stabilizing algorithm. Available at http://www.pst.informatik.uni-muenchen.de/˜merz/papers/dijkstra.ps.gz, 1998.
27. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *Transactions On Programming Languages and Systems*, 4(3):455–495, July 1982.
28. H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, 1999. Springer-Verlag.
29. STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.