Journal of Universal Computer Science, vol. 7, no. 2 (2001), 159-174
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/2/01

Diagram Refinements for the Design of Reactive Systems

Dominique Cansell
(Université de Metz & LORIA, France
cansell@loria.fr)

Dominique Méry
(Université Henri Poincaré & LORIA, France
mery@loria.fr)

Stephan Merz
(Institut für Informatik, Universität München, Germany
merz@informatik.uni-muenchen.de)

Abstract: We define a class of predicate diagrams that represent abstractions of—possibly infinite-state—reactive systems. Our diagrams support the verification of safety as well as liveness properties. Non-temporal proof obligations establish the correspondence between the original specification, whereas model checking can be used to verify behavioral properties. We define a notion of refinement between diagrams that is intended to justify the top-down development of systems within the framework of diagrams. The method is illustrated by a number of mutual-exclusion algorithms.

Key Words: refinement, abstraction, verification, temporal logic, TLA, diagrams.
Category: D.2.2, D.2.4, F.3.1

1 Introduction

Applications of increasingly complex reactive systems in sensitive areas such as process control or protocols for telecommunication or electronic commerce call for the use of formal methods and tools to establish the correctness of such systems. Model checking techniques have found wide acceptance, essentially because they are easy to integrate into conventional design methods and because they provide useful information in the form of counter-examples. However, they are usually applicable only to finite-state system models of relatively small size. This has been found acceptable in certain domains such as hardware design, but is a rather strong limitation when trying to develop software-intensive systems. On the other hand, interactive theorem provers can in principle be used to verify systems of arbitrary size and complexity, but they require special expertise and often lack sufficient automation for use in industrial-scale development projects.

Abstraction and composition are generally considered as the two key ingredients to bridge the gap between finite-state model checking and general-purpose theorem proving [KP98]. Among abstraction techniques, Boolean abstractions [LGS$^+$95] have become very popular because they are both powerful and easy to define in terms of predicates over the concrete state space. In previous work [CMM00] we have defined the concept of *predicate diagrams* to represent Boolean abstractions. Besides abstract states and state transitions, which are necessary to reason about safety properties of systems, predicate diagrams also include annotations related to fairness conditions and well-founded orderings, and can therefore be used to reason about liveness properties.

Temporal system properties are established by model checking the predicate diagram, viewed as a finite-state transition system. In particular, failed verification attempts produce counter-examples in terms of the abstract system, which can help to either debug the algorithm or improve the abstraction. We have also presented techniques based on abstract interpretation and automatic theorem proving that can be used to compute a predicate diagram for a given system specification and a set of predicates that define the abstraction.

In this paper, we propose the use of predicate diagrams in a top-down approach: starting from an initial diagram that expresses key correctness properties, the system is obtained as the result of successive, property-preserving refinements. Although the idea of development by stepwise refinement is a classical one, our contribution is the definition of proof obligations for refinement that are both *flexible* and *tractable*. By flexibility, we mean that non-trivial refinements must be supported. In particular, it must be possible to implement high-level fairness constraints by a combination of fairness assumptions on the lower-level actions and arguments based on well-founded orderings that underly typical implementations based on counters or queues. We say that a set of proof obligations is tractable if it can be effectively discharged, at least in principle, using standard verification tools. In our case, all verification conditions are either non-temporal (and therefore fall in the domain of interactive or automatic theorem provers) or are amenable to finite-state model checking.

We give two definitions of refinement between predicate diagrams: the simpler one assumes that the state space of the implementation extends that of the specification, whereas the more general one allows the state spaces to be connected by a gluing invariant, and thus provides support for hiding of abstract state components. We argue that our definitions are useful by deriving a series of mutual-exclusion protocols.

We formalize our concepts in Lamport's Temporal Logic of Actions TLA [Lam94]. Although this choice is not essential, it is convenient for two reasons. First, TLA formulas are invariant with respect to invisible state changes and therefore naturally support refinement. Second, TLA distinguishes two layers of formulas: action formulas concern states and state transitions, whereas temporal formulas are interpreted over runs, and this distinction is reflected in our definitions.

2 Predicate Diagrams

We briefly recall the main concepts of TLA, a variant of linear-time temporal logic that we use to write system specifications and proof obligations. TLA formulas are built from *state predicates* and *action formulas*; the latter may contain primed state variables. For example, $x > 3$ is a state predicate, and $x \leq y' + 1$ is an action. For an action formula A, we denote by $\text{ENABLED}\ A$ 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 formula obtained from P by replacing all flexible variables v that occur in P by v'. For an action formula A and a tuple v of state variables, $[A]_v$ denotes the formula $A \vee v' = v$, and $\langle A \rangle_v$ denotes the dual formula $A \wedge v' \neq v$.

Temporal formulas are built from state predicates and action formulas $[A]_v$ using boolean connectives, the *always* operator $\Box$, and quantification over rigid (state-independent) variables (written $\exists x : F$) or flexible (state-dependent) variables (written $\boldsymbol{\exists} x : F$). We write $\Diamond F$ for $\neg\Box\neg F$ and $\Diamond\langle A \rangle_v$ for $\neg\Box[\neg A]_v$. Other derived connectives include the *leadsto* operator, which is defined by $F \leadsto G \equiv \Box(F \Rightarrow \Diamond G)$ and the formulas

$$\begin{aligned} \mathrm{WF}_v(A) &\equiv \Diamond\Box \text{ ENABLED } \langle A\rangle_v \Rightarrow \Box\Diamond\langle A\rangle_v \\ \mathrm{SF}_v(A) &\equiv \Box\Diamond \text{ ENABLED } \langle A\rangle_v \Rightarrow \Box\Diamond\langle A\rangle_v \end{aligned}$$

that assert weak and strong fairness conditions for the action formula $\langle A\rangle_v$.

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. Action formulas 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 interpreted over *behaviors*, i.e. ω-sequences $\sigma = s_0 s_1 \ldots$ of states [Lam94]. In particular, the formula $\exists\, x : F$ is true of a behavior $\sigma = s_0 s_1 \ldots$ if there exists some behavior $\tau = t_0 t_1 \ldots$ such that s_i and t_i differ at most in the valuation of x. The precise semantics of flexible quantification is somewhat involved because it has to ensure invariance under stuttering, but the details are unimportant for this paper.

System specifications can be written as formulas of the form $Init \wedge \Box[Next]_v \wedge L$ where $Init$ is a state predicate that characterizes the system's initial state, $Next$ is an action formula representing the next-state relation, v is the tuple of state variables of interest, and L is a conjunction of formulas $\mathrm{WF}_v(A)$ or $\mathrm{SF}_v(A)$.

In the following 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 $\mathcal{O}$.

A predicate diagram [CMM00] is a finite graph whose nodes are labelled with sets of (possibly negated) predicates, and whose edges are labelled with action names as well as optional annotations that assert certain expressions to decrease with respect to an ordering in $\mathcal{O}^=$. Intuitively, a node of a predicate diagram represents the set of system states that satisfy the formulas contained in the node. (We indifferently write n for the set and the conjunction of its elements.) An edge (n,m) is labelled with action A if the action may cause a transition from a state represented by n to a state represented by m. An action A may have an associated fairness condition; it applies to all transitions labelled by A rather than to individual edges. We let edges be labelled with action names instead of action formulas because, in a top-down development, the precise action formula that defines an action is not known until the final specification has been derived.

Formally, the definition of predicate diagrams is relative to finite sets $\mathcal{P}$ and $\mathcal{A}$ that contain the state predicates and the (names of) actions; we will later use $\tau \notin \mathcal{A}$ to denote the stuttering action. We write $\overline{\mathcal{P}}$ to denote 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

- 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 $\delta_A \subseteq N \times N$ (by $\delta_=$ we denote the reflexive closure of the union of these relations),
- an edge labelling o that associates 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 $(n,m) \in \delta$, and

– 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 $m \in N$, and denote by $En(A) \subseteq N$ the set of nodes where A can be taken. □

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 $\mathcal{A}$. A *run of G* is an ω-sequence $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of triples where s_i is a state, $n_i \in N$ is a node, and $A_i \in \mathcal{A} \cup \{\tau\}$ is an action such that all of the following conditions hold:

– $n_0 \in I$ is an initial node,

– $s_i \models n_i$ holds for all $i \in \mathbb{N}$,

– for all $i \in \mathbb{N}$, either $A_i = \tau$ and $n_i = n_{i+1}$, or $A_i \in \mathcal{A}$ and $(n_i, n_{i+1}) \in \delta_{A_i}$,

– if $A_i \in \mathcal{A}$ and $(t, \prec) \in o(n_i, n_{i+1})$, then $(s_i, s_{i+1}) \models t' \prec t$,

– if $A_i = \tau$ then $(s_i, s_{i+1}) \models t' \preceq t$ holds for all $m \in N$ and $(t, \prec) \in o(n_i, m)$,

– 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.

The set $tr(G)$ of *traces through G* consists of all behaviors $\sigma = s_0 s_1 \ldots$ such that there exists a run $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of G based on the states in σ. □

In addition to the transitions that are explicitly represented by edges of the diagram, we allow stuttering transitions that remain in the source node. Note that we do not require that the states s_i and s_{i+1} be identical when $A_i = \tau$ is a stuttering step; they are only required to satisfy the same node label. Moreover, if node n_i has an outgoing edge with an ordering annotation $(t, \prec)$, stuttering transitions are forbidden to increase the value of t. Fairness conditions are used to prevent infinite stuttering.

For example, consider the predicate diagram shown on the right-hand side of Fig. 1 (the example is due to [DGG94]). By convention, the sets $\mathcal{P}$ and $\mathcal{A}$ contain the predicates and action names that appear in the diagram, and the edge label *Next* is omitted. Traces through the diagram are ω-sequences of states that satisfy the node labels and are related by edges shown in the diagram or by stuttering transitions. For example, the diagram rules out transitions from a state represented by the upper left node to one represented by the lower right node. In addition, the ordering annotations exclude behaviors that cycle forever between the two left-hand nodes because the value assigned to the variable n would have to decrease infinitely often, but can never increase. Assuming weak fairness for *Next*, it follows that all states of the diagram must be visited infinitely often during each run.

$$
\begin{array}{rl}
\mathit{Init} \equiv & n \in \mathit{Nat} \wedge n \neq 0 \\
& \wedge\, c_0 = \text{“t”} \wedge c_1 = \text{“t”} \\
\mathit{Eat}_0 \equiv & c_0 = \text{“t”} \wedge \mathrm{even}(n) \\
& \wedge\, c_0' = \text{“e”} \wedge c_1' = c_1 \wedge n' = n \\
\mathit{Thk}_0 \equiv & c_0 = \text{“e”} \wedge c_0' = \text{“t”} \\
& \wedge\, n' = n \operatorname{div} 2 \wedge c_1' = c_1 \\
\mathit{Eat}_1 \equiv & c_1 = \text{“t”} \wedge \neg\mathrm{even}(n) \\
& \wedge\, c_1' = \text{“e”} \wedge c_0' = c_0 \wedge n' = n \\
\mathit{Thk}_1 \equiv & c_1 = \text{“e”} \wedge c_1' = \text{“t”} \\
& \wedge\, n' = 3 * n + 1 \wedge c_0' = c_0 \\
\mathit{Next} \equiv & \mathit{Eat}_0 \vee \mathit{Thk}_0 \vee \mathit{Eat}_1 \vee \mathit{Thk}_1 \\
v \equiv & \langle c_0, c_1, n \rangle \\
\mathit{DM} \equiv & \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge \mathrm{WF}_v(\mathit{Next})
\end{array}
$$

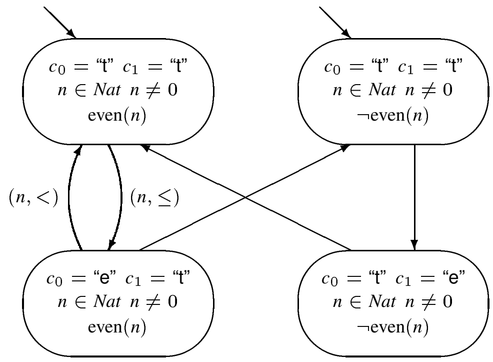

Figure 1: The “dining mathematicians” example.

3 System verification using predicate diagrams

We briefly describe the use of predicate diagrams for the verification of reactive systems. In linear-time formalisms such as TLA, trace inclusion is the appropriate implementation relation. Thus, a specification *Spec* implements a property or high-level specification F if and only if the implication $\mathit{Spec} \Rightarrow F$ is valid [Lam94]. Predicate diagrams can be used to refine this implication into two conditions: first, all behaviors allowed by *Spec* must also be traces through the diagram (in other words, the diagram is a correct abstraction of *Spec*), and second, every trace through the diagram must satisfy F. Although both conditions are stated in terms of trace inclusion, we use different proof techniques. To show the correctness of the abstraction, we consider the node and edge labels of diagrams as predicates on the concrete state space of *Spec*, and reduce trace inclusion to a set of proof obligations that concern individual states and transitions. On the other hand, to show that the diagram implies the high-level property, we regard all labels as Boolean variables. The predicate diagram can therefore be encoded as a finite labelled transition system, whose temporal properties are established by model checking. In this respect, predicate diagrams act as an interface between interactive and automatic proof methods. We now consider both aspects in more detail.

3.1 Relating specifications and predicate diagrams

In order to compare a TLA specification and a predicate diagram, we must first assign meaning to the action names that appear in the diagram. We assume given a function α that assigns an action formula to every action name. Because no confusion is possible, we will leave this assignment implicit, and again write A instead of $\alpha(A)$ when referring to the formula assigned to the name A.

We say that a predicate diagram G *conforms* to a specification *Spec* if every behavior that satisfies *Spec* is a trace through G. In general, proving conformance requires reasoning about entire behaviors. The following proposition, whose proof appears in [CMM00], allows us to reduce this temporal reasoning to a set of “local” proof obligations.

Proposition 3. *Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram over $\mathcal{P}$ and $\mathcal{A}$, and $Spec \equiv Init \wedge \Box[Next]_v \wedge L$ be a system specification. If all of the following conditions hold then G conforms to Spec.*

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

Proposition 3 can be used to show that the predicate diagram of Fig.1 conforms to the specification *DM* that appears on the left-hand side of Fig. 1. For example, we have

$$\begin{array}{l} \quad c_0 = \text{“e”} \wedge c_1 = \text{“t”} \wedge n \in Nat \wedge n \neq 0 \wedge \mathrm{even}(n) \\ \quad \wedge\ c_0' = \text{“t”} \wedge c_1' = \text{“t”} \wedge n' \in Nat \wedge n' \neq 0 \wedge \mathrm{even}(n') \\ \quad \wedge\ \langle Next \rangle_v \\ \Rightarrow n' < n \end{array}$$

because only the action Thk_0 is applicable in any state described by the precondition, and n div $2 < n$ holds for any positive integer n.

Because of condition 4(d), which ensures that the effects of actions with nontrivial fairness assumptions are fully represented in the diagram, the number of proof obligations can be quadratic in the number of nodes of the diagram. We have studied techniques based on abstract interpretation that can construct predicate diagrams that conform to a given specification *Spec* [CMM00].

3.2 Model checking predicate diagrams

Regarding predicate diagrams as finite labelled transition systems, their runs can be encoded in the input language of standard model checkers such as Spin [Hol97]. Two variables 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 $\neg P$. (Our actual implementation is slightly more general, because variables over arbitrary finite types rather than just the Booleans are allowed to appear in node labels.) 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 $o(n, m)$. These variables are set to 2 if the last transition taken is labelled by $(t, \prec)$, to 1 if it is labelled by $(t, \preceq)$ or is a stuttering transition, and to 0 otherwise.

The fairness conditions associated with the actions of a diagram are easily expressed as LTL assumptions for Spin. As in definition 2 we assume that action A is enabled whenever the currently active node has an outgoing edge in δ_A. To capture the effect of the ordering annotations, we add Streett-type formulas

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

as additional assumptions for every variable $b_{(t,\prec)}$ introduced. These assumptions ensure that transitions known to strictly decrease t with respect to $\prec$ can not be taken infinitely often unless infinitely often some transitions are taken that may increase the value of t.

Properties F whose atomic formulas are contained in the set $\mathcal{P}$ of predicates of interest can now be established by model checking the transition system that results from the encoding. If verification succeeds then every trace through the diagram satisfies F, and by transitivity of trace inclusion it follows that F holds of any specification that conforms to the diagram. On the other hand, counter-examples produced by the model checker need not correspond to actual system runs, because detail has been lost in the abstraction. Nevertheless, such counter-examples are helpful in order to refine the abstraction, for example by adding ordering annotations. Obviously, the size of diagrams that can be effectively analyzed in this way is mostly limited by the number of fairness conditions and ordering annotations present in the diagram.

Continuing the "dining mathematicians" example shown in Fig. 1, the following properties can all be verified from the predicate diagram:

$$\begin{array}{llll} (Pos) & \Box(n \in Nat \land n \neq 0) & (Excl) & \Box\neg(c_0 = \text{“e”} \land c_1 = \text{“e”}) \\ (Live_0) & \Box\Diamond(c_0 = \text{“e”}) & (Live_1) & \Box\Diamond(c_1 = \text{“e”}) \end{array}$$

The first two properties are invariants; they assert that n remains a positive natural number and that mutual exclusion is ensured. The remaining properties are liveness properties that assert starvation-freedom for both processes. As indicated at the end of section 2, the verification of property $(Live_1)$ relies on the ordering annotations that forbid the cycle between the left-hand nodes to be followed indefinitely.

4 Refinement of predicate diagrams

Not only can predicate diagrams be used to support bottom-up system verification, but they can also conveniently underpin top-down system development: starting from a coarse abstraction that represents overall system behavior, more and more detail is added until the system specification can be directly "read off" the final design. Technically, we have to define an appropriate refinement relation between diagrams. Whereas trace inclusion is all that is semantically required—in particular, addition of detail is possible thanks to stuttering invariance, methodologically we require a definition that is both flexible (i.e., supports non-trivial refinements) and tractable. Keeping with our general methodology, we require a definition in terms of "local" proof obligations. In other words, the overall structure of the abstract diagram will be preserved in the refining diagram.

4.1 Structural refinement

To simplify matters, we initially assume that the refining diagram G^1 interprets all predicates and actions that occur in the abstract diagram G^2. In particular, the set of state variables that appear in G^1 is a superset of those that appear in G^2. A more general definition that allows predicates of G^2 to be represented differently in G^1, and therefore allows for change of representation of the underlying data, will be considered in section 4.3.

Definition 4. Assume given two predicate diagrams $G^1 = (N^1, I^1, \delta^1, o^1, \zeta^1)$ over $\mathcal{P}^1$ and $\mathcal{A}^1$ and $G^2 = (N^2, I^2, \delta^2, o^2, \zeta^2)$ over $\mathcal{P}^2$ and $\mathcal{A}^2$ where $\mathcal{P}^1 \supseteq \mathcal{P}^2$ and $\mathcal{A}^1 \supseteq \mathcal{A}^2$, and let $f : N^1 \to N^2$. We say that G^1 *structurally refines* G^2 *w.r.t.* f iff all the following conditions hold:

1. $f(I^1) \subseteq I^2$
2. $\models n \Rightarrow f(n)$ holds for every node $n \in N^1$.
3. For all $n, m \in N^1$ and all $A \in \mathcal{A}^1$ such that $(n, m) \in \delta^1_A$:
   (a) if $A \in \mathcal{A}^2$ then $(f(n), f(m)) \in \delta^2_A$, and
   (b) if $A \in \mathcal{A}^1 \setminus \mathcal{A}^2$ then $(f(n), f(m)) \in \delta^2_{=}$.
4. For all $n, m \in N^1$, all $A \in \mathcal{A}^1$ such that $(n, m) \in \delta^1_A(n, m)$, all terms t and relations $\prec \in \mathcal{O}^{=}$:
   (a) $(t, \prec) \in o^1(n, m)$ whenever $(t, \prec) \in o^2(f(n), f(m))$,
   (b) $(t, \preceq) \in o^1(n, m)$ whenever $f(n) = f(m)$ and $(t, \prec) \in o^2(f(n), m')$ for some $m' \in N^2$.
5. For every run $\rho^1 = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of G^1 and every action $A \in \mathcal{A}^2$ such that $\zeta^2(A) = \mathrm{WF}$, either $A_i = A$ or $f(n_i) \notin En^2(A)$ holds for infinitely many $i \in \mathbb{N}$.
6. For every run $\rho^1 = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of G^1 and every action $A \in \mathcal{A}^2$ such that $\zeta^2(A) = \mathrm{SF}$, either $A_i = A$ for infinitely many $i \in \mathbb{N}$ or $f(n_i) \in En^2(A)$ for only finitely many $i \in \mathbb{N}$.

We say that G^1 *structurally refines* G^2 iff G^1 structurally refines G^2 w.r.t. some function $f : N^1 \to N^2$. $\square$

The definition of structural refinement w.r.t. f is such that conditions (1)–(4) are either purely structural or can be verified using non-temporal reasoning. On the other hand, conditions (5) and (6) are based on the runs of the diagrams and can therefore be established by model checking, using the encoding described in section 3.2. Note in particular that we do not require high-level fairness conditions to be syntactically preserved in the refining diagram; as we will see in the examples below, this makes our definition flexible. On the other hand, we require that transitions in G^1 preserve any ordering annotations asserted for the corresponding transitions of G^2, and that transitions of G^1 that correspond to stuttering steps in G^2 do not increase terms for which ordering annotations are present in G^1. This requirement of strict preservation could be relaxed by allowing a transition of the abstract diagram with an associated ordering constraint $(t, \prec)$ to be simulated by a sequence of transitions in the refining diagram such that each transition ensures $t' \preceq t$ and at least one transition ensures $t' \prec t$.

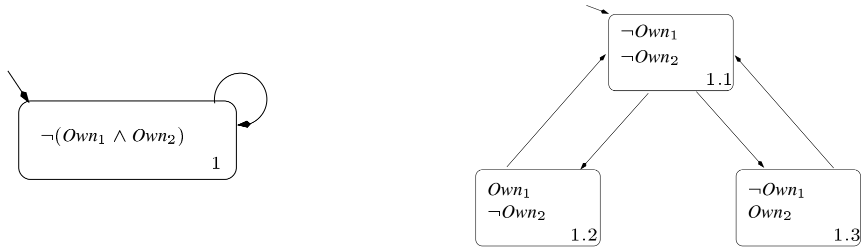

Figure 2: Mutual exclusion: initial diagram and first refinement.

The following theorem asserts that structural refinement ensures trace inclusion, and is therefore sound. In particular, all properties shown of G^2 remain valid for G^1.

Theorem 5. *If G^1 structurally refines G^2 then $tr(G^1) \subseteq tr(G^2)$.*

Proof. Assume that G^1, G^2 and f are as in definition 4, and that $\sigma = s_0 s_1 \dots$ is a trace of G^1. We must prove that $\sigma \in tr(G^2)$.

By the definition of a trace, choose some run $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\dots$ of G^1 based on σ. We will define a sequence $\pi = (s_0, f(n_0), B_0)(s_1, f(n_1), B_1)\dots$ and prove that π is a run of G^2. Define the sets $\mathcal{B}_i$ by

$$\mathcal{B}_i = \{A \in \mathcal{A}^2 : (f(n_i), f(n_{i+1})) \in \delta_A^2\}$$

and let $\mathcal{B}_i^* = \mathcal{B}_i$ if $f(n_i) \neq f(n_{i+1})$, and $\mathcal{B}_i^* = \mathcal{B}_i \cup \{\tau\}$ otherwise. Condition (3) of definition 4 ensures that $\mathcal{B}_i^* \neq \emptyset$ holds for all $i \in \mathbb{N}$. Now choose a sequence $B_0, B_1, \dots$ of actions $B_i \in \mathcal{B}_i^*$ such that every action $A \in \mathcal{A}^2$ that appears in infinitely many sets $\mathcal{B}_i^*$ is chosen infinitely often. (Such a choice is possible because the set $\mathcal{A}^2$ is finite.)

Since ρ is a run of G^1, we know that $n_0 \in I^1$, hence condition (1) ensures that $f(n_0) \in I^2$. Moreover, $s_i \models n_i$ for all $i \in \mathbb{N}$, and condition (2) implies $s_i \models f(n_i)$. Similarly, the choice of B_i and condition (3) ensures that either $B_i = \tau$ and $f(n_i) = f(n_{i+1})$ or $B_i \in \mathcal{A}^2$ and $(f(n_i), f(n_{i+1})) \in \delta_{B_i}^2$. Finally, using condition (4) and the fact that ρ is a run of G^1, it follows that $(s_i, s_{i+1}) \models t' \prec t$ holds whenever $(t, \prec) \in o^2(f(n_i), f(n_{i+1}))$ if $B_i \in \mathcal{A}^2$, and that $(s_i, s_{i+1}) \models t' \preceq t$ if $B_i = \tau$ and node $f(n_i)$ has an outgoing edge labelled by $(t, \prec)$.

It remains to show that π satisfies the fairness conditions associated with G^2. Thus, let $A \in \mathcal{A}^2$ be some action such that $\zeta^2(A) = \text{WF}$, and assume that $f(n_i) \in En^2(A)$ holds for all but finitely many $i \in \mathbb{N}$ (otherwise, the fairness condition is trivially satisfied). Condition (5) implies that $A_i = A$ holds for infinitely many $i \in \mathbb{N}$, and condition (3a) ensures that $A \in \mathcal{B}_i^*$ holds for infinitely many $i \in \mathbb{N}$. The choice of actions B_i implies $B_i = A$ for infinitely many $i \in \mathbb{N}$, which completes the proof. The argument for actions $A \in \mathcal{A}^2$ such that $\zeta^2(A) = \text{SF}$ is analogous, replacing "all but finitely many" by "infinitely many" and using condition (6) instead of (5). $\square$

4.2 Example: mutual exclusion protocols

We will now argue that the notion of structural refinement is flexible by performing a sequence of refinements that lead to different two-process mutual exclusion protocols,

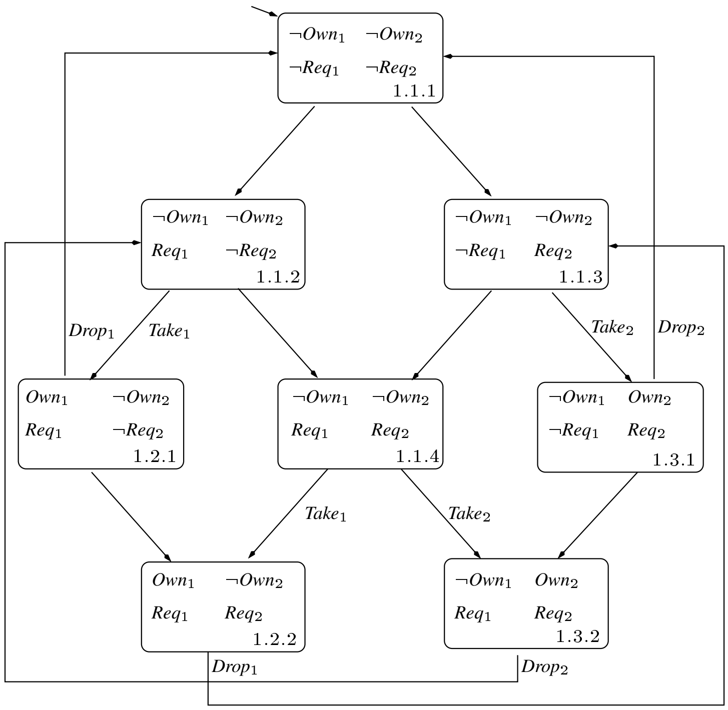

Figure 3: Mutual exclusion: adding requests.

a standard benchmark problem in the design of reactive systems. The basic requirement is to ensure that the two processes can never simultaneously access a shared resource. The simplest diagram that expresses this requirement is shown on the left-hand side of Fig. 2. It does not indicate how mutual exclusion is achieved, nor does it satisfy any liveness properties.

A first step towards implementation is represented by the diagram shown on the right-hand side of Fig. 2, which rules out direct hand-overs of the resource from one process to the other. (We no longer show the loops on the nodes because they are implicit in the definition of a run.) Here and in the following diagrams, we use a numbering convention to indicate the refinement function: a node $x.y$ of the refining diagram is mapped to node x of the abstract one. It is easy to see that the second diagram is a structural refinement of the first one because every node label implies $\neg(Own_1 \wedge Own_2)$ and every transition is allowed by the loop on the only node of the initial diagram.

Continuing our development, we want to add fairness requirements for processes that have requested the resource. We therefore add predicates Req_1 and Req_2 that indicate which processes have issued a request and split the nodes accordingly (see Fig. 3).

We have also introduced actions $Take_1$, $Take_2$, $Drop_1$, and $Drop_2$ that model acquisition and release of the resource by the processes. The conditions of definition 4 are again easily checked; in particular, transitions such as the one from node 1.1.1 to 1.1.2 where process 1 requests the resource are covered by stuttering transitions in the

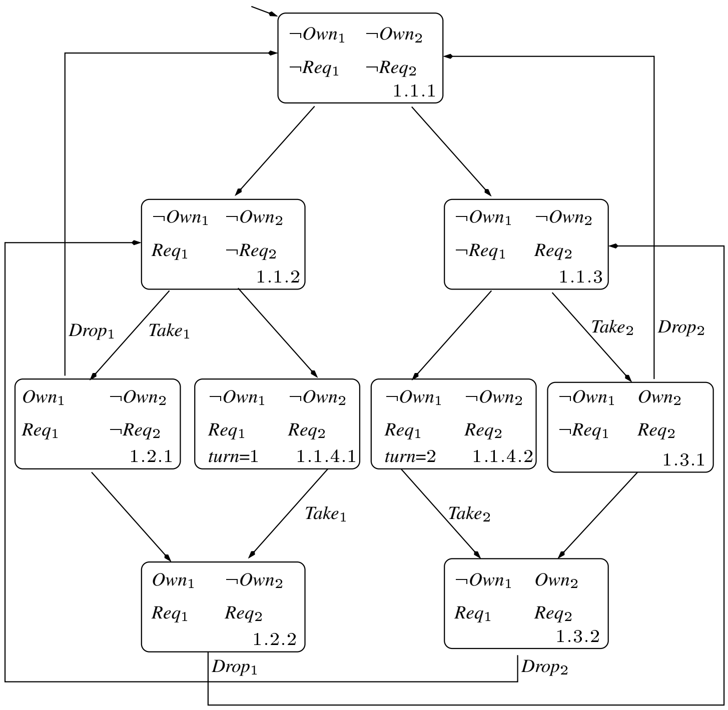

Figure 4: Mutual exclusion: priority-based implementation.

previous diagram.

We require strong fairness of the $Take_i$ actions, and weak fairness of the $Drop_i$ actions to ensure that conflicts are resolved in a fair way, and that no process monopolizes the resource. (Note that strengthening fairness conditions is a particular case of structural refinement.) Model checking establishes that the diagram satisfies the liveness properties

$$Req_1 \leadsto Own_1 \qquad \text{and} \qquad Req_2 \leadsto Own_2$$

We have now reached a diagram that represents a semaphore implementation of mutual exclusion: the semaphore is taken in those states where some process owns the resource, and free otherwise. Interestingly, we can continue our development to obtain an implementation along the lines of Peterson's algorithm if we assign priorities to the processes and ensure that priorities change whenever both processes compete for access to the resource. We only need to split node 1.1.4 into two subnodes as shown in Fig. 4; the nodes 1.1.4.1 and 1.1.4.2 assign priority to different processes. Because there is no longer a conflict between the actions $Take_1$ and $Take_2$, the strong fairness requirement for these actions is replaced by weak fairness.

It is easy to establish conditions (1)–(4) in order to prove that the diagram of Fig. 3 is structurally refined by that of Fig.4 because the latter diagram allows fewer transitions than the former. Proving refinement of the original fairness conditions is more subtle because the nodes 1.1.4.1 and 1.1.4.2 are both mapped to the abstract node 1.1.4 where both $Take_1$ and $Take_2$ are enabled. However, the alternation of priority ensures that conflicts are fairly resolved, and in fact the model checker confirms that condition (6) of definition 4 is satisfied. The predicate diagram shown in Fig. 4 therefore inherits all the desired safety and liveness properties that had been obtained for the previous diagrams.

4.3 Data refinement

Structural refinement ensures that every trace through the refining diagram is also a trace through the original diagram. We now present a generalization of definition 4 and allow the refining diagram G^1 to be based on a different set of predicates than the original diagram G^2. Consequently, there may be variables x^2 that occur in G^2 but not in G^1; in other words, the representation of data may have changed. We require the state spaces of the two diagrams to be connected by a "gluing invariant" that, intuitively, allows the abstract variables and predicates to be computed from the concrete ones. The definition ensures that for every trace through the refining diagram there exists a trace through the original diagram such that corresponding states in the traces satisfy the gluing invariant. It follows that properties established for the original diagram are preserved up to the gluing invariant. Similar notions of data refinement are used in the refinement calculus [BvW98] and its descendants such as Z [Spi92] and B [Abr96a].

Definition 6. Assume given two predicate diagrams $G^1 = (N^1, I^1, \delta^1, o^1, \zeta^1)$ over $\mathcal{P}^1$ and $\mathcal{A}^1$ and $G^2 = (N^2, I^2, \delta^2, o^2, \zeta^2)$ over $\mathcal{P}^2$ and $\mathcal{A}^2$ where $\mathcal{A}^1 \supseteq \mathcal{A}^2$, as well as a state predicate L. Let x^2 be a tuple of all variables that occur in $\mathcal{P}^2$, but not in $\mathcal{P}^1$, and let $f : N^1 \to N^2$. We say that G^1 *structurally refines* G^2 *up to* L *w.r.t.* f iff all the following conditions hold:

1. $f(I^1) \subseteq I^2$,
2. $\models n \Rightarrow \exists x^2 : L \wedge f(n)$ holds for every node $n \in N^1$.
3. For all $n, m \in N^1$ and all $A \in \mathcal{A}^1$ such that $(n, m) \in \delta^1_A$:
   (a) if $A \in \mathcal{A}^2$ then $(f(n), f(m)) \in \delta^2_A$, and
   (b) if $A \in \mathcal{A}^1 \setminus \mathcal{A}^2$ then $(f(n), f(m)) \in \delta^2_=$.
4. For all $n, m \in N^1$ and all $A \in \mathcal{A}^1$ such that $(n, m) \in \delta^1_A$:
   (a) for all $(t_2, \prec_2) \in o^2(f(n), f(m))$ there exists some $(t_1, \prec_1) \in o^1(n, m)$ such that
      – $\models n \wedge m' \wedge L \wedge L' \wedge t_1' \prec_1 t_1 \Rightarrow t_2' \prec_2 t_2$ and
      – $\models n \wedge n' \wedge L \wedge L' \wedge t_1' \preceq_1 t_1 \Rightarrow t_2' \preceq_2 t_2$,
   (b) if $f(n) = f(m)$ and $(t_2, \prec_2) \in o^2(f(n), m')$ holds for some $m' \in N^2$ then there exists some $(t_1, \prec_1) \in o^1(n, m)$ such that
   $$\models n \wedge m' \wedge L \wedge L' \wedge t_1' \prec_1 t_1 \Rightarrow t_2' \preceq_2 t_2$$
5. For every run $\rho^1 = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of G^1 and every action $A \in \mathcal{A}^2$ such that $\zeta^2(A) = \mathrm{WF}$, either $A_i = A$ or $f(n_i) \notin En^2(A)$ holds for infinitely many $i \in \mathbb{N}$.

6. For every run $\rho^1 = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of G^1 and every action $A \in \mathcal{A}^2$ such that $\zeta^2(A) = \text{SF}$, either $A_i = A$ for infinitely many $i \in \mathbb{N}$ or $f(n_i) \in En^2(A)$ for only finitely many $i \in \mathbb{N}$.

We say that G^1 *structurally refines* G^2 *up to* L iff G^1 structurally refines G^2 up to L w.r.t. some function $f : N^1 \to N^2$. □

We still require the actions of the original diagram to appear in the refining diagram. This requirement could be dropped, in particular for actions without fairness conditions, but this would complicate the statement of definition 6, and does not seem to be very useful—recall that only action names appear in predicate diagrams, and that an implementation may assign the same action formula to different action names. Other generalizations would allow the abstract variables x^2 to be related to concrete-level variables by history and prophecy simulations [AL91] instead of a simple gluing invariant L. Our current experience is too limited to determine a useful and tractable formulation of such a more flexible definition.

The correctness property associated with structural refinement up to a gluing invariant is stated in the following theorem.

Theorem 7. *Assume that G^1 structurally refines G^2 up to L, and let x^2 again denote a tuple of all variables that occur in G^2, but not in G^1.*

1. *For every trace $\sigma = s_0 s_1 \ldots$ through G^1 there exists a trace $\tau = t_0 t_1 \ldots$ through G^2 such that t_i differs from s_i at most in the valuation of the variables x^2, and such that $t_i \models L$ holds for all $i \in \mathbb{N}$.*
2. *If G^2 satisfies formula F then G^1 satisfies $\exists\!\!\exists\, x^2 : \Box L \wedge F$.*

Proof. 1. The proof follows the argument used in the proof of theorem 5, except that for the definition of the corresponding run of G^2 some x^2-variant t_i of every state s_i needs to be chosen such that $t_i \models L$. The existence of such states is ensured by condition (2). The actions B_i are chosen as before, and the proof obligations for the ordering annotations in G^1 ensure that the run of G^2 meets all required conditions.

2. An immediate consequence of assertion 1. □

To illustrate the application of definition 6, we continue the development of mutual-exclusion protocols to derive an atomic version of Lamport's two-process Bakery algorithm. Figure 5 shows a diagram with the same structure as the diagram of Fig. 4, but with a different labelling of the nodes and some additional action names. (Think of p_1 and p_2 as program counters whose values correspond to "non-critical", "requesting", and "critical".) As before, we require weak fairness of the $Take_i$ and $Drop_i$ actions.

It is easy to prove that the diagram of Fig. 5 structurally refines the diagram of Fig. 4 up to the gluing invariant L defined as

$$\begin{array}{l} (Own_1 \equiv p_1 = \text{“cr”}) \wedge (Own_2 \equiv p_2 = \text{“cr”}) \\ \wedge\ (Req_1 \equiv t_1 \neq 0) \wedge (Req_2 \equiv t_2 \neq 0) \\ \wedge\ (turn = \textbf{ if } t_1 \leq t_2 \textbf{ then } 1 \textbf{ else } 2) \end{array}$$

In particular, theorem 7.2 and simple temporal reasoning implies that the properties

$$t_1 \neq 0 \rightsquigarrow p_1 = \text{“cr”} \qquad \text{and} \qquad t_2 \neq 0 \rightsquigarrow p_2 = \text{“cr”}$$

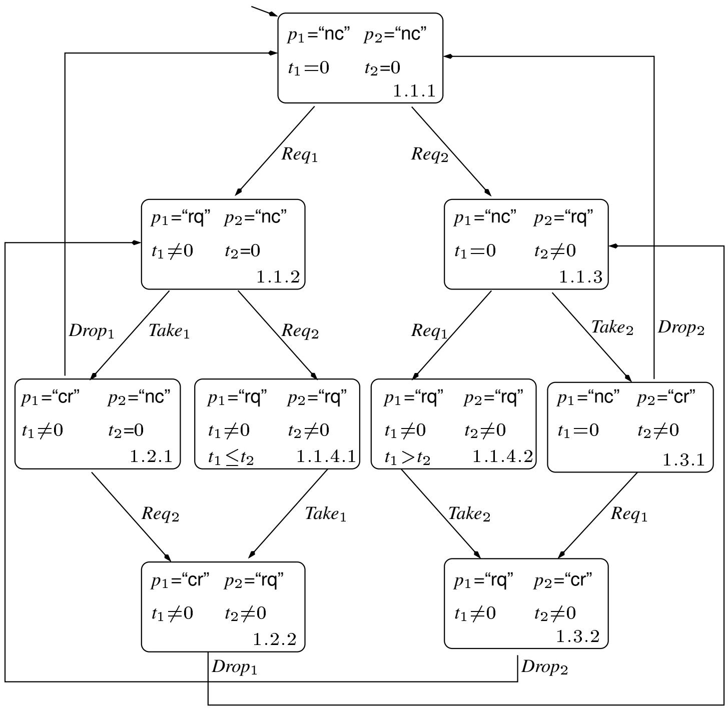

Figure 5: Mutual exclusion: Bakery algorithm.

hold of the diagram of Fig. 5. Thus, whenever a process has "drawn a ticket", it is guaranteed to eventually enter its critical section.

In order to arrive at an actual implementation, we must find an assignment of action formulas to action names that satisfies the conditions of proposition 3. One possible solution is to let the *Req* actions assign the ticket value of the other process, incremented by one, to the ticket of the own process, and to let the *Drop* action reset the ticket value to 0. We thus arrive at a conventional presentation of the atomic version of Lamport's Bakery algorithm, as shown in Fig. 6.

5 Related Work

There have been many suggestions for visual notations that support the formal development and analysis of systems. Owicki and Lamport [OL82] defined proof lattices to structure proofs of concurrent programs. A proof lattice is an acyclic finite graph with input assertion P and output assertion Q; it represents the proof of a liveness property $P \leadsto Q$. A structuring mechanism was sketched to introduce or remove invariants. Anderson's implementation of the UNITY logic in the interactive theorem prover HOL [And92] makes use of proof lattices. Radhia Cousot [Cou85] introduced proof charts to reason about parallel programs. Termination proofs are based on annotations

$$\begin{aligned}
Init &\equiv p_1 = \text{“nc”} \wedge p_2 = \text{“nc”} \wedge t_1 = 0 \wedge t_2 = 0\\
Req_1 &\equiv p_1 = \text{“nc”} \wedge p_1' = \text{“rq”} \wedge t_1' = t_2 + 1 \wedge \text{UNCHANGED } \langle p_2, t_2\rangle\\
Req_2 &\equiv p_2 = \text{“nc”} \wedge p_2' = \text{“rq”} \wedge t_2' = t_1 + 1 \wedge \text{UNCHANGED } \langle p_1, t_1\rangle\\
Take_1 &\equiv p_1 = \text{“rq”} \wedge t_1 \leq t_2 \wedge p_1' = \text{“cr”} \wedge \text{UNCHANGED } \langle t_1, p_2, t_2\rangle\\
Take_2 &\equiv p_2 = \text{“rq”} \wedge t_1 > t_2 \wedge p_2' = \text{“cr”} \wedge \text{UNCHANGED } \langle t_2, p_1, t_1\rangle\\
Drop_1 &\equiv p_1 = \text{“cr”} \wedge p_1' = \text{“nc”} \wedge t_1' = 0 \wedge \text{UNCHANGED } \langle p_2, t_2\rangle\\
Drop_2 &\equiv p_2 = \text{“cr”} \wedge p_2' = \text{“nc”} \wedge t_2' = 0 \wedge \text{UNCHANGED } \langle p_1, t_1\rangle\\
Next &\equiv Req_1 \vee Req_2 \vee Take_1 \vee Take_2 \vee Drop_1 \vee Drop_2\\
v &\equiv \langle p_1, p_2, t_1, t_2\rangle\\
Bakery &\equiv Init \wedge \Box[Next]_v \wedge \mathrm{WF}_v(Take_1) \wedge \mathrm{WF}_v(Take_2) \wedge \mathrm{WF}_v(Drop_1) \wedge \mathrm{WF}_v(Drop_2)
\end{aligned}$$

Figure 6: TLA specification of the two-process atomic Bakery algorithm.

that identify well-founded orderings. Lamport [Lam95] defines predicate-action diagrams that represent TLA specifications, restricted to safety properties. His diagrams are intended to communicate aspects of specifications; diagrammatic verification is not considered.

Work by Manna et al [dAMSU97, MBSU98] in the context of the STeP verification system is most closely related to our use of diagrams. Our definitions differ in technical details, but share the general motivations. For example, our ordering annotations are local to edges of the diagram, which simplifies the presentation and reduces the number of proof obligations. Refinement transformations on diagrams such as node splitting and transition removal are considered in [dAMSU97]. In contrast, our definitions of refinement are not restricted to isolated modifications applied to single nodes or edges, but support non-trivial refinements of fairness and liveness properties.

Our style of top-down development owes heavily to the B approach to system development [Abr96a, Abr96b, Abr00]. However, working in the more general setting of temporal logic, we prefer to state abstract fairness requirements early in the development; they can be implemented by well-founded orderings later on.

6 Conclusions

Predicate diagrams provide a framework for reasoning about reactive systems, based on Boolean abstractions. We have proposed to use predicate diagrams as the basis of a top-down refinement method. Throughout, care has been taken to separate the reasoning about states and state transitions from the reasoning about behaviors, which is required for the proof of liveness properties. This separation makes the proof obligations tractable. In particular, all proof obligations concerning temporal properties are stated in terms of finite transition systems and can therefore be discharged by model checking. Nevertheless, we have argued that our definitions are flexible enough to support non-trivial refinements of fairness conditions by a mix of low-level fairness assumptions and arguments based on well-founded orderings.

We are developing a prototype tool that will enable us to carry out more substantial case studies. We expect that these will point us to useful generalizations of the refinement relations defined in this paper, especially concerning data refinement. We also intend to study the representation of parameterized systems by predicate diagrams.

Acknowledgements

We are grateful to the anonymous referees for many helpful comments and suggestions. This work has been partly supported by a PROCOPE grant from EGIDE and DAAD.

References

[Abr96a] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.

[Abr00] J.-R. Abrial. Event-driven sequential programs. http://www-sop.inria.fr/EJC2000/edspc.V12.ps.gz, March 2000.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.

[And92] F. Anderson. *A theorem prover for* UNITY *in Higher Order Logic*. PhD thesis, Technical University of Denmark, 1992.

[BvW98] R. Back and J. von Wright. *Refinement calculus—A systematic introduction*. Springer-Verlag, 1998.

[CMM00] Dominique Cansell, Dominique Méry, and Stephan Merz. Predicate diagrams for the verification of reactive systems. In *2nd Intl. Conf. on Integrated Formal Methods (IFM 2000)*, volume 1945 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, November 2000. Springer-Verlag.

[Cou85] Radhia Cousot. *Fondements de méthodes de preuve d'invariance et de fatalité de programmes parallèles*. PhD thesis, INPL, 1985.

[dAMSU97] Luca de Alfaro, Zohar Manna, Henny B. Sipma, and Tomás Uribe. Visual verification of reactive systems. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 334–350. Springer-Verlag, 1997.

[DGG94] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods, and Calculi (PROCOMET '94)*, pages 561–581, Amsterdam, 1994. North Holland/Elsevier.

[Hol97] Gerard Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, may 1997.

[KP98] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. In *23rd Intl. Symp. Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 1998.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam95] Leslie Lamport. TLA in Pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, September 1995.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[MBSU98] Z. Manna, A. Browne, H.B. Sipma, and T.E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1998.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Spi92] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.