Formal Analysis of a Self-Stabilizing Algorithm Using Predicate Diagrams ⋆

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

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

Abstract

We present the verification of a protocol designed to ensure self-stabilization in a ring of processors. The proof is organized as a series of refinements; it is mechanized based on a combination of theorem proving and model checking to guarantee the correctness of these refinements. We argue that the framework of predicate diagrams is flexible enough to carry out a non-trivial verification task, that it provides a natural interface between automatic and interactive verification technology, and that it allows to present the correctness argument in an accessible manner.

Keywords: *formal methods, theorem proving, model checking, abstraction, refinement*

1 Introduction

In a seminal paper [3], Dijkstra introduced the concept of self-stabilizing systems and presented three algorithms that achieve self-stabilization in a ring of N processors. We present a verification of one of Dijkstra's algorithms in terms of Boolean abstractions. Specifically, we use the format of predicate diagrams that we have introduced in previous work [1, 2] to justify the correctness of the protocol both to the human reader and to verification tools. The proof is given as a series of refinements

$$\mathit{Prop} \sqsubseteq \mathcal{D}_{abs} \sqsubseteq \mathcal{D}_{conc} \sqsubseteq \mathit{Spec}$$

where *Prop* is a temporal logic formula that expresses stabilization, $\mathcal{D}_{abs}$ and $\mathcal{D}_{conc}$ are predicate diagrams that represent, respectively, an "abstract" and a more detailed view of the protocol, and *Spec* is a TLA$^+$ specification of Dijkstra's algorithm. The correctness of the property over the abstract diagram is obtained using model checking. Alternatively, $\mathcal{D}_{abs}$ could be constructed from *Prop* using standard automata-theoretic techniques. Model checking is also used to justify that $\mathcal{D}_{conc}$ is indeed a refinement of $\mathcal{D}_{abs}$. (In general, a combination of model checking and interactive, non-temporal theorem proving may be necessary to establish the correctness of diagram refinements). Finally, non-temporal theorem proving establishes the correctness of $\mathcal{D}_{conc}$ with respect to the protocol specification *Spec*. In contrast to a previous "flat" verification [9] of the protocol in PVS, our proof has a clearer structure and also requires less user interaction because part of the verification effort is delegated to model checking. The diagrammatic presentation of the abstraction helps to clarify the structure of the proof and provides a natural interface between the model checker and the interactive theorem prover. The definition of diagram refinement given in [2], which in particular allows the implementation of fairness conditions at the higher level by a combination of lower-level fairness conditions and arguments based on well-founded orderings, turned out to be flexible enough for the purposes of this case study.

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

2 Formal Background

2.1 Temporal Logic of Actions

Our formalism is framed in Lamport's TLA [6], and we therefore briefly recall its main concepts. 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. State formulas are interpreted over *states*, which assign values to state variables. Action formulas are interpreted over pairs (s,t) of states, where s and t interpret respectively the unprimed and primed state variables. For an action formula A, we denote by ENABLED A the state predicate obtained from A by existential quantification over the primed state variables; that predicate holds of precisely

those states s for which there exists some state t such that (s,t) satisfies A. 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 $\exists x : F$). 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 *leadsto* formulas $F \rightsquigarrow G \equiv \Box(F \Rightarrow \Diamond G)$ and the weak and strong fairness conditions

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

Temporal formulas are interpreted over *behaviors*, i.e. ω-sequences $\sigma = s_0 s_1 \ldots$ of states [6].
System specifications are usually 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 a tuple of state variables, and L is a conjunction of formulas $\mathrm{WF}_v(A)$ or $\mathrm{SF}_v(A)$.
We assume that the underlying assertion language contains a set O of binary relation symbols $\prec$ that are interpreted by well-founded orderings. For $\prec \in O$, we denote by $\preceq$ its reflexive closure. We write $O^{=}$ to denote the set of relation symbols $\prec$ and $\preceq$, for $\prec$ in O.

2.2 Predicate Diagrams

A predicate diagram [1, 2] is a finite graph whose nodes are labelled with sets of (possibly negated) state predicates, and whose edges are labelled with action names and may carry annotations that assert certain expressions to decrease with respect to an ordering in $O^{=}$. 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 A 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, which applies to all transitions labelled by A. We let edges be labelled with action names instead of action formulas because, in a top-down development, the 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 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 function* o *that associates pairs* $(t_1, \prec_1), \ldots, (t_k, \prec_k)$ *of terms* t_i *and relation symbols* $\prec_i \in O^{=}$ *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.*

A behavior $\sigma = s_0 s_1 \ldots$ is a *trace* through diagram G if there exists a corresponding run $n_0 \xrightarrow{A_0} n_1 \xrightarrow{A_1} \ldots$ such that all node and edge labels are satisfied. 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. Besides the transitions that are explicitly represented by edges of the diagram, we allow for stuttering transitions that loop at the source node. The precise definition has been given in [2].
We say that a predicate diagram G *conforms* to a TLA 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 [1] states a set of "local" proof obligations sufficient for proving conformance.

Proposition 2 *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. *for every node* $n \in N$,
   $\models n \wedge [Next]_v \Rightarrow n' \vee \bigvee_{\{(A,m) : (n,m) \in \delta_A\}} \langle A \rangle_v \wedge m'$
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) $\models n \Rightarrow$ 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* $(n,m) \notin \delta_A$.

2.3 Model Checking Predicate Diagrams

Viewing predicate diagrams as finite-state transition systems, temporal properties of their traces can be established using LTL model checking. We encode predicate diagrams in PROMELA, the modelling language of the model checker SPIN [5], as follows: two variables *node* and *action* keep track of the current node and the last action taken; they are updated nondeterministically according to the transition relation $\delta_=$. The predicates P in $\mathcal{P}$ are represented by Boolean variables that are updated according to the label of the current node—nondeterministically, if the label contains neither P nor $\neg P$. We also introduce variables $b_{(t,<)}$ for every pair $(t,<)$ that appears in some ordering annotation. These variables are set to 2 if the last transition taken was labelled by $(t,<)$, to 1 if it was labelled by $(t,\leq)$ or if a stuttering transition was taken, and to 0 otherwise.

The fairness conditions associated with the actions of a predicate diagram give rise to assumptions for the verification by SPIN, which are expressed as LTL formulas. We again consider an action A to be enabled whenever the current node has an outgoing edge in δ_A; this assumption is warranted by condition 4(c) of Proposition 2. Similarly, the effect of ordering annotations is expressed by assumptions of the form

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

that assert that any run during which t is known to have decreased infinitely often must also contain infinitely many transitions that may have increased t. An equivalent formulation is

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

Because formulas of type $\Diamond\Box F$ are conjunctive, the latter formulation gives rise to LTL assumptions with fewer temporal operators; this can be an advantage when SPIN generates the corresponding Büchi automaton during verification.

2.4 Refinement of Predicate Diagrams

In [2] we have studied concepts of refinement between predicate diagrams. This can be useful if diagrams are to be used during top-down development of protocols or, as in the present case study, to decompose a complex verification effort into smaller parts. Working in a linear-time setting, a diagrams G^1 refines a diagram G^2 if every trace through G^1 is also a trace through G^2. However, as in the case of Proposition 2, we are looking for a notion of refinement that can be checked "locally", avoiding temporal logic reasoning at the level of behaviors. The following definition, reproduced from [2], introduces a "structural" refinement concept. For simplicity, it assumes that the sets of predicates, action names, and ordering annotations that occur in the refining diagram are supersets of the respective sets in the refined diagram. A more general definition that allows for a change of representation has also been considered in [2].

Definition 3 *Assume given two predicate diagrams* $G^1 = (N^1, I^1, \delta^1, o^1, \zeta^1)$ *over predicates* $\mathcal{P}^1$ *and actions* $\mathcal{A}^1$ *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* $A \in \mathcal{A}^1$ *and all* $(n,m) \in \delta^1_A$*:*

   (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* $A \in \mathcal{A}^1$*, all* $(n,m) \in \delta^1_A(n,m)$*, all terms* t *and relations* $\prec \in O^=$*:*

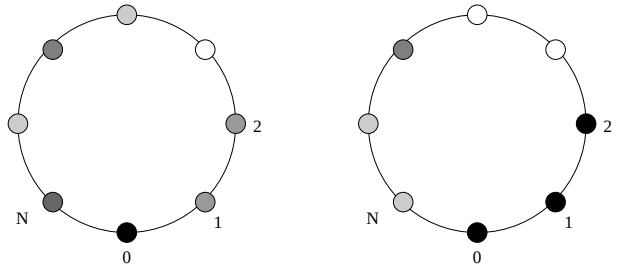
   (a) *if* $(t,\prec) \in o^2(f(n), f(m))$ *then* $(t,\prec) \in o^1(n,m)$*,*

   (b) *if* $f(n) = f(m)$ *and* $(t,\prec) \in o^2(f(n), m')$ *for some* $m' \in N^2$ *then* $(t,\preceq) \in o^1(n,m)$.

5. *For every run* $\rho^1 = n_0 \xrightarrow{A_0} n_1 \xrightarrow{A_1} \ldots$ *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* $\rho^1 = n_0 \xrightarrow{A_0} n_1 \xrightarrow{A_1} \ldots$ *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* $i \in \mathbb{N}$.

During refinement, nodes of the "abstract" diagram G^2 will be split into several nodes (distinguished by newly introduced predicates) in the "concrete" diagram G^1; the



(a) Initial configuration. (b) Later configuration.

Figure 1: Evolution of Dijkstra's protocol.

association between nodes of the two diagrams is given by the refinement mapping $f : N^1 \to N^2$. Conditions (1)–(4) of Definition 3 are "structural" in that they can either be checked from the graph structure or can be verified using non-temporal reasoning. On the other hand, conditions (5) and (6) ensure that the refinement notion is flexible. In particular, they allow high-level fairness conditions to be implemented by any combination of fairness conditions and ordering annotations at the lower level. Because they are formulated at the level of runs, these conditions can be established by model checking along the lines explained in section 2.3.

Correctness of structural refinement is asserted by the following proposition [2]. It ensures that all temporal properties shown of G^2 remain valid for G^1.

Proposition 4 *If G^1 structurally refines G^2 then every trace through G^1 is a trace through G^2.*

3 Proving Self-Stabilization

3.1 Dijkstra's Protocol

A system is self-stabilizing if it will reach some "stable" state (and then remain "stable"), no matter what state it is started in. Self-stabilizing protocols are useful for initialization and for error-recovery after faults. Dijkstra [3] introduced the problem at the hand of a ring of $N+1$ processors numbered 0 to N. Each processor i is equipped with a register $v[i]$ that can hold values in the range $0,\ldots,M$ where $M \geq N$. Figure 1(a) illustrates a possible configuration where different shades of grey represent different register values.

The processors operate according to the following rules: processor 0 can make a move whenever its register holds the same value as that of its left-hand neighbor, processor N. It then increments its register (modulo $M+1$). Any other processor $i+1$ can make a move when the value in its register is different from the value held by its left-hand

MODULE *DIJKSTRA*

EXTENDS *Naturals*
CONSTANTS N, M
VARIABLES v

ASSUMPTIONS $M \geq N$

$SITES \triangleq \{ x \in Nat : 0 \leq x \wedge x \leq N \}$
$NUM \triangleq \{ x \in Nat : 0 \leq x \wedge x \leq M \}$

$Zero \triangleq \wedge v[0] = v[N]$
$\quad \wedge v' = [v \text{ EXCEPT } ![0] = @+1 \bmod (M+1)]$
$NonZero(i) \triangleq \wedge v[i+1] \neq v[i]$
$\quad \wedge v' = [v \text{ EXCEPT } ![i+1] = v[i]]$

$Init \triangleq v \in [SITES \to NUM]$
$Next \triangleq \vee Zero$
$\quad \vee \exists\, i \in SITES \setminus \{N\} : NonZero(i)$

$Spec \triangleq Init \wedge \Box\, [Next]_v \wedge \mathrm{WF}_v(Next)$

Figure 2: A TLA$^+$ specification of Dijkstra's protocol.

neighbor, processor i. Processor $i+1$ then copies the value of $v[i]$ into its own register. The ability to make a move is interpreted as the possession of a token; the system is in a stable state when exactly one processor has a token.

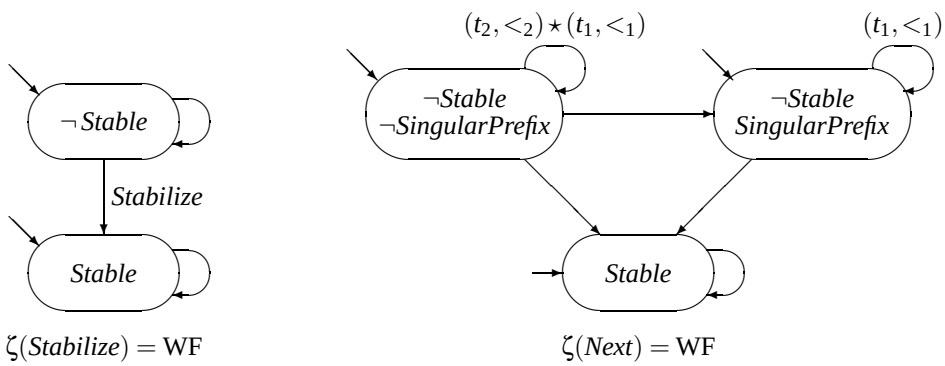
A pseudo-code representation of the system is shown below; a TLA$^+$ specification of the protocol appears in figure 2.

var v : **array** $[0\,..\,N]$ **of** $[0\,..\,M]$

$$\begin{array}{ll} & v[0] = v[N] \longrightarrow v[0] := (v[0]+1) \bmod (M+1) \\ \Box_{i<N} & v[i+1] \neq v[i] \longrightarrow v[i+1] := v[i] \end{array}$$

Fig. 1(b) shows a configuration that can be reached from the configuration of Fig. 1(a) after a few steps. Observe that at least one processor is enabled (i.e., has a token) in any configuration: for if all non-zero processors are disabled then $v[i] = v[i+1]$ holds for all $0 \leq i < N$. Therefore, $v[0] = v[N]$, implying that processor 0 is enabled. Stable configurations are those in which exactly one processor can move, which is true precisely if for some i, all processors $j \leq i$ hold the value $v[0]$ while all processors $j > i$ hold the value $v[N]$. In particular, configurations where all registers hold the same value are stable. It is also easy to see that whenever a stable state has been reached, the token will circulate counter-clockwise through the ring, and that all subsequent configurations will also be stable.

It was only a decade after the publication of the proto-



(a) Abstract view $\mathcal{D}_{abs}$.

(b) Refinement $\mathcal{D}_{conc}$ for Dijkstra's protocol.

Figure 3: Predicate diagrams to express self-stabilization.

col [4] that Dijkstra published a proof demonstrating its correctness, acknowledging that it was not entirely trivial. In 1998, Qadeer and Shankar [9] published a machine-checked verification of the algorithm using the interactive theorem prover PVS. They reported considerable difficulty both in constructing a convincing informal proof and in its formalization. One of us [7] has formalized a somewhat more economical proof in Isabelle/HOL [8]. We now reexamine the latter proof in terms of predicate diagrams. The benefits of such an exposition are twofold: first, the structure of the proof becomes clearer and second, part of the verification effort can be automated through the use of model checking.

3.2 Verification of Self-Stabilization

The property of self-stabilization can be abstractly represented by the predicate diagram $\mathcal{D}_{abs}$ of figure 3(a): initially, the system can either be in a stable or unstable state. Transitions from a stable to an unstable state are not allowed; eventual stabilization is enforced by placing a fairness condition on an action named *Stabilize* that causes a transition from unstable to stable states. (By convention, we suppress the label *Next* for transitions that correspond to the system's next-state relation.) Running SPIN on the encoding of the diagram confirms:

Theorem 5 *Every trace through $\mathcal{D}_{abs}$ satisfies $\Diamond\Box$Stable.*

Diagram $\mathcal{D}_{abs}$ concisely represents the desired property, but it is too simple. Formally, it does not conform to the formula *Spec* of figure 2 that represents Dijkstra's protocol: there is no action corresponding to *Stabilize* that would be enabled in all unstable states and lead to immediate stabilization. We have to find a refinement of $\mathcal{D}_{abs}$ that conforms to *Spec*.

Informally, the correctness of Dijkstra's protocol can be explained as follows. Consider first a configuration where processor 0 holds a value that does not occur in any other register (such a configuration is depicted in figure 1(a)) or, more generally, a configuration where $v[0] = v[1] = \ldots = v[i]$ holds for some $i \leq N$, but $v[j] \neq v[0]$ holds for all j where $i < j \leq N$. We say that such a configuration has a *singular prefix*. In such a configuration, processor 0 cannot move unless $v[0] = v[N]$, which means that the prefix extends around the entire ring and the system has stabilized. The non-zero processors in the prefix are disabled in such a configuration. On the other hand, moves of processors outside the prefix do not introduce fresh values and will eventually extend the singular prefix around the ring.

It is therefore enough to show that some configuration with a singular prefix will be reached from any initial configuration, unless a stable configuration is reached before. Now, observe that the number $M+1$ of possible register values has been assumed to be at least the number of processors. By the pigeonhole principle it follows that some value $r \in \{0,\ldots,M\}$ does not initially occur among $v[1],\ldots,v[N]$. Assuming that processor 0 moves infinitely often, thereby incrementing its register, there will be a first subsequent configuration where $v[0] = r$. In that configuration it must still be the case that $v[i+1] \neq r$ for all i because only moves of processor 0 introduce fresh values; thus we have reached a configuration with a singular prefix. It thus remains to show that processor 0 is guaranteed to move infinitely often. But assume that processor 0 does not move from some state on: then the actions of the non-zero processors will ensure that $v[0]$ spreads around the ring, and thus a stable configuration will be reached.

This idea can be formally represented by the predicate diagram $\mathcal{D}_{conc}$ shown in figure 3(b). Like the previous diagram $\mathcal{D}_{abs}$, it does not allow transitions from stable

MODULE *STABILITY*

EXTENDS *DIJKSTRA*

$$
\begin{array}{l}
\mathit{stable}(i) \triangleq \ \wedge\ \forall j \in \mathit{SITES} : j \leq i \Rightarrow v[j] = v[0] \\
\qquad\qquad\quad \wedge\ \forall j \in \mathit{SITES} : j > i \Rightarrow v[j] = v[N] \\
\mathit{Stable} \triangleq \exists i \in \mathit{SITES} : \mathit{stable}(i) \\
\mathit{SP}(i) \triangleq \ \wedge\ \forall j \in \mathit{SITES} : j \leq i \Rightarrow v[j] = v[0] \\
\qquad\qquad\quad \wedge\ \forall j \in \mathit{SITES} : j > i \Rightarrow v[j] \neq v[0] \\
\mathit{SingularPrefix} \triangleq \exists i \in \mathit{SITES} : \mathit{SP}(i) \\
t_1 \triangleq [\, i \in \mathit{SITES} \setminus \{N\} \mapsto \text{IF } v[i] = v[i{+}1] \text{ THEN } 1 \text{ ELSE } 0\,] \\
t_2 \triangleq \text{LEAST } k : \neg\exists i \in \mathit{SITES} \setminus \{0\} : v[i] = (v[0]{+}k) \bmod (M{+}1)
\end{array}
$$

Figure 4: Definitions added for the proof.

to unstable configurations. Similarly, it rules out moves from unstable configurations with a singular prefix to unstable configurations without a singular prefix. However, the fairness condition of $\mathcal{D}_{abs}$ has been replaced with a weaker progress assumption: weak fairness of the system's next-state relation asserts that the protocol should not stop. Also, ordering annotations on transitions have been introduced whose meaning will be explained below. Definition 3 of structural refinement turns out to be liberal enough to allow for such non-local refinement of fairness. Obviously, the refinement mapping f is defined to map the two upper-hand nodes of $\mathcal{D}_{conc}$ to the single upper-hand node of $\mathcal{D}_{abs}$, and the lower-hand node of $\mathcal{D}_{conc}$ to the lower-hand node of $\mathcal{D}_{abs}$.

Theorem 6 *$\mathcal{D}_{conc}$ structurally refines $\mathcal{D}_{abs}$.*

Proof. Conditions (1)–(4) of Definition 3 hold trivially. In particular, condition (3) is true because $\mathcal{D}_{conc}$ does not allow any transitions that are disallowed by $\mathcal{D}_{abs}$, and condition (4) holds because $\mathcal{D}_{abs}$ does not contain any ordering annotations that would have to be preserved.
Moreover, there are no actions with strong fairness, and therefore only condition (5) needs to be verified. Now, the ordering annotations (for any well-founded orderings $<_1$ and $<_2$) ensure that any run through $\mathcal{D}_{conc}$ must eventually reach the node labelled *Stable*, and then remain there forever. In particular, the run contains infinitely many nodes whose image disables the *Stabilize* action, which implies the fairness condition. In fact, a run of SPIN suffices to prove the condition. Q.E.D.

To conclude the verification of Dijkstra's protocol, it remains to show that $\mathcal{D}_{conc}$ conforms to the protocol specification. In particular, we must define the predicates that appear as node labels of $\mathcal{D}_{conc}$, and the terms t_1 and t_2 and their associated well-founded orderings $<_1$ and $<_2$. Figure 4 collects the definitions of these predicates and terms as a TLA$^+$ module. The definitions of *Stable* and *SignularPrefix* have been motivated before. The term t_1 records progress that can be attributed to moves of the non-zero processors: it is defined as a bit vector indicating which of the non-zero processors are enabled. Because it is obvious that every move of a non-zero processor disables that processor and does not affect the enabledness of its neighbors to the left, the corresponding well-founded ordering $<_1$ is chosen as the lexicographic ordering on bit vectors of length N. The term t_2, which records progress brought about by moves of processor 0, is defined as the "distance" to the (cyclically) smallest value that does not occur among the register values of processors that are part of the singular prefix. As we have motivated before, a configuration with singular prefix is reached when that distance becomes zero, and therefore $<_2$ is simply chosen as the standard ordering on natural numbers.

Theorem 7 *$\mathcal{D}_{conc}$ conforms to formula Spec of figure 2.*

Proof. We consider the conditions of Proposition 2, working in the context of module *STABILITY*, which extends module *DIJKSTRA* by the necessary definitions to interpret the annotations that appear in diagram $\mathcal{D}_{conc}$. Condition (1) holds because the disjunction of the labels of initial states is trivial. For condition (2), it has to be shown that all actions preserve stability and the existence of a singular prefix, respectively. This is proven by considering the possible moves as defined by formula *Next* of module *DIJKSTRA*: every move disables the processor that made the move, it may enable its right-hand neighbor, and does not affect the enabledness of any other processor.
Establishing condition (3) is harder. It follows from the following observations:

- Every step of a non-zero processor i disables itself but does not affect the enabledness of processors j

for $0 < j < i$. Therefore, such steps decrease t_1.

- Steps of non-zero processors do not increase the value of t_2: the set $\{v'[1], \ldots, v'[N]\}$ of register values after the step is a subset of the set $\{v[1], \ldots, v[N]\}$ of values before the step. Therefore, if $(v[0]+k) \bmod (M+1)$ did not occur in the set of register values before the step, it still does not occur in the set of register values thereafter, and the smallest such value k could only have decreased.

- Every action of processor 0 from an unstable state without a singular prefix decreases the value of t_2: if there is no singular prefix, the value $v[0]$ must occur as $v[i+1]$, for some i, and therefore t_2 must be positive. Now, an action of processor 0 increments $v[0]$ but leaves all other registers unchanged, hence t_2 decreases by one.

Condition (4a) is obvious and condition (4b) is vacuously true. Condition (4c) holds because some processor is enabled in all states, as we have argued before. Condition (4d) is a consequence of condition (2) that was already proven, because only one action occurs in the diagram. Overall, the conformance proof has been mechanized in Isabelle/HOL, requiring approximately 300 interactions. We found the proofs that rely on modulus arithmetic to be the hardest to carry out in Isabelle, because only little automation was provided. Q.E.D.

4 Conclusions

We have presented a correctness proof for Dijkstra's protocol of self-stabilization in terms of predicate diagrams. Self-stabilizing algorithms are usually considered quite challenging for formal verification, and our experience is no exception. Nevertheless, we have found it helpful to represent the structure of the proof in the form of predicate diagrams, which give it a comprehensible visual representation. From a more technical point of view, predicate diagrams provide an interface between interactive and automatic verification techniques. We found it particularly encouraging that the refinement of an abstract-level fairness condition by an implementation based on well-founded orderings could be proved entirely by model checking whereas interactive theorem proving only had to be applied when reasoning about individual transitions. In contrast, the "flat" proofs in [7, 9] required rather elaborate theorem proving at the temporal level, which made a significant contribution to the overall effort.

We are currently working on an integrated set of tools to support verification and development based on predicate diagrams. These tools include graphical editors as well as components based on abstract interpretation, model checking, and automatic and interactive theorem proving technology. All components share an XML-based format to represent the intermediate documents in order to give the user the flexibility to apply whichever style of development is appropriate for the problem at hand.

References

[1] D. Cansell, D. Méry, and S. 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, Nov. 2000. Springer-Verlag.

[2] D. Cansell, D. Méry, and S. Merz. Diagram refinements for the design of reactive systems. *Journal of Universal Computer Science*, 7(2):159–174, 2001.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Nov. 1974.

[4] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

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

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

[7] 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.

[8] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1994. See also the Isabelle home page at `http://isabelle.in.tum.de/`.

[9] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In D. Gries and W.-P. de Roever, editors, *Programming Concepts and Methods*, pages 424–443, Shelter Island, N.Y., June 1998. Chapman & Hall.