DIXIT: a Graphical Toolkit for Predicate Abstractions

Loïc Fejoz, Dominique Méry, Stephan Merz

DIXIT: a Graphical Toolkit for Predicate Abstractions

Loïc Fejoz Dominique Méry Stephan Merz

LORIA UMR 7503, Nancy, France
{Loic.Fejoz,Dominique.Mery,Stephan.Merz}@loria.fr

Abstract

We describe a toolkit to support the use of predicate diagrams, a representation of predicate abstractions that includes annotations for proving liveness properties. Centered around a graphical editor for drawing predicate diagrams, proof obligations for proving correctness of the abstraction w.r.t. TLA$^+$ system specifications can be generated, correctness properties expressed in temporal logic can be verified by model checking, and counterexamples can be visualized. The toolkit also supports stepwise development of systems, based on a notion of refinement of predicate diagrams.

Key words: predicate abstraction, model checking, theorem proving, liveness, fairness, TLA+

1 Introduction

The use of predicate abstractions has become popular for the verification of infinite-state systems. By focusing on a finite set of predicates of interest, the system under investigation can be finitely represented, and finite-state model checking can establish run-time properties. Moreover, predicate abstractions are natural to define: candidate predicates include those appearing in the description of a system (initial conditions, action guards, case distinctions etc.). We have proposed the format of *predicate diagrams* [3] for the representation of predicate abstractions, with a particular focus on the verification of liveness properties via annotations representing fairness assumptions and well-founded orderings. In this paper, we describe the DIXIT toolkit that supports the construction and analysis of predicate diagrams. Its present version is mainly oriented towards the verification of TLA$^+$ models [6], but the architecture is intended to adapt easily to other modeling or programming languages. After a concise overview of predicate diagrams and of the DIXIT implementation, we illustrate its capabilities at the hand of the well-known Alternating Bit Protocol.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

2 Predicate Diagrams

2.1 Predicate diagrams for verification

A predicate diagram is a finite graph whose nodes are labeled with sets of literals. A node represents the set of states satisfying the formulas contained in the node label. Edges in the diagram represent possible state transitions and are labeled with action names. A fairness condition (weak fairness or strong fairness) may be associated with each action name. Moreover, edges may be labeled with annotations of the form $(t, \prec)$ or $(t, \preceq)$ asserting that the term t decreases, or does not increase, with respect to the well-founded ordering $\prec$. Formally, predicate diagrams are defined as follows:

Definition 2.1 A predicate diagram $G = (\mathcal{P}, \mathcal{A}, O, N, I, \delta, \zeta, o)$ is given by

- finite sets $\mathcal{P}$, $\mathcal{A}$, and O of (symbols for) predicates, actions, and orderings; we write $\overline{\mathcal{P}}$ to denote the set of literals formed from $\mathcal{P}$ and write $O^{=}$ for the set containing $\prec$ and $\preceq$ for all $\prec \in O$,
- a finite set $N \subseteq 2^{\overline{\mathcal{P}}}$ of nodes, a subset $I \subseteq N$ of which are initial; if $n \in N$ is a node, we sometimes also write n to denote the conjunction of the formulas in n,
- a family $(\delta_A)_{A \in \mathcal{A}}$ of relations $\delta_A \subseteq N \times N$; we denote by δ the union of the relations δ_A, and by $\delta_=$ the reflexive closure of that union,
- a mapping $\zeta : \mathcal{A} \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ that associates with each action name a fairness condition (no fairness, weak fairness, or strong fairness), and
- an edge labeling o that associates a finite set of pairs $(t, \prec)$ of terms t and symbols $\prec \in O^{=}$ to each edge in δ.

Predicate diagrams represent the possible runs of a system: a run through the diagram is an ω-sequence of system states s_i, diagram nodes n_i and actions A_i such that each state satisfies the label of n_i, A_i relates n_i and n_{i+1}, and all edge annotations are satisfied. Besides the transitions that correspond to edges in the diagram, we also allow for stuttering transitions that remain at a node, preserving all its predicates (but not necessarily the values of the underlying variables). Infinite stuttering is prevented by appropriate fairness assumptions. A trace through the diagram is an ω-sequence of states for which there exists a run. The precise definition of runs and traces appears in [3].

We say that a predicate diagram G conforms to a system specification $Spec$ if every possible run of $Spec$ is a trace through G. DIXIT is currently oriented towards the verification of specifications written in TLA$^+$ [6], which are usually of the form $Init \wedge \Box[Next]_v \wedge L$ where $Init$ is a state predicate describing the initial condition, $Next$ is an action formula that specifies the next-state relation, v is a tuple containing all state variables, and L is a conjunction of fairness conditions. We can formally compare a predicate diagram G and a TLA$^+$ system specification $Spec$ that appears in a TLA$^+$ module which also defines interpretations for the symbols in the sets $\mathcal{P}$, $\mathcal{A}$, and O. In fact, a set of essentially non-temporal proof obliga-

tions expressing correct initialization, state consecution, and respect of ordering annotations (see Prop. 3 of [3]) are sufficient to establish conformance. Temporal reasoning may be required for proving that $Spec$ implies the fairness assumptions of the diagram, however, in many cases, the same conditions appear in both models.

On the other hand, correctness properties expressed in linear-time temporal logic (including TLA) and built from the atomic predicates in $\mathcal{P}$, can be verified over a predicate diagram G by model checking. To do so, G is considered as a finite-state transition system defining a set of fair runs. The predicates in $\mathcal{P}$ are encoded as Boolean variables, and ordering annotations $(t, \prec)$ give rise to hypotheses

$$\Box\Diamond(t' \prec t) \Rightarrow \Box\Diamond\neg(t' \preceq t)$$

which are an expression of the well-foundedness condition in temporal logic and can be verified in the finite-state representation by tracking, for every transition taken, whether it carries an annotation indicating that t decreases w.r.t. $\prec$ or $\preceq$. The current version of DIXIT uses the SPIN [5] or LWAASPIN [4] model checkers. (LWAASPIN is less sensitive to the size of the formulas, which is important in the presence of many fairness and ordering annotations.)

2.2 Refinement of predicate diagrams

Beyond their use in system verification, predicate diagrams can also be employed to compare two models of the same system at different levels of abstraction. We say that a predicate diagram G^1 refines a predicate diagram G^2 if every trace through G^1 is also a trace through G^2. We assume that the sets of predicates, action names, and orderings underlying G^1 extend the corresponding sets underlying G^2. We are again interested in "local" conditions that establish refinement of diagrams, and the following notion of structural refinement of predicate diagrams underlies DIXIT.

Definition 2.2 Let $G^i = (\mathcal{P}^i, \mathcal{A}^i, O^i, N^i, I^i, \delta^i, o^i, \zeta^i)$ for $i = 1, 2$ be two predicate diagrams and $f : N^1 \to N^2$ be a mapping. G^1 structurally refines G^2 w.r.t. f iff the following conditions hold:

(i) $\mathcal{P}^2 \subseteq \mathcal{P}^1$, $\mathcal{A}^2 \subseteq \mathcal{A}^1$, $O^2 \subseteq O^1$,

(ii) $\models n \Rightarrow f(n)$ for every node $n \in N^1$,

(iii) $f(n) \in I^2$ for all $n \in I^1$,

(iv) for all n, m, A such that $(n, m) \in \delta^1_A$ we have $(f(n), f(m)) \in \delta^2{}_A$ if $A \in \mathcal{A}^2$, and $(f(n), f(m)) \in \delta^2{}_=$ otherwise,

(v) for all n, m, A such that $(n, m) \in \delta^1_A$ and all $\prec \in (O^2)^=$:
- $(t, \prec) \in o^1(n, m)$ whenever $(t, \prec) \in o^2(f(n), f(m))$,
- $(t, \preceq) \in o^1(n, m)$ whenever $f(n) = f(m)$ and $(t, \prec) \in o^2(f(n), m')$ for some $m' \in N^2$.

(vi) 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 $\zeta^2(A) = \text{WF}$, either $A_i = A$ or $f(n_i) \notin \text{dom}(\delta^2{}_A)$ holds for infinitely many $i \in \mathbb{N}$.

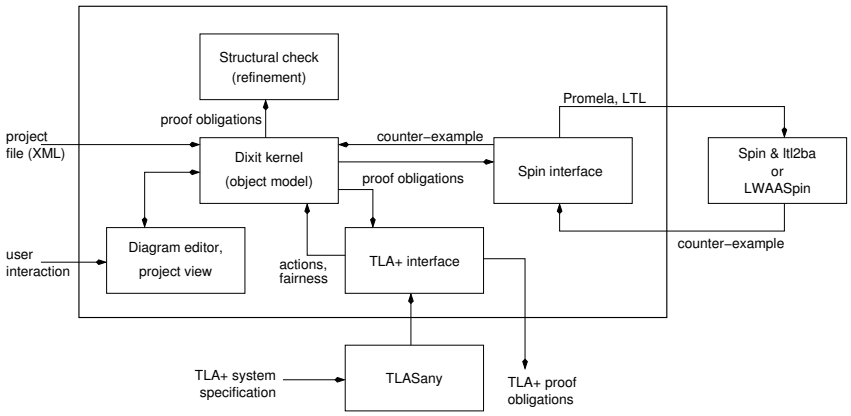


Fig. 1. DIXIT tool architecture.

(vii) 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 $A_i = A$ holds for infinitely many $i \in \mathbb{N}$ or $f(n_i) \in \mathrm{dom}(\delta^2{}_A)$ for only finitely many $i \in \mathbb{N}$.

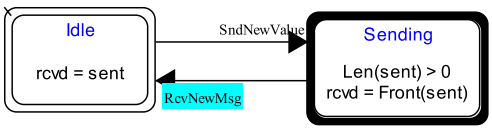
The conditions (ii)–(v) require the transition graphs of the two diagrams to be closely related: node labels of G^1 imply those of the corresponding nodes of G^2, initial nodes of G^1 are related to initial nodes of G^2, transitions in G^1 must map to (possibly stuttering) transitions in G^2, and ordering annotations already present in G^2 must be preserved in G^1. In contrast, the conditions (vi) and (vii) ensure that fair runs through the refining diagram can be mapped to fair runs through the refined diagram without requiring a syntactic preservation of high-level fairness conditions. Establishing structural refinement relies on three kinds of verification, beyond the purely syntactic check of condition (i): the conditions (iii)–(v) can be verified by inspection of the graph structure and the annotations of the diagrams. Condition (ii) requires (non-temporal) theorem proving, whereas conditions (vi) and (vii) can be verified by model checking the finite transition systems generated from the diagrams. At any rate, structural refinement ensures refinement of traces, and thus preservation of temporal-logic properties [3]:

Theorem 2.3 *If G^1 refines G^2 w.r.t. some node mapping f, then every trace through G^1 is also a trace through G^2.*

3 Functionality and Architecture of DIXIT

The DIXIT toolkit is intended to assist a user in performing the kind of reasoning illustrated in Sect. 2. Predicate diagrams can be drawn in a graphical editor, either from scratch or as a structural refinement of an existing diagram. In the latter case, the node mapping is defined implicitly, as no new nodes may be added, but existing nodes may be split.

Temporal logic properties (expressed over the set of literals that appear in the node labels) can be verified by model checking as discussed in Sect. 2.1. Similarly, the conditions (vi) and (vii) of Def. 2.2 that relate to the refinement of high-level



(a) Predicate diagram.

MODULE *DataTransmission*

EXTENDS *Naturals*, *Sequences*

VARIABLES *sent*, *rcvd*

$Front(s) \triangleq [i \in 1..Len(s)-1 \mapsto s[i]]$

$Last(s) \triangleq s[Len(s)]$

$front\_sent \triangleq Front(sent)$

(b) Associated TLA$^+$ module.

Fig. 2. High-level protocol model.

fairness conditions, can be established by model checking. Any counter-examples reported by the model checker are visualized in the graphical editor.

DIXIT can also generate proof obligations that ensure that a diagram conforms to a TLA$^+$ system specification associated with the diagram. These obligations are output as a TLA$^+$ module; they can currently not be proven within DIXIT, as we have not yet implemented an interface for an external theorem proving component.

Figure 1 illustrates the architecture of the toolkit. The DIXIT kernel is responsible for updating the internal representation of a project. The main point of user interaction is via the graphical editor, derived from the GEF framework (http://gef.tigris.org/). Verification steps can be initiated from a hierarchical project view in a separate window. The kernel interacts with external verification tools through well-defined interfaces. It generates proof obligations for theorem provers, model checkers, as well as structural conditions that can be verified at the diagram level itself. Currently, DIXIT is oriented towards the analysis of TLA$^+$ models, and therefore it interacts with the TLA$^+$ parser TLASANY, but the architecture should adapt easily to other modeling languages. Externally, a DIXIT project is stored in XML format; it may also include (pointers to) files that are not processed by the kernel, such as TLA$^+$ modules. Diagrams can be exported in Postscript, GIF, and SVG formats.

4 Alternating Bit Protocol in DIXIT

4.1 First abstraction: data transmission

The purpose of the Alternating Bit Protcol is to transmit a sequence of data from a sender to a receiver process via an unreliable channel. In a first abstraction we represent the state by two variables *sent* and *rcvd* that represent the history of data sent and received. The predicate diagram [1] shown in Fig. 2(a) describes a first abstraction of the protocol; in particular, it expresses that a new data item is sent only if the preceding one has been received.

[1] This diagram, as well as the following ones, has been exported from the DIXIT tool. Nodes with a hollow border represent initial nodes of the diagram. A cyan (resp., magenta) background for an action name indicates that weak (resp., strong) fairness is assumed.

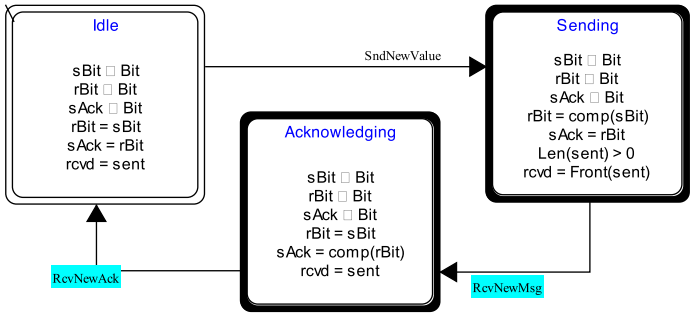


Fig. 3. Adding synchronization bits.

The left-hand node, named Idle, is labeled by the predicate $rcvd = sent$ indicating that all data that has been sent has been received successfully; this is also the initial node. The sender may now send a new value, resulting in a state represented by the right-hand node Sending and satisfying $rcvd = front\_sent$.[2]

With this diagram we associate the TLA$^+$ module $DataTransmission$ shown in Fig.2(b) that fixes the interpretation of the terms and predicates of the predicate diagram. The module is based on the TLA$^+$ library modules $Naturals$ and $Sequences$ that define natural numbers and finite sequences. Next, the module declares the variables $sent$ and $rcvd$ and defines the operators $Front$ and $Last$ such that for a non-empty sequence s, $Front(s)$ is the subsequence without the last element, and $Last(s)$ is the last element of s. Finally, $front\_sent$ is defined as $Front(sent)$. The module does not define the actions $SndNewValue$ and $RcvNewMsg$ that appear in the diagram: whereas such definitions are necessary for verifying conformance w.r.t. a system specification, they can be left abstract for the verification of temporal properties of a high-level model or for verifying refinements.

We assume weak fairness for the action $RcvMsg$, and therefore every message sent must eventually be received. Formally, the LTL property $\Box\Diamond(rcvd = sent)$ holds of the diagram and can be verified within DIXIT. In particular, it is guaranteed to be preserved by any refinement of this diagram.

4.2 Intermediate Model: Synchronization Bits

At the preceding level of abstraction, data transmission was modeled as being instantaneous. We now introduce an explicit acknowledgement phase and augment the state space by three bits to ensure synchronization. We do not yet explicitly represent message and acknowledgement channels, nor message loss.

The predicate diagram shown in Fig. 3 is obtained as a refinement of the diagram of Fig.2(a) by splitting node Idle, resulting in the nodes Idle and Acknowledging, and by adding predicates concerning the bits. In state Idle, all bits are identical. The sender inverts the value of $sBit$ upon sending a new value whereas $rBit$ and $sAck$ remain equal. A successful message reception restores the equality of $rBit$ and $sBit$, causing $sAck$ to be the complement of $rBit$. The action $RcvNewAck$

[2] DIXIT restricts the predicates that appear in a node to be either identifiers or of the form $id_1 = id_2$ or $id_1 \in id_2$ for identifiers id_1 and id_2, or negations of such predicates.

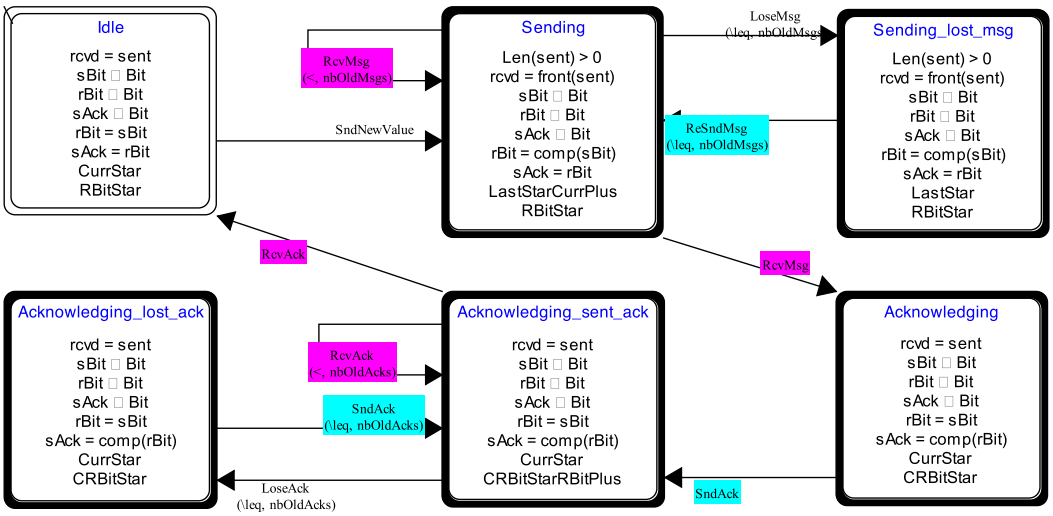


Fig. 4. Third Refinement: predicate diagram for the Alternating Bit Protocol

(which did not yet exist at the previous level of abstraction) restores the equality of all three bits and leads back to state Idle. The set Bit and the associated operations are again defined in an associated TLA$^+$ module, not shown here for brevity.

Using DIXIT, we can verify that the new diagram refines the previous one: first, a structural test confirms that initial nodes of the refining diagram are related to initial nodes of the refined diagram and that transitions at the lower level respect the transitions that existed before—in particular, the new transition $RcvNewAck$ refines a stuttering transition at the abstract level. Second, proof obligations are generated to show that the predicates appearing in the refined nodes imply those of the corresponding abstract nodes. In our example, these obligations are trivially satisfied because we have only added new predicates. Third, model checking ensures that the abstract-level fairness properties are preserved.

4.3 Final Refinement: Alternating Bit Protocol

At the next level of refinement, we introduce transmission channels and message loss. We obtain the predicate diagram shown in Fig. 4, which conforms to a TLA$^+$ specification of the Alternating Bit Protocol, parts of which are shown in Fig. 5. The predicate diagram has been obtained by splitting the nodes Sending and Acknowledging and by adding predicates describing the shape of the message and acknowledgement channels at each node. For example, when the protocol is in a state described by node Idle, there may still be some copies of the current message and of the current acknowledgement on the channels, as expressed by the predicates $CurrStar$ and $RBitStar$, which are also defined in the TLA$^+$ module. After sending a new value, the previously current message becomes the old message, and there is at least one copy of the new message on the message channel; this is expressed by the predicate $LastStarCurrPlus$ in node Sending.

The nodes of the diagram reflect the different phases of the protocol, and the edges represent the possible transitions. For example, two actions are enabled from

MODULE *AlternatingBit*

EXTENDS *SyncBit*
VARIABLES *msgQ*, *ackQ*

$$Init \triangleq \land sent = \langle\rangle \land rcvd = sent \land msgQ = \langle\rangle \land ackQ = \langle\rangle$$
$$\land sBit \in Bit \land rBit = sBit \land sAck = rBit$$

$$SndNewValue \triangleq \exists\, d \in Value :$$
$$\land sAck = sBit \land sent' = Append(sent, d)$$
$$\land sBit' = comp\_sBit \land msgQ' = Append(msgQ, \langle d, sBit'\rangle)$$
$$\land \text{UNCHANGED } \langle rcvd, rBit, sAck, ackQ\rangle$$
$$\vdots$$

$$ABLive \triangleq \land \mathrm{WF}_{vars}(ReSndMsg) \land \mathrm{SF}_{vars}(RcvMsg)$$
$$\land \mathrm{WF}_{vars}(SndAck) \land \mathrm{SF}_{vars}(RcvAck)$$

$$ABSpec \triangleq Init \land \Box[ABNext]_{vars} \land ABLive$$

$$Star(e) \triangleq Seq(\{e\})$$
$$Plus(e) \triangleq Star(e) \setminus \{\langle\rangle\}$$
$$Concatenations(S, T) \triangleq \{s \circ t : s \in S, t \in T\}$$
$$StarStar(e1, e2) \triangleq Concatenations(Star(e1), Star(e2))$$
$$StarPlus(e1, e2) \triangleq Concatenations(Star(e1), Plus(e2))$$
$$CurrStar \triangleq$$
$$\text{IF } sent = \langle\rangle \text{ THEN } msgQ = \langle\rangle \text{ ELSE } msgQ \in Star(\langle Last(sent), sBit\rangle)$$
$$LastStarCurrStar \triangleq$$
$$\text{IF } Len(sent) \leq 1 \text{ THEN } CurrStar$$
$$\text{ELSE } msgQ \in StarStar(\langle sent[Len(sent) - 1], comp\_sBit\rangle, \langle Last(sent), sBit\rangle)$$
$$\vdots$$
$$nbOldMsgs \triangleq Cardinality(\{i \in \text{DOMAIN } msgQ : msgQ[i] \neq Last(sent)\})$$
$$nbOldAcks \triangleq Cardinality(\{i \in \text{DOMAIN } ackQ : ackQ[i] \neq rBit\})$$

Fig. 5. Module AlternatingBit (excerpts).

node Sending. First, a message can be received by the receiver process. If there are still old messages on the channel, the first copy of them will be received, and we are back in node Sending, but one copy of the old messages has been consumed, and this is reflected by the edge annotation. Otherwise, the current message (with the current value of $sBit$) will be received, and the protocol moves to node Acknowledging. The second enabled action is that of message loss, represented by a transition to node Sending_lost_msg. Observe that in such a state the message queue may be empty, and so the action $RcvMsg$ is not necessarily enabled. However, the sender will eventually resend the current message, causing a move back to node Sending. The reasoning for the remaining transitions is similar.

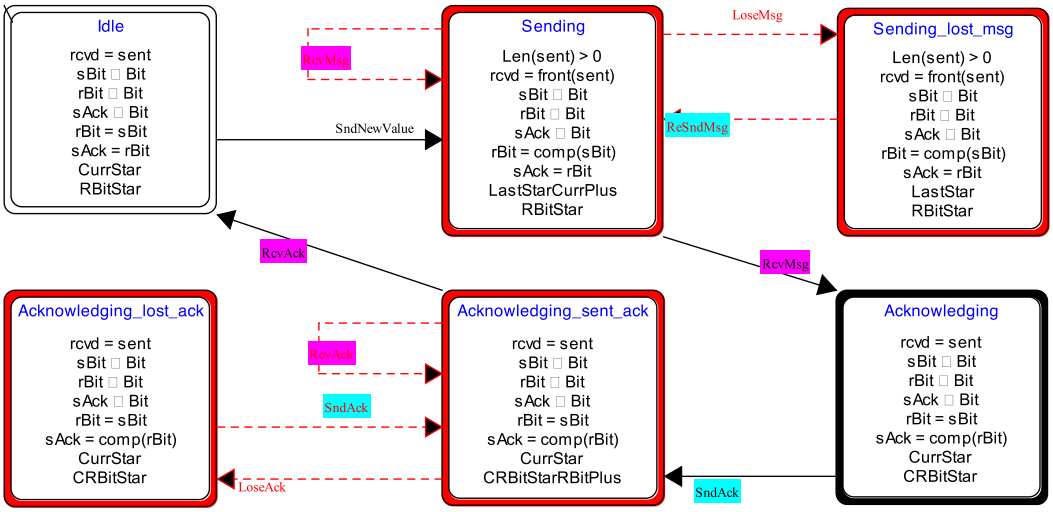


Fig. 6. Visualization of counter-examples by DIXIT.

We can again verify that the new predicate diagram is a correct refinement of the previous one. Whereas the structural refinement conditions and the implication of the predicates are again trivial, refinement of the previous fairness conditions is more interesting. In fact, the action $RcvMsg$ is not enabled in all nodes derived from node Sending at the previous level. This is compensated by an assumption of strong fairness for that action, reflecting the assumption that the channel does not lose all messages, as well as by adding ordering annotations for the number of "old" messages that are still on the channel. The interplay of fairness and ordering annotations is quite intricate, and DIXIT is of great help in determining the necessary annotations. For example, when we try to establish refinement for a diagram without the ordering annotations, the model checker produces a counter-example, which is visualized in Fig. 6: the suffix of a run violating the high-level fairness condition for the action $RcvNewMsg$ is highlighted in red.

Finally, DIXIT can generate proof obligations that ensure the conformance of the predicate diagram w.r.t. the TLA$^+$ module $AlternatingBit$. However, the current version of DIXIT lacks an external prover to discharge these proof obligations.

5 Conclusion

DIXIT is a toolkit to support the use of predicate abstractions for the verification of infinite-state systems, with a particular focus on proving liveness properties. The current version is centered around a GEF-based graphical editor. It enables a user to draw predicate diagrams and to verify correctness properties using model checking. Additionally, one can check that a predicate diagram is a correct refinement of another one, implying that all properties expressed in LTL are preserved. The proof obligations for establishing refinement require that the refining diagram faithfully reflects the transition structure of the abstract diagram. On the other hand, fairness conditions of the high-level diagram can be implemented in very flexible ways.

Additionally, DIXIT can generate proof obligations that ensure that a predicate diagram conforms to (i.e., is a correct abstraction of) a TLA$^+$ system specification.

Already at its present state, we have found DIXIT to be quite valuable for our teaching and research. In fact, predicate diagrams are intuitive enough to be understood quickly, and the model checking functionality offered by DIXIT is very useful to determine adequate fairness assumptions for a specification and to think about appropriate ordering annotations to break cycles. At present, the tool lacks a prover backend for proof obligations expressed in first-order logic. Perhaps even more interesting would be functionality to compute a predicate diagram for a given TLA$^+$ system specification and a set of predicates of interest. We have carried out preliminary work in this direction [2] and much inspiration can be gained from [1,7], but this is not currently implemented in DIXIT. We also intend to replace the use of off-the-shelf model checkers with specific adaptations of model checking algorithms to the structure and annotations of predicate diagrams. Finally, more experience with practical case studies will guide extensions of the format of predicate diagrams to make effective use of hierarchy and decomposition. DIXIT is freely available at `http://www.loria.fr/equipes/mosel/dixit/`.

References

[1] Ball, T. and S. K. Rajamani, *The SLAM project: Debugging system software via static analysis*, in: *Principles of Programming Languages (POPL 2002)*, 2002, pp. 1–3.

[2] Cansell, D., D. Méry and S. Merz, *Predicate diagrams for the verification of reactive systems*, in: *2nd Intl. Conf. Integrated Formal Methods (IFM 2000)*, Lecture Notes in Computer Science **1945** (2000), pp. 380–397.

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

[4] Hammer, M., A. Knapp and S. Merz, *Truly on-the-fly LTL model checking*, in: N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Lecture Notes in Computer Science **3440** (2005), pp. 191–205.

[5] Holzmann, G. J., "The SPIN Model Checker," Addison-Wesley, 2003.

[6] Lamport, L., "Specifying Systems," Addison-Wesley, Boston, Mass., 2002.

[7] Podelski, A. and A. Rybalchenko, *Transition predicate abstraction and fair termination*, in: *32nd ACM Symp. Principles of Programming Languages (POPL2005)* (2005), pp. 132–144.