DIXIT: a Graphical Toolkit for Predicate Abstractions

Loïc Fejoz, Dominique Méry, and Stephan Merz

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

Abstract. We describe a toolkit to support the use of predicate diagrams, a visual representation of predicate abstractions. 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.

1 Predicate Diagrams

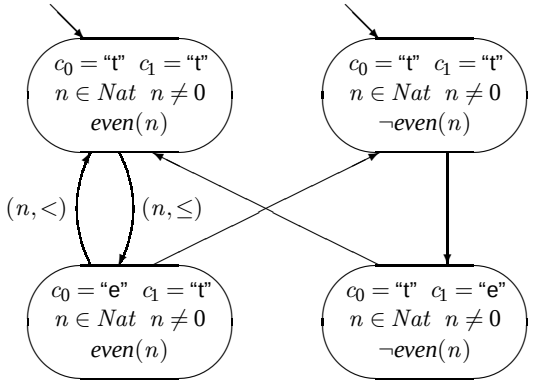
The idea of using predicate abstractions has become very popular for the verification of infinite-state systems. By focusing on a finite set of predicates of interest, the system under investigation can be finitely represented, and automatic methods for the verification of finite-state systems can thus be employed. Moreover, it is quite natural to define an abstraction in terms of predicates; in simple cases, candidate predicates can even be gleaned from the system description (e.g., initial conditions, action guards or case distinctions). We have proposed the format of *predicate diagrams* [3] for the verification and development of reactive and distributed systems, with a particular focus on the verification of liveness properties based on annotations capturing fairness properties and well-founded orderings. In this paper, we describe a toolkit that supports the construction and analysis of predicate diagrams. The present version is mainly oriented towards the verification of TLA$^+$ models, but it is intended to be easily adaptable to other modeling, or even programming, languages.

We do not reiterate a formal definition of predicate diagrams, but introduce them by way of a simple example, the "dining mathematicians" problem introduced in [4]. Figure 1(a) contains (the main part of) a TLA$^+$ specification of the system, consisting of two processes (whose control states are represented by the variables c_0 and c_1) that communicate via a shared integer variable n. Each process can be in either thinking ("t") or eating ("e") state. The variable n, initialized to some positive integer, controls access to the eating states: process 0 may eat when n is even and divides n by 2 when returning to state "t". Conversely, process 1 may eat when n is odd and assigns $3 * n + 1$ to n when it stops eating. The purpose of the protocol is to ensure mutual exclusion of the "e" states without introducing starvation for either process.

A predicate diagram for this system appears in Fig. 1(b). It consists of four nodes, each labeled with a set of literals. The two top nodes are initial; in fact, while the control states are fixed and n is known to be a non-zero natural number, it can be even or odd. Considering the top left-hand node, it is easy to see that only possible successor state

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

(a) TLA$^+$ specification



(b) Predicate diagram

Fig. 1. The “dining mathematicians” example.

is represented by the lower left-hand node, corresponding to the occurrence of action Eat_0, which sets c_0 to “e” without changing c_1 or n. In particular, n is still positive and even. From there, only the action Thk_0 is possible, and will lead back to a state where both processes are in their “t” states. Moreover, since n must be at least 2 in the source state, $n/2$ is at least 1, so n is still a positive integer. However, it could be even or odd, as represented by the two abstract transitions of the diagram. The justification of the remaining transitions is similar. Formally, the correctness of the predicate diagram w.r.t. the system specification of Fig. 1(a) can be justified by discharging a set of non-temporal proof obligations defined in [3].

A predicate diagram represents every possible behavior of the system, and properties (over the predicates represented in the abstraction) can therefore be verified by model checking. For example, it is easy to prove mutual exclusion ($\Box\neg(c_0 = \text{“e”} \wedge c_1 = \text{“e”})$) just by looking at the states of the diagram. Similarly, weak fairness of the next-state relation Next ensures that process 0 will eat infinitely often ($\Box\Diamond(c_0 = \text{“e”})$), since no trace through the diagram can forever avoid visiting the lower left-hand node. However, the symmetric property for process 1 is not so obvious because of the loop between the left-hand nodes of the diagram. To break the loop, we decorate some edges with ordering annotations. In fact, Thk_0, when performed from a state satisfying the predicates of the lower left-hand node, is guaranteed to decrease n, and Eat_0 does not increase n as it leaves n unchanged. As $<$ is a well-founded ordering over the natural numbers, one cannot have an infinite descent of n, reflected by performing model checking under the extra temporal logic hypothesis

$$\Box\Diamond\langle n' < n \rangle_v \Rightarrow \Box\Diamond\langle \neg(n' \leq n) \rangle_v .$$

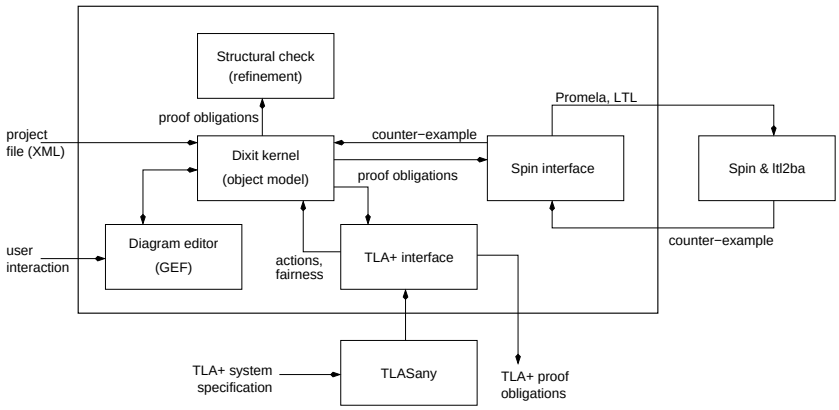


Fig. 2. DIXIT tool architecture.

2 Functionality and Architecture of DIXIT

DIXIT is intended to assist a user in performing the kind of reasoning illustrated in Sect. 1. Besides the verification of temporal logic properties, it also supports the stepwise development of a system by refinement; we illustrate the use of DIXIT more completely in the appendix at the hands of the alternating bit protocol; another case study appears in [7].

The functionalities of DIXIT are the following: a user can draw a predicate diagram, either by starting from scratch or by refining an existing diagram. Two slightly different instances of the diagram editor are used, as no new nodes may be introduced in refinements, but existing nodes may be split. Nodes are characterized by their names and by the set of literals they contain; edges are annotated by action names (associated with fairness conditions) and by ordering constraints.

Temporal logic properties (expressed over the set of literals that appear in the node labels) can be verified by model checking. DIXIT generates a model for the SPIN model checker and has it verify an appropriate LTL formula. If SPIN reports a counterexample, it is visualized in the graphical editor.

DIXIT can generate proof obligations that ensure that a diagram conforms to (is a correct abstraction of) a TLA$^+$ system specification associated with the model. These obligations are written to a TLA$^+$ module; they can currently not be proven within DIXIT.

For a diagram that has been created as a refinement, the correctness of the refinement can be verified, and three kinds of proof obligations are generated: first, the diagrams are structurally compared (low-level initial nodes are derived from high-level initial nodes, and transitions and ordering annotations match similarly). Second, the labels of refined nodes should imply those of the corresponding abstract nodes (again, verification conditions are generated, but not checked). Third, the fairness conditions of the abstract diagram can be implemented by a mix of low-level fairness constraints and ordering annotations, again using SPIN.

Figure 2 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 a graphical editor, derived from the GEF framework [1]. 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 different modeling languages. Similarly, DIXIT currently uses the SPIN model checker [5], but adapters for different model checkers could be easily substituted. Finally, an interface to an external theorem proving component is not yet instantiated. 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$^+$ models. Diagrams can be exported in Postscript, GIF, and SVG formats.

3 Conclusion and Future Work

We have outlined the design and the use of the DIXIT toolkit for the verification and refinement of reactive and distributed systems based on predicate abstractions. We have found DIXIT to be very helpful in teaching, as it allows to clearly focus on the abstractions underlying a given protocol. In particular, fairness conditions, which are notoriously difficult to get right, can be found very conveniently by considering the SCCs of the diagram, a process automated by the model checking component.

For more extensive use on large systems, more automation will be useful. In a first step, we plan to discharge many of the proof obligations using automated provers, letting the user focus on the difficult ones interactively. Preliminary experience with the B prover [6] has been encouraging. We also intend to assist in the generation of diagrams using techniques of abstract interpretation that have already been employed very successfully in the context of predicate abstraction [2]. Most importantly, we would be interested in feedback on the use of the tool by others; DIXIT is freely available at `http://www.loria.fr/equipes/mosel/dixit/`.

References

1. Graphical editing framework. `http://gef.tigris.org/`.
2. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages (POPL 2002)*, pages 1–3, 2002.
3. Dominique Cansell, Dominique Méry, and Stephan Merz. Diagram refinements for the design of reactive systems. *Journal of Universal Computer Science*, 7(2):159–174, 2001.
4. 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.
5. Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
6. STERIA Méditerrannée. *Atelier B, Manuel de Référence du Langage B*. GEC Alsthom Transport and STERIA and SNCF and INFRETS and RATP, 1997.
7. Stephan Merz. TLA+ case study: A resource allocator. Rapport de recherche A04-R-101, LORIA, Nancy, France, 2004.

A Example: Alternating Bit Protocol

We illustrate the use of DIXIT by reconstructing the well-known alternating bit protocol in a series of refinements.

A.1 First abstraction: data transmission

The purpose of the protocol is to transmit a sequence of data from a sender to a receiver process, so in a first abstraction we represent the state by two variables $sent$ and $rcvd$ that contain the history of values sent and received. The following is a first predicate diagram for this model:

The left-hand node 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. In such a state, the sender may send a new value, resulting in a state represented by the right-hand node and satisfying $rcvd = front\_sent$. (DIXIT allows either atomic predicates P, equalities $id_1 = id_2$ or formulas $id_1 \in id_2$ where id_1 and id_2 are identifiers, or negations of such predicates.) With this diagram we associate the following TLA$^+$ module $DataTransmission$ that fixes the interpretation of the terms and predicates of the predicate diagram.

MODULE $DataTransmission$

EXTENDS $Naturals, Sequences$

VARIABLES $sent, rcvd$

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

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

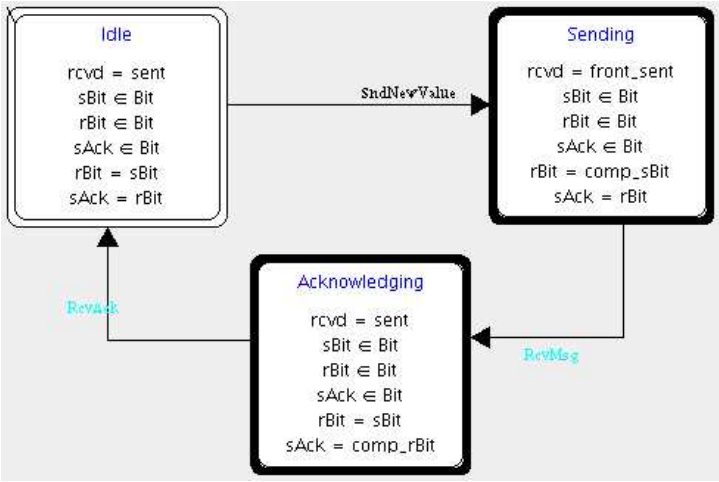
$front\_sent \triangleq Front(sent)$

The module is based on the TLA$^+$ library modules $Naturals$ and $Sequences$ that define natural numbers and finite sequences, with associated operators. It 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 yet define the actions $SndNewValue$ and $RcvMsg$, which are left abstract at this stage.

We require weak fairness for action $RcvMsg$ (indicated by the cyan color in the predicate diagram), and therefore every message sent must eventually be received. We can use SPIN to verify the formula $\Box\Diamond(rcvd = sent)$ over the diagram, and this property is guaranteed to be preserved by the subsequent refinement steps.

A.2 Synchronization of Bits

At the next level of abstraction, we introduce an explicit acknowledgement phase and add sender, receiver, and acknowledgement bits to the state representation. We do not yet explicitly represent message and acknowledgement channels, nor message loss.



The predicate diagram shown above was obtained as a refinement of the previous diagram by splitting the abstract node "Idle" to obtain nodes "Idle" and "Acknowledging" at the refinement level, 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 unchanged; consequently, $rBit$ will be the complement of $sBit$. A successful message reception restores the equality of $rBit$ and $sBit$, causing $sAck$ to be different from $rBit$. A new action $RcvAck$, again with a weak fairness condition, restores the equality of all three bits and therefore leads back to state "Idle". For completeness, we again show the associated TLA$^+$ module.

MODULE $SyncBit$

EXTENDS $DataTransmission$

VARIABLES $sBit$, $rBit$, $sAck$

$Bit \triangleq \{0,1\}$

$complement(b) \triangleq 1 - b$

$comp\_sBit \triangleq complement(sBit)$

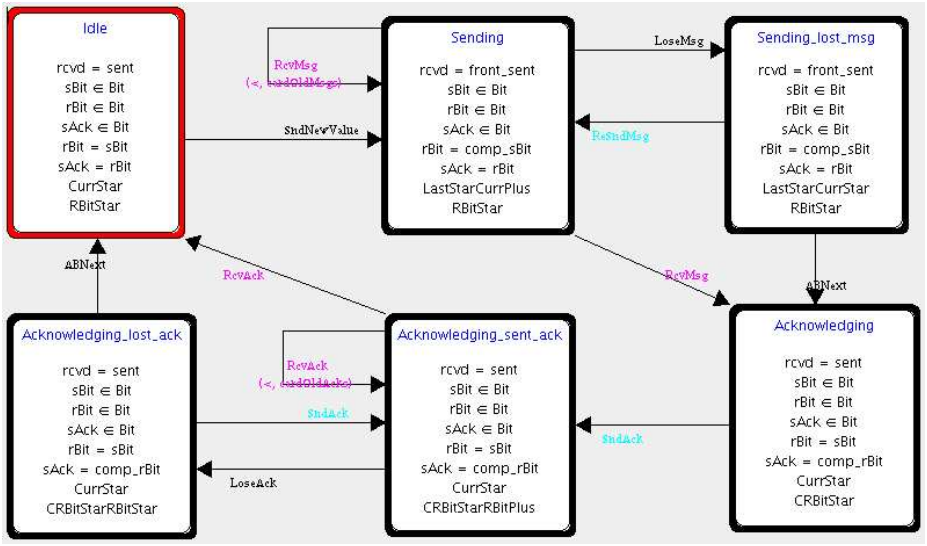
$comp\_rBit \triangleq complement(rBit)$

Using DIXIT, we can verify that the new diagram refines the previous one. First, a structural test confirms that every initial node of the diagram has been derived from an initial node of the higher-level diagram and that transitions of the lower level respect the transitions that existed before (in particular, the new $RcvAck$ transition refines the previously implicit stuttering transition at node "Idle"). 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 trivial because we have only added new predicates). Third, SPIN is invoked to ensure that the abstract-level fairness properties are preserved. Again, this is trivial for this refinement, but we will see a more interesting case at the next level.

A.3 Alternating Bit Protocol

Our final refinement introduces transmission channels and message loss; it corresponds to the final protocol and is intended to be verified as conforming to a TLA$^+$ system specification.



The above predicate diagram has been obtained by splitting the "Sending" and "Acknowledging" nodes and by adding predicates describing the form of the message and acknowledgement channels. For example, in state "Idle", there may still be some copies of the current message (with the current value of $sBit$) and of the acknowledgement on the channels, as expressed by the predicates $CurrStar$ and $RBitStar$ (these predicates are defined in the TLA$^+$ module $AlternatingBit$, reproduced in Figs. 3 and 4). 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 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 state "Sending", but one copy of the old messages has been consumed. Otherwise,

MODULE *AlternatingBit*

EXTENDS *SyncBit*

VARIABLES $msgQ$, $ackQ$

$$
\begin{array}{l}
Init \triangleq \land\ sent = \langle\rangle \land rcvd = sent \\
\qquad\quad \land\ sBit \in Bit \land rBit = sBit \land sAck = rBit \\
SndNewValue \triangleq \\
\quad \exists\, d \in Value : \land\ sAck = sBit \land sent' = Append(sent, d) \\
\qquad\qquad\qquad\ \land\ sBit' = comp\_sBit \land msgQ' = Append(msgQ, \langle d, sBit' \rangle) \\
\qquad\qquad\qquad\ \land \text{UNCHANGED } \langle rcvd, rBit, sAck, ackQ \rangle \\
ReSndMsg \triangleq \land\ sAck \neq sBit \land msgQ' = Append(msgQ, \langle Last(sent), sBit \rangle) \\
\qquad\qquad\ \land \text{UNCHANGED } \langle sent, rcvd, sBit, rBit, sAck, ackQ \rangle \\
RcvMsg \triangleq \land\ msgQ \neq \langle\rangle \\
\qquad\qquad\ \land\ msgQ' = Tail(msgQ) \land rBit' = Head(msgQ)[2] \\
\qquad\qquad\ \land\ rcvd' = \text{IF } rBit' \neq rBit \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } Append(rcvd, Head(msgQ)[1]) \text{ ELSE } rcvd \\
\qquad\qquad\ \land \text{UNCHANGED } \langle sent, sBit, sAck, ackQ \rangle \\
SndAck \triangleq \land\ ackQ' = Append(ackQ, rBit) \\
\qquad\qquad\ \land \text{UNCHANGED } \langle sent, rcvd, sBit, rBit, sAck, msgQ \rangle \\
RcvAck \triangleq \land\ ackQ \neq \langle\rangle \\
\qquad\qquad\ \land\ ackQ' = Tail(ackQ) \land sAck' = Head(ackQ) \\
\qquad\qquad\ \land \text{UNCHANGED } \langle sent, rcvd, sBit, rBit, msgQ \rangle \\
Lose(c) \triangleq \land\ c \neq \langle\rangle \\
\qquad\qquad\ \land \text{LET } l \triangleq Len(c) \\
\qquad\qquad\qquad \text{IN } \exists\, i \in 1..l : c' = [j \in 1..l-1 \mapsto \text{IF } j < i \text{ THEN } c[j] \text{ ELSE } c[j+1]] \\
LoseMsg \triangleq \land\ Lose(msgQ) \\
\qquad\qquad\ \land \text{UNCHANGED } \langle sent, rcvd, sBit, rBit, sAck, ackQ \rangle \\
LoseAck \triangleq \land\ Lose(ackQ) \\
\qquad\qquad\ \land \text{UNCHANGED } \langle sent, rcvd, sBit, rBit, sAck, msgQ \rangle \\
ABNext \triangleq \lor\ SndNewValue \lor ReSndMsg \lor RcvAck \\
\qquad\qquad\ \lor\ RcvMsg \lor SndAck \\
\qquad\qquad\ \lor\ LoseMsg \lor LoseAck \\
vars \triangleq \langle sent, rcvd, sBit, rBit, sAck, msgQ, ackQ \rangle \\
ABLive \triangleq \text{WF}_{vars}(ReSndMsg) \land \text{SF}_{vars}(RcvMsg) \land \text{WF}_{vars}(SndAck) \land \text{SF}_{vars}(RcvAck) \\
ABSpec \triangleq Init \land \Box[ABNext]_{vars} \land ABLive
\end{array}
$$

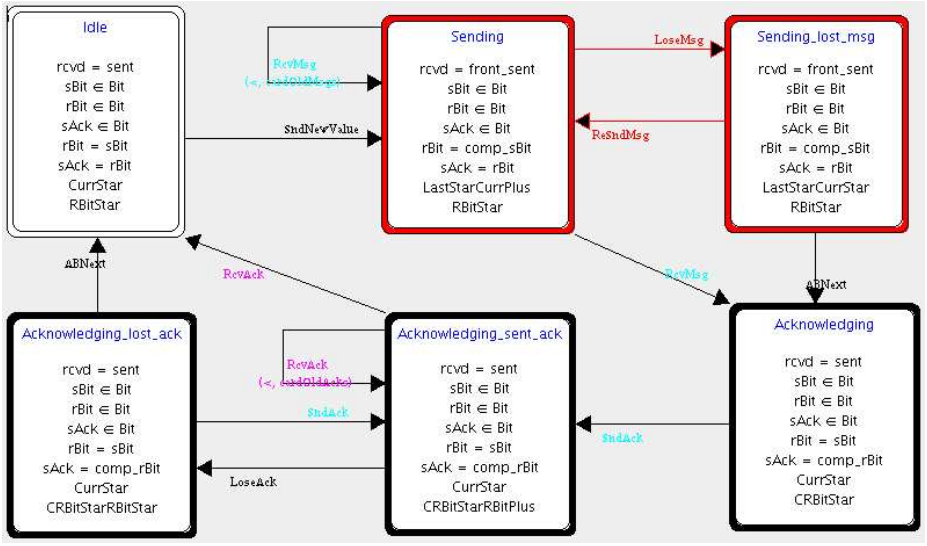
Fig. 3. Module AlternatingBit (part 1: system specification).

$Star(c, e) \triangleq \forall i \in 1..Len(c) : c[i] = e$
$Plus(c, e) \triangleq c \neq \langle\rangle \wedge Star(c, e)$
$StarStar(c, e1, e2) \triangleq \exists c1, c2 : c = c1 \circ c2 \wedge Star(c1, e1) \wedge Star(c2, e2)$
$StarPlus(c, e1, e2) \triangleq \exists c1, c2 : c = c1 \circ c2 \wedge Star(c1, e1) \wedge Plus(c2, e2)$
$CurrStar \triangleq \vee\ sent = \langle\rangle \wedge msgQ = \langle\rangle$
$\qquad\qquad \vee\ sent \neq \langle\rangle \wedge Star(msgQ, \langle Last(sent), sBit\rangle)$
$LastStarCurrStar \triangleq$
$\vee\ Len(sent) \leq 1 \wedge CurrStar$
$\vee\ Len(sent) > 1 \wedge StarStar(msgQ, \langle sent[Len(sent) - 1], comp\_sBit\rangle, \langle Last(sent), sBit\rangle)$
$LastStarCurrPlus \triangleq$
$\vee\ Len(sent) = 1 \wedge Plus(msgQ, \langle Last(sent), sBit\rangle)$
$\vee\ Len(sent) > 1 \wedge StarPlus(msgQ, \langle sent[Len(sent) - 1], comp\_sBit\rangle, \langle Last(sent), sBit\rangle)$
$RBitStar \triangleq Star(ackQ, rBit)$
$CRBitStar \triangleq Star(ackQ, comp_r Bit)$
$CRBitStarRBitStar \triangleq StarStar(ackQ, comp\_rBit, rBit)$
$CRBitStarRBitPlus \triangleq StarPlus(ackQ, comp\_rBit, rBit)$
$cardOldMsgs \triangleq Cardinality(\{i \in \text{DOMAIN } msgQ : msgQ[i] \neq Last(sent)\})$
$cardOldAcks \triangleq Cardinality(\{i \in \text{DOMAIN } ackQ : ackQ[i] \neq rBit\})$

Fig. 4. Module AlternatingBit (part 2: auxiliary predicates).

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. In particular, the only copy of the current message on the channel may be lost, causing a move to node "Sending_lost_msg". Observe that in that state the $RcvMsg$ action is not necessarily enabled, as the message queue may be empty. Therefore, the transition to state "Acknowledging" is only marked as a possible transition (labeled with the next-state relation $ABNext$), but no fairness is asserted. However, the sender will eventually resend the current message, causing a move back to node "Sending". The reasoning for the remaining transitions is similar.

To verify that the new predicate diagram is a correct refinement of the previous one, one again has to perform the three verifications described in Sect. A.2. 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 the old "Sending" node, and one therefore has to assert strong fairness, corresponding to the assumption that the channel does not lose all messages. Moreover, receiving a message does not ensure immediate progress from "Sending" to "Acknowledging", and so an ordering annotation is necessary for the loop at node "Sending". DIXIT helps to find these annotations. For example, when only weak fairness is asserted of action $RcvMsg$, SPIN produces a counter-example when trying to verify refinement, and this counter-example is visualized in the diagram as shown in the following screenshot, with the nodes and the edges highlighted in red.



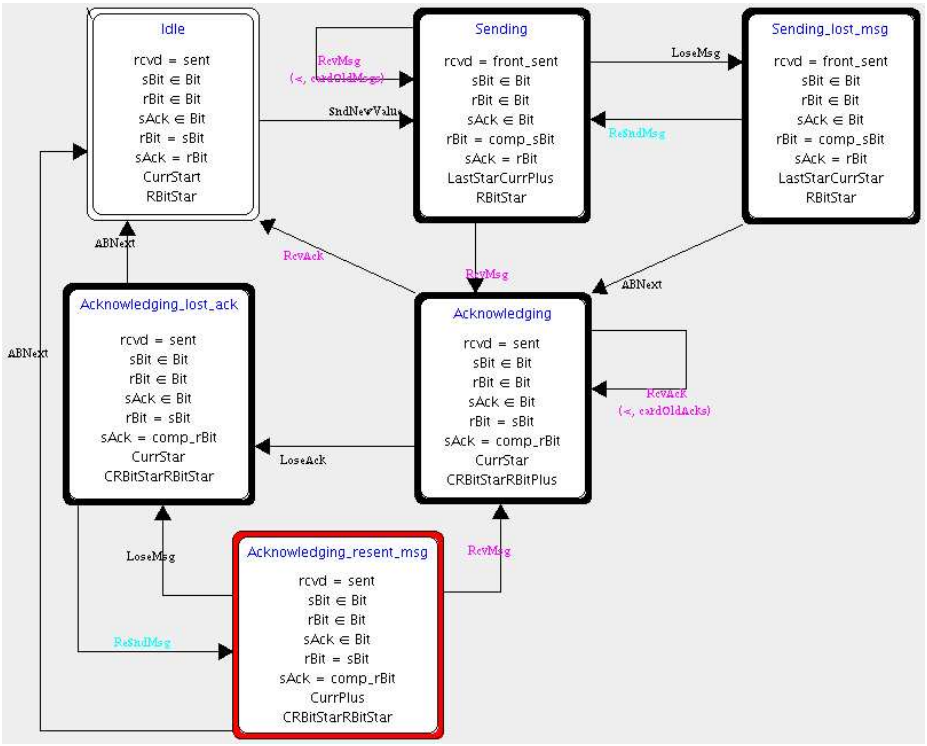
Finally, one can generate proof obligations to prove that the predicate diagram is a correct abstraction of the TLA$^+$ module $AlternatingBit$, as defined in [3]. However, the currently distributed version lacks an external prover to discharge these proof obligations.

A.4 Alternative Model of Alternating Bit Protocol

In the previous model of the alternating bit protocol, the receiver was symmetric to the sender in that it was required to (re)send acknowledgement messages. An alternative model has the receiver send acknowledgements only upon reception of (new or duplicate) messages. A corresponding predicate diagram appears in the screenshot on the following page. It differs from the previous one by the reaction to loss of acknowledgements: instead of relying on weak fairness of the $SndAck$ action modeling sending of acknowledgements by the receiver, one must rely on the *sender* to resend the current message, upon which the system moves to node "Acknowledging_resent_msg". Although that message may again be lost, strong fairness of $RcvMsg$ ensures that some message will eventually be received and a corresponding acknowledgement be sent. Formally, the diagram is again a refinement of the $SyncBit$ model discussed in Sect. A.2, although it is incomparable to the previous model of the alternating bit protocol.

Figure 5 shows a screen shot of the main interaction window of DIXIT. The elements of every diagram are shown in a tree structure, as are associated TLA$^+$ modules and properties, with their current status. In this picture, the structural refinement verifications of the model "Synchronization of Bits" presented in Sect. A.2 by the second model of the alternating bit protocol have been performed successfully, the proof obligations for the predicates have not been discharged, and the refinement of weak fairness

appears in dark green, indicating that it had been verifi ed previously, but that it may have been invalidated by some modifi cation of the diagram.



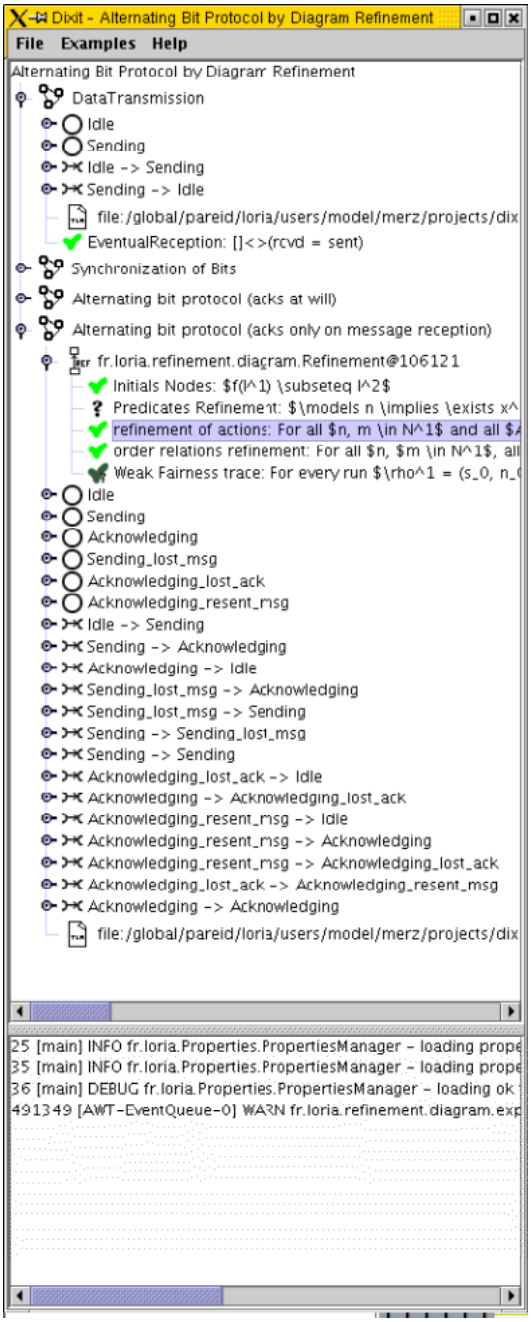


Fig. 5. DIXIT control center.