Model checking of component connectors

Izadi, M.

7 A Reo Model Checker

In this chapter, we introduce the main theoretical and practical concepts we used to implement a binary decision diagrams (BDD) based model checking tool for Reo specifications. This implementation is based on the augmented Büchi automata of records semantic model introduced in the previous chapters. Moreover, this tool accepts properties expressed in the ρLTL linear temporal logic as input and verifies a Reo specification against these properties. The Reo language has a wide range of applicability in coordination modeling and many real world case studies need a large number of channels to model the sophisticated orchestration patterns among constituent components. To address complex coordination patterns in large Reo circuits our proposed solution is based on BDDs.

Applying BDDs and using a symbolic representation for the underlying state space, helps us to improve the performance in small and middle size cases and also expands the applicability of our tool to larger Reo circuits. In the remainder of this chapter we first introduce a method for encoding of an ABAR as BDDs and reformulating of ABAR join operation in BDD terms. Next, we propose a method for converting a ρLTL formula to its equivalent Büchi automata and also apply the previously described procedure to represent the automata with BDDs. Having the BDD representation of an underlying ABAR of a Reo circuit and the BDD representation of a ρLTL property, we explain our model checking procedure in the following section. Finally, we present some experimental results of our tool.

7.1 Binary Decision Diagrams

Binary decision diagrams are data structures used for compressed representation of switching functions or Boolean formulas [111, 50, 29]. They are often more compact than other ways of representing of Boolean formulas, such as conjunctive and disjunctive normal forms, and they can be manipulated more efficiently [50]. In this section, we briefly describe binary decision diagrams and review their preliminaries. Our presentation and notations in this section is based on the textbooks [29, 50].

Definition 7.1 Let $Var = \{x_1, \ldots, x_n\}$ be a set of Boolean variables and $Eval(Var)$ be the set of all evaluations for $x_1, \ldots, x_n$, that is, the set of all total functions of the type $Var \rightarrow \{0, 1\}$. A *switching function* for $Var = \{x_1, \ldots, x_n\}$ is a function $f : Eval(Var) \rightarrow \{0, 1\}$. The switching functions for the empty variable set ($Var = \emptyset$) are just constants 0 or 1.

Obviously, each Boolean formula over the Boolean propositions $x_1, \ldots, x_n$ represents a switching function and vise versa.

Definition 7.2 Let $f : Eval(z, y_1, \ldots, y_m) \longrightarrow \{0, 1\}$ be a switching function. The *positive cofactor* of f with respect to variable z is the switching function $f|_{z=1}$ of the type $Eval(y_1, \ldots, y_m) \longrightarrow \{0, 1\}$ in which for all m-tuple of bits $(b_1, \ldots, b_m)$,

$$f|_{z=1}(b_1, \ldots, b_m) = f(1, b_1, \ldots, b_m).$$

Similarly, the *negative cofactor* of f with respect to variable z is the switching function $f|_{z=0}$ of the type $Eval(y_1, \ldots, y_m) \longrightarrow \{0, 1\}$ in which for all m-tuple of bits $(b_1, \ldots, b_m)$,

$$f|_{z=0}(b_1, \ldots, b_m) = f(0, b_1, \ldots, b_m).$$

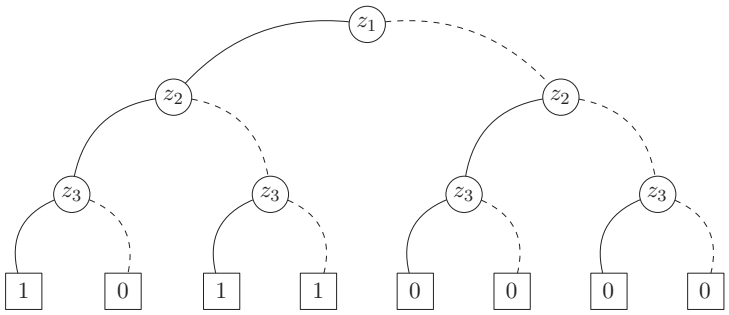

Figure 7.1: Binary decision tree for switching function $f = z_1 \wedge (\neg z_2 \vee z_3)$ [29].

The following result shows how a switching function f can be decomposed into its cofactors, called *Shannon expansion* [29]:

Lemma 7.1 If f is a switching function for the set of variables Var, then for each variable $z \in Var$,

$$f = (\neg z \wedge f|_{z=0}) \vee (z \wedge f|_{z=1}).$$

Using the Shannon expansion, one can represent switching functions by *binary decision trees*: Let f be a switching function for some variable set Var and fix an arbitrary enumeration $x_1, \ldots, x_n$ for the variables in Var. Now, we can represent f using a binary tree of height n such that the two outgoing edges of the inner nodes at level i stand for the cases $x_i = 0$ (depicted by a dashed line) and $x_i = 1$ (depicted by a solid line). Thus, the paths from the root to a leaf in that tree represent the evaluations and their corresponding values. The leaves (terminal nodes represented by boxes) stand for the function values 0 or 1 of f.

As an example, the binary decision tree for switching function $f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$ appears in Figure 7.1.

Binary decision trees are quite close to the representation of Boolean functions as truth tables as far as their sizes are concerned. However, they often contain some redundancy which we can exploit. Since 0 and 1 are the only terminal nodes of binary decision trees, we can optimize the representation by having pointers to just one copy of 0 and one copy of 1. Also, for more optimization we can remove unnecessary decision points in the tree, collapse constant subtrees (i.e., subtrees all whose terminal nodes have the same value) into a single node and merge isomorphic subtrees. This results in a directed acyclic graph (DAG) called a *binary decision diagram* (BDD).

As an example, the binary decision diagram obtained from the binary decision tree illustrated in Figure 7.1 is given in Figure 7.2.

A representation of switching functions is called a *canonical* representation if it satisfies the property that two switching functions are logically equivalent if and only if they have isomorphic representations [50]. This property simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not. Bryant [40] showed how to obtain a canonical representation for switching functions by placing two restrictions on binary

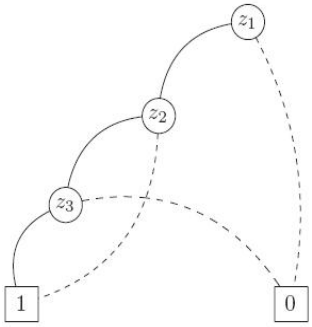

Figure 7.2: Binary decision diagram for switching function $f = z_1 \wedge (\neg z_2 \vee z_3)$ [29].

decision diagrams. First, the variables should appear in the same order along each path from the root to a terminal. Second, there should be no isomorphic subtrees or redundant vertices in the diagram. The first requirement is achieved by imposing a total ordering on the variables and the second is achieved by repeatedly applying three transformation rules that do not alter the function represented by the diagram [40, 50]:

- *Remove duplicate terminals*. Eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one.

- *Remove duplicate nonterminals*. If two nonterminal nodes u and v have the same label and their both subtrees are equal with each other, then eliminate u or v and redirect all incoming arcs to the other vertex.

- *Remove redundant tests*. If the left and right subtrees of a nonterminal node v are the same, then eliminate v and redirect all of its incoming arcs to the root of the subtree of v.

Starting with a binary decision diagram satisfying the ordering property, its canonical form is obtained by applying the above transformation rules until the size of the diagram can no longer be reduced. Bryant shows how this can be done in a bottom-up manner which is linear in the size of the original binary decision diagram [40]. The term *ordered binary decision diagram* (OBDD) will be used to refer to the graph obtained in this manner [50].

The size of an OBDD can depend critically on the selected variable ordering. In general, finding an optimal ordering for a set of variables is infeasible; in fact, it can be shown that even checking that a particular ordering is optimal is NP-complete. Moreover, there are Boolean functions that have exponential size OBDDs for any variable ordering. Several heuristics have been developed for finding a good variable ordering when such an ordering exists [50].

7.2 Encoding ABARs as BDDs

In this section we introduce a symbolic representation method for the ABAR model introduced in previous chapters. As our final aim is to implement a BDD based model checker, the state space generation should be transformed to BDD terms. Therefore, in our next step we propose an operation to mimic the join operation of ABAR models in BDD domain.

In the first step of our model checking procedure we represent the ABAR corresponding to each Reo channel in a symbolic way. Let $B = \langle Q, \Sigma, \longrightarrow, Q_0, F, l\rangle$ be an ABAR, where Σ is a set of records over a finite set of port names $\mathcal{N}$ and a finite data set $\mathcal{D}$. The ABAR B can be represented by a tuple $E_B = \langle \mathcal{N}_B, Q_B, \longrightarrow_B, Q_{B_0}, F_B\rangle$ of Boolean expressions which encodes port names, states and their labels, transition relation, initial states, and final states, respectively. In the following paragraphs, we present the encodings step by step.

- Let $V = \{n_1, \ldots, n_j\}$ be considered as a set of Boolean propositions corresponding to each port in $\mathcal{N}$. We represent $\mathcal{N}$ by the following Boolean expression:

$$\mathcal{N}_B = \bigvee_{n \in V} n.$$

- To symbolically represent the set of states Q, we use a set of Boolean state propositional variables $V_q = \{q_1, \ldots, q_k\}$, where $k = \lceil \log_2(|Q|)\rceil$, and assign a unique evaluation of $(q_1, \ldots, q_k) = (b_1, \ldots, b_k)$ to each state where $b_i \in \{0, 1\}$. Each state $q \in Q$ can be uniquely identified by the Boolean expression,

$$\phi_q = l(q) \wedge (\bigwedge_{1 \leq i \leq k} q_i)$$

  where, $l(q)$ is the propositional label of the state q. Also, the set of states Q is represented by the following Boolean expression:

$$Q_B = \bigvee_{q \in Q} \phi_q.$$

  Where we know that for all $q, q' \in Q$ it is the case that $l(q) \not\models l(q')$, we simply represent each state by its label ($\phi_q = l(q)$) and the set of all states Q by

$$Q_B = \bigvee_{q \in Q} l(q).$$

  Initial and final states are encoded in a similar way.

- Let $r = [n_1 = d_1, \ldots, n_i = d_i]$ be a record. It can be represented as the Boolean expression,

$$r_B = (\bigwedge_{n \in dom(r)} com_n) \bigwedge (\bigwedge_{n \in \mathcal{N} \setminus dom(r)} \neg com_n) \bigwedge (\bigwedge_{(n,d) \in r} \psi_{n,d})$$

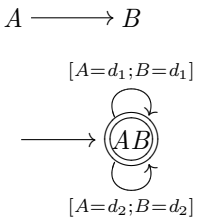

Figure 7.3: A synchronous channel and its ABAR model

where, for each $n \in \mathcal{N}$, com_n is a Boolean variable that intuitively says that port n is communicating and for each $(n, d) \in (\mathcal{N} \times \mathcal{D})$, $\psi_{n,d}$ is a Boolean variable that intuitively says that the data item in port n is d.

- Symbolic representation of a single transition involves the encoding of its source state, symbolic representation of its corresponding record label, and the label of its target state. To distinguish the source and the target states of a single transition, we use the above mentioned set of variables $V_q = \{q_1, \ldots, q_k\}$ and the set of ports $N = \{n_1, \ldots, n_j\}$ for the source state and a primed version of them, $V'_q = \{q'_1, \ldots, q'_k\}$ and $N' = \{n'_1, \ldots, n'_j\}$, for the target state. Therefore, the transition $T = q \xrightarrow{r} p$ is encoded by

$$T_B = \phi_q \wedge r_B \wedge \phi'_p,$$

where by ϕ'_p we mean the representation of the state p by ϕ_p (as we defined above) using the primed version of the state and port variables.

The transition relation is encoded by the disjunction of the symbolic encodings of all individual transitions.

Based on the above representation of each ABAR model using Boolean formulas (or switching functions), we can transform the Boolean formulas corresponding to each ABAR into BDDs.

Example 7.1 Figure 7.3 depicts a synchronous channel and its corresponding ABAR model. In this example, we assume that the data set is $\mathcal{D} = \{d_1, d_2\}$). Because there is only one state, we can ignore considering any state variable. Thus, we can represent the model by the following set of Boolean expressions:

$$\begin{gathered}
\mathcal{N}_{Sync} = A \vee B \\
Q_{Sync} = A \wedge B \\
Q_{0_{Sync}} = A \wedge B \\
F_{Sync} = A \wedge B \\
T_{Sync} = (A \wedge B \wedge com_A \wedge com_B \wedge (A = d_1) \wedge (B = d_1) \wedge A' \wedge B') \\
\vee \\
(A \wedge B \wedge com_A \wedge com_B \wedge (A = d_2) \wedge (B = d_2) \wedge A' \wedge B')
\end{gathered}$$

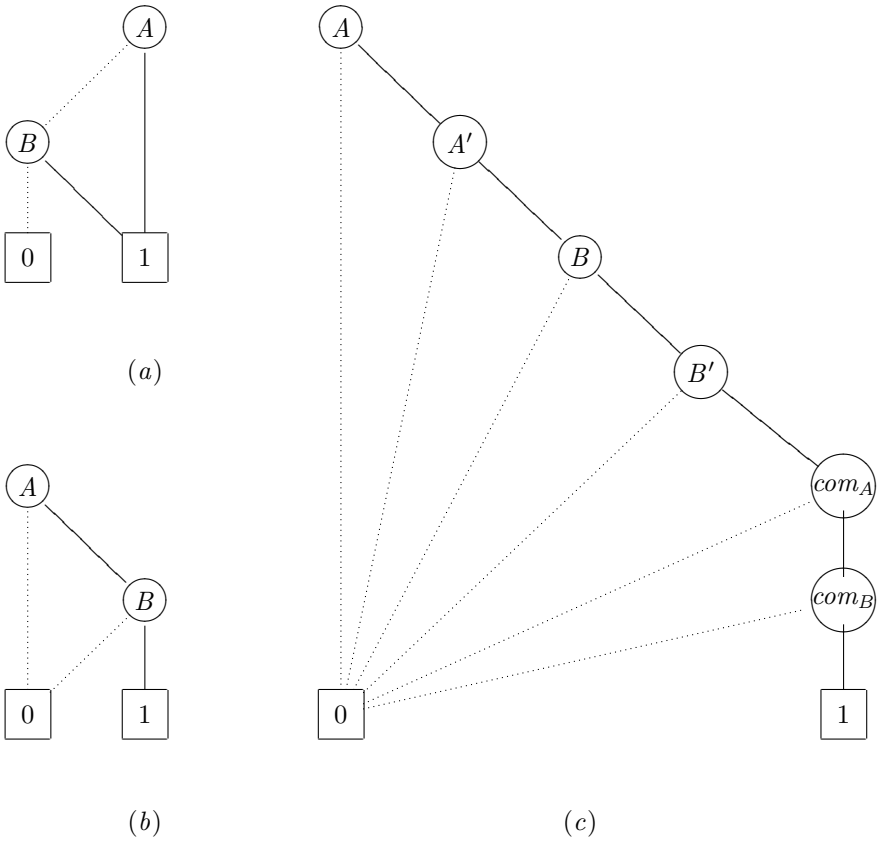

Figure 7.4: BDD representation of a synchronous channel: (a) ports, (b) states, initial states, final states and (c) transition relation.

If we assume that the data set is a singleton $D = \{d\}$, we can abstract away from the data set and the complexities introduced by its multiplicity. As a result the transition relation of the aforementioned synchronous channel can be expressed as follows:

$$T_{Sync} = A \wedge B \wedge com_A \wedge com_B \wedge A' \wedge B'$$

where com_x evaluates to true if and only if port x participates in the corresponding data communication. As the data set is assumed to be a singleton, the set of com_x variables is enough to represent the data communications. In other words, if a variable com_x evaluates to true it means that in its corresponding record we have $x = d$.

Based on the above representation of the ABAR model by Boolean formulas, now we can transform the Boolean formulas corresponding to each ABAR into BDDs. Figure 7.4 depicts a symbolic representation of the Boolean formulas representing the synchronous channel with BDDs. According to Figure 7.3 the single state of the ABAR representing a synchronous

channel is a final state and also an initial state. Therefore, the BDD in Figure 7.4.(a) is enough to store these three pieces of information.

Example 7.2 In a similar way we can symbolically represent a FIFO1 channel. Figure 7.5 depicts a FIFO1 channel, its ABAR interpretation, and a symbolic representation of this ABAR. Because none of the labels of the states implies the other, we use the labels of states as their Boolean representations. We assume that the data set is a singleton $D = \{d\}$. The BDDs in Figure 7.5 are equivalent to the following Boolean expressions:

$$\begin{gathered}\mathcal{N}_{FIFO} = B \vee C \\ Q_{FIFO} = B \vee C \\ T_{FIFO} = (B \wedge com_B \wedge \neg com_C \wedge C') \vee (C \wedge com_C \wedge \neg com_B \wedge B') \\ Q_{0_{FIFO}} = B \\ F_{FIFO} = B\end{gathered}$$

7.2.1 Symbolic Join

So we introduced a symbolic representation for the ABAR models of individual channels. Next, we represent a symbolic equivalent for the join operation of ABAR models. Informally speaking, the join of two ABAR models is computed following two different scenarios. Independent transitions of each ABAR can be interleaved. On the other hand, transitions of two ABARs whose record labels are compatible are synchronized.

Let B_1 and B_2 be two ABARs encoded by $E_{B_1} = \langle \mathcal{N}_1, Q_{B_1}, T_{B_1}, Q_{B_{01}}, F_{B_1} \rangle$ and $E_{B_2} = \langle \mathcal{N}_2, Q_{B_2}, T_{B_2}, Q_{B_{02}}, F_{B_2} \rangle$, respectively. The join of B_1 and B_2 is a generalized augmented Büchi automaton of records B which can be encoded by $E_B = \langle \mathcal{N}, Q_B, T_B, Q_{B_0}, F_B \rangle$ according to following expressions where $\neg com(\mathcal{N}_i)$ stands for $\bigwedge_{n \in \mathcal{N}_i} (\neg com_n)$.

$$\begin{gathered}\mathcal{N} = \mathcal{N}_1 \vee \mathcal{N}_2 \\ Q_B = Q_{B_1} \wedge Q_{B_2} \\ Q_{B_0} = Q_{B_{01}} \wedge Q_{B_{02}} \\ F_B = (F_{B_1} \wedge Q_{B_2}) \wedge (Q_{B_1} \wedge F_{B_2}) \\ T_B = Interleave_1 \vee Interleave_2 \vee Sync_{1,2} \\ Interleave_1 = \bigvee_{q \in Q_2} (\phi_q \wedge T_{B_1} \wedge \neg com(\mathcal{N}_2) \wedge \phi'_q) \\ Interleave_2 = \bigvee_{q \in Q_1} (\phi_q \wedge T_{B_2} \wedge \neg com(\mathcal{N}_1) \wedge \phi'_q) \\ Sync_{1,2} = T_{B_1} \wedge T_{B_2}\end{gathered}$$

Example 7.3 Consider the synchronous channel in Figure 7.3 and the FIFO1 channel in Figure 7.5(a). The join of these two channels is the connector in Figure 7.6(a) with its ABAR model depicted in Figure 7.6(b). The symbolic representation of this connector involves the BDDs in Figures 7.6(c)-(e).

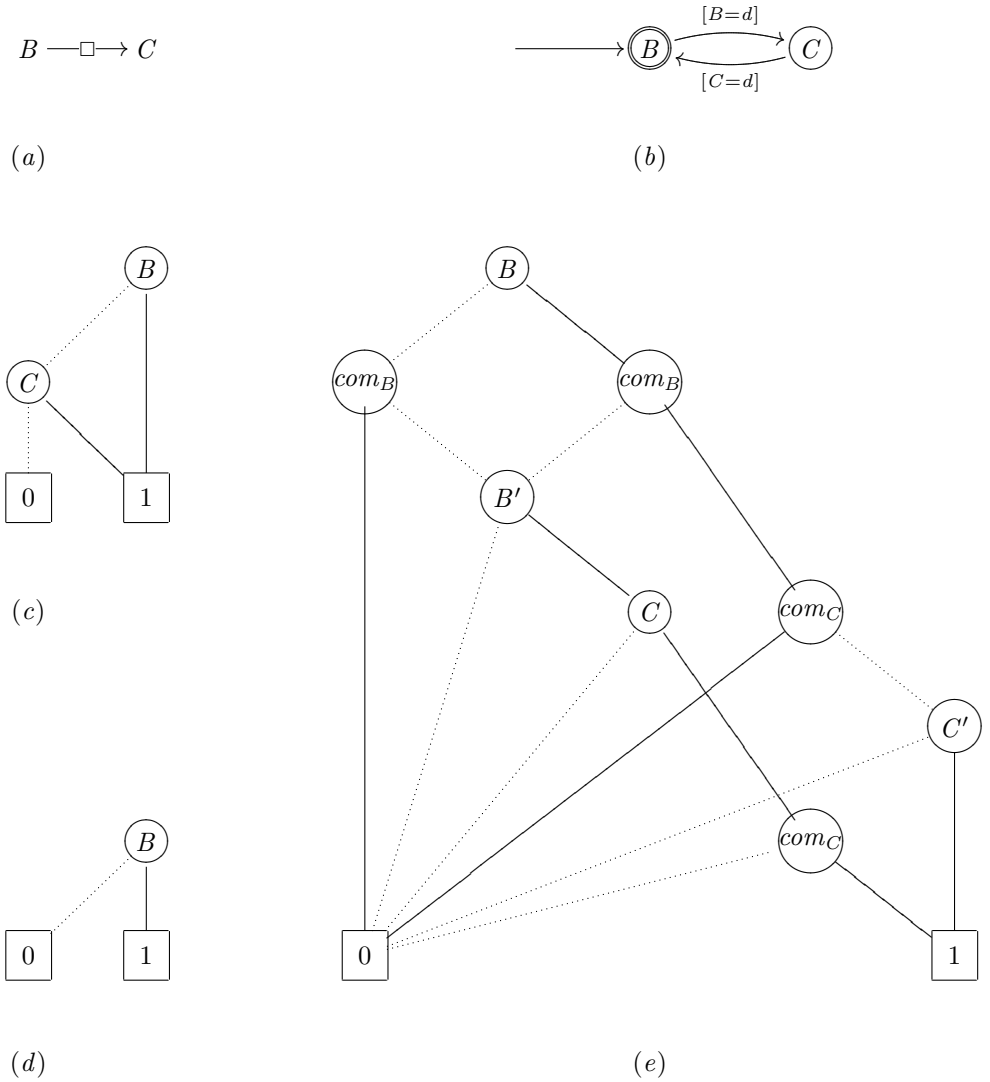

Figure 7.5: (a) FIFO1 channel, (b) its ABAR model, and BDD representation of (c) ports and states, (d) initial states and final states and (e) transition relation.

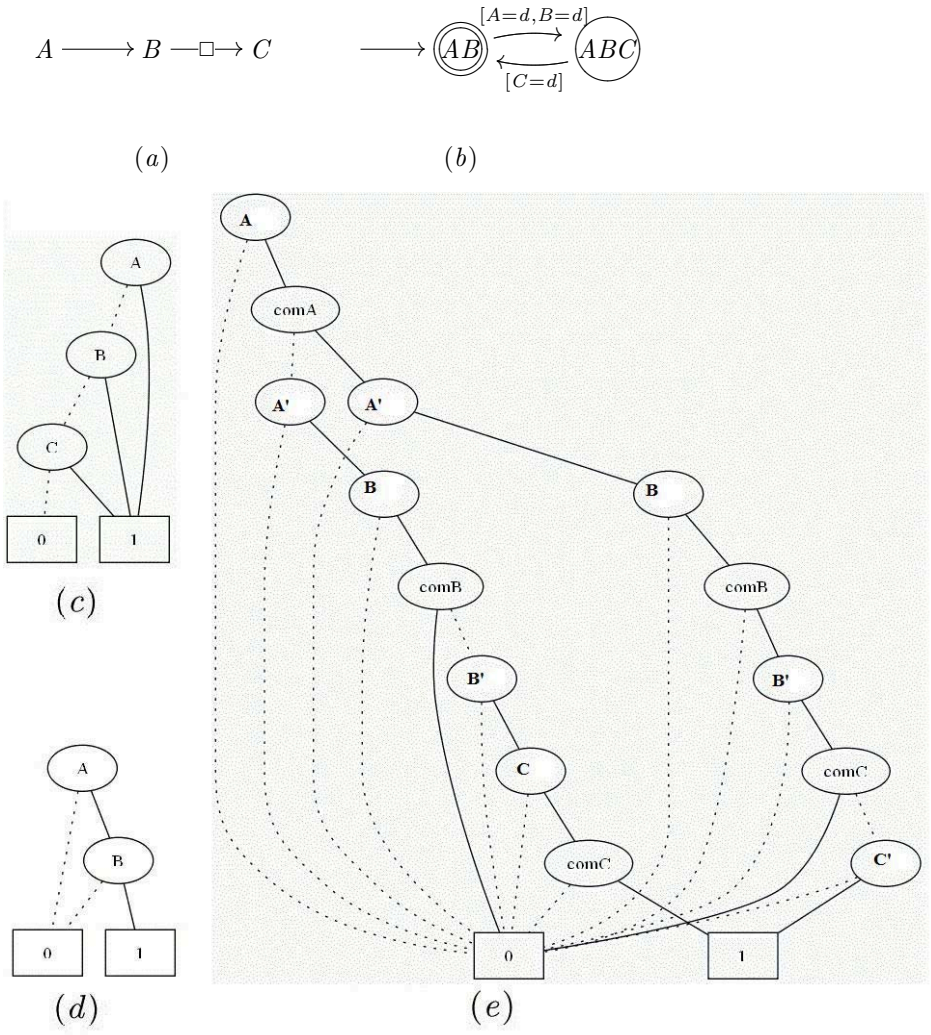

Figure 7.6: (a) Join of a synchronous channel and a FIFO1 channel, (b) its ABAR and BDD representation of (c) ports, (d) states, initial states, final states and (e) transition relation.

7.3 Property Specification by BDD

In the previous section we introduced a symbolic representation for the ABARs corresponding to Reo channels and an operation to mimic the join operation on ABARs in the BDD domain. In this section we continue our symbolic approach by introducing a BDD representation for ρLTL formulas. As mentioned earlier, ρLTL formulas are inductively generated by the following grammar:

$$\phi ::= N \mid \neg\phi \mid \phi \vee \phi \mid \langle r \rangle \phi \mid \phi U \phi.$$

where N is a subset of port names $\mathcal{N}$. Assuming a data set $\mathcal{D}$, r is a record from $Rec_{\mathcal{N}}(\mathcal{D})$.

Our ultimate goal is to implement our methods of global and on-the-fly translations of ρLTL formulas into augmented Büchi automata of records. However, as the first attempt, and in order to use the previously implemented tools (such as LTL2BA [58]) which translate LTL into Büchi automata, we transform ρLTL to LTL. For this purpose, we consider the set of port names $\mathcal{N}$ as a set of atomic propositions. Each $N \subseteq \mathcal{N}$ considered as a ρLTL formula represented by the following LTL formula:

$$\alpha_N = (\bigwedge_{n_i \in N} n_i) \wedge (\bigwedge_{n_i \in \mathcal{N} \setminus N} \neg n_i).$$

Assuming $r = [n_1 = d, n_2 = d, \ldots, n_m = d]$, the ρLTL formula $\langle r \rangle \phi$ is transformed into an LTL formula:

$$\langle r \rangle \phi = r_B \wedge \bigcirc \phi$$

where r_B was defined Section 7.2. As a result, for each ρLTL formula we have a semantically equivalent LTL formula.

Applying the procedure introduced in [58] and implemented as a tool (LTL2BA), an LTL formula is converted to a Büchi automaton whose transition labels are propositional expressions constructed from atomic propositions of the form n and com_n, where $n \in \mathcal{N}$. To unify our approach we transform the result to its equivalent ABAR. Formally, given a Büchi automaton $B = \langle Q, \Sigma, \longrightarrow, Q_0, F \rangle$ where Σ is the set of propositional expressions over the set of atoms $\mathcal{N} \cup \{com_n \mid n \in \mathcal{N}\}$, its equivalent ABAR is $\langle B', l \rangle$ such that the transition labels expressing a data communication constraint are resolved as their equivalent record representation, and the transition labels representing the port enabledness are resolved as the state label of their source state.

Example 7.4 Consider the ρLTL formula $\langle r \rangle (A \wedge B)$ where $r = [A = d, B = d]$. Assuming $\mathcal{D} = \{d\}$, the LTL equivalent of this formula is $com_A \wedge com_B \wedge \bigcirc(A \wedge B)$. A Büchi automaton for this LTL formula is depicted in Figure 7.7(a). This automaton is equivalent to the ABAR depicted in Figure 7.7(b) whit $r = [A = d, B = d]$. In this figure, by using Σ as a transition label, we mean that this transition is enabled for all records in $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$. Finally, the ABAR in Figure 7.7(b) can be symbolically represented as Figure 7.8. Variables $q1$ and $q2$ are used to encode states. For the case of a transition, $q1$ and $q2$ are used to encode the starting states and $q3$ and $q4$ to encode the target states (respectively, as primed version of $q1$).

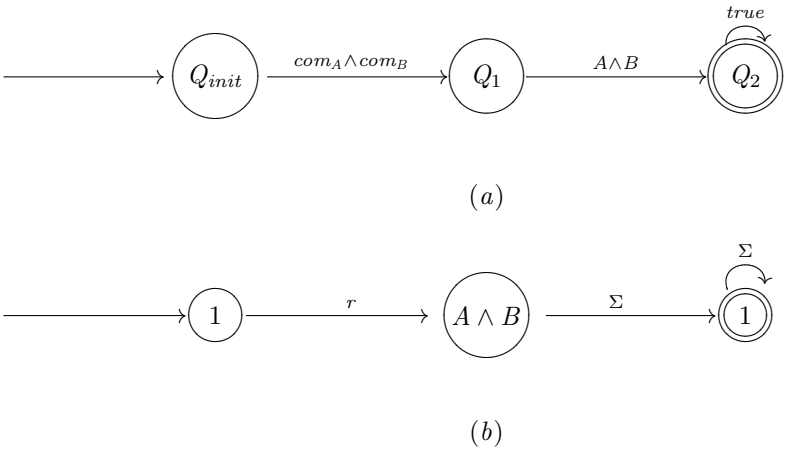

Figure 7.7: (a) A Büchi automaton and (b) an ABAR for $\langle r \rangle (A \wedge B)$

One of our planned future work to enhance our tool is to implement our own on-the-fly translation of ρLTL formulas directly into ABARs.

7.4 A symbolic model checking algorithm

We have introduced a symbolic representation for the ABARs representing Reo connectors in Section 7.2 and a BDD representation for ρLTL formulas in Section 7.3. Next we introduce the algorithm we implemented for model checking of Reo connectors. We follow an automata-based model checking approach. First we assume that the behavioral aspects of a system are modeled by an ABAR model. Second, a system property must be specified by a linear temporal logic. The negation of the property specified by a ρLTL formula is translated into an equivalent automaton. The join of the two automata representing the system and the ρLTL formula is computed. The final stage involves a procedure for language emptiness checking. Emptiness of the language of the resulting automaton implies that the system satisfies the property. Otherwise, the property is violated by the system and any word in the language of the resulting automaton is a counter-example.

Let $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \longrightarrow, Q_0, F, l \rangle$ be an ABAR model representing the behaviors exhibited by a Reo connector and $B_{\neg\phi} = \langle Q', Rec_{\mathcal{N}}(\mathcal{D}), \longrightarrow', Q'_0, F', l' \rangle$ be an ABAR translation of the negation of a ρLTL formula ϕ. The join of these two ABARs is a generalized ABAR. According to [29] the model checking problem for the formula ϕ is reduced to the classical problem of *fair cycle detection*. Stating the problem in our Büchi context, given the join ABAR $B \bowtie B_{\neg\phi} = \langle Q_{\bowtie}, Rec_{\mathcal{N}}(\mathcal{D}), \longrightarrow_{\bowtie}, Q_{0_{\bowtie}}, \mathcal{F}_{\bowtie} \rangle$ the system violates the property ϕ if and only if there exists a reachable cycle from one of the initial states $q_0 \in Q_0$ such that this cycle is fair with respect to the final states sets of the ABAR $B \bowtie B_{\neg\phi}$.

Not only the problem of LTL model checking, but also some other problems such as

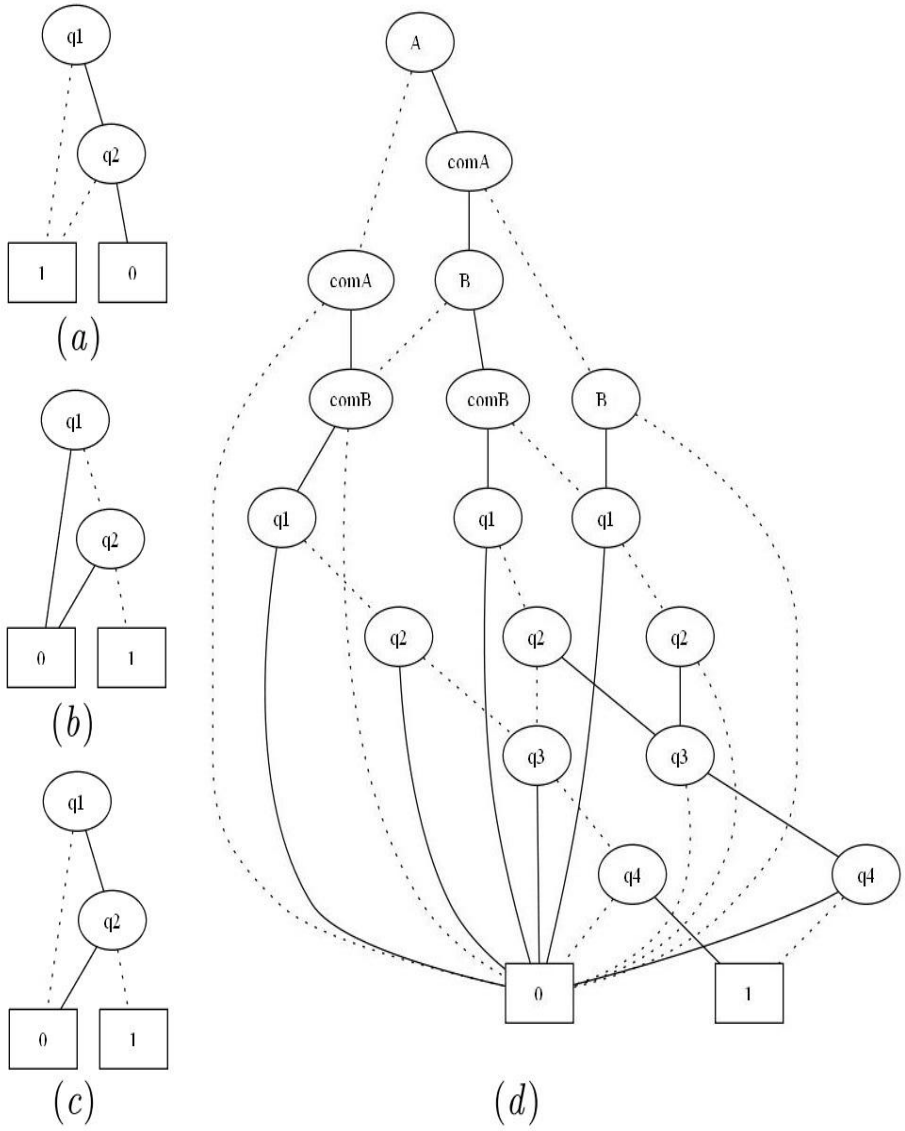

Figure 7.8: BDD representation for the ABAR equivalent of $\langle r \rangle (A \wedge B)$ (a) states, (b) initial states, (c) final states, and (d) transition relation.

language containment based on several types of automata, and CTL model checking with fairness constraints, all reduce to checking the emptiness of the language of a Büchi automaton. The language of the Büchi automaton is nonempty if and only if the automaton contains a *fair cycle*: a (reachable) cycle that contains at least one state from every accepting set, or, equivalently, a *fair strongly connected component* (SCC): a (reachable) nontrivial strongly connected component that intersects each accepting set.

The traditional approach to determine the existence of a fair SCC is to use Tarjans algorithm [137]. This algorithm is based on depth-first search and runs in linear time in the size of the graph. In order to do the depth-first search, the algorithm manipulates the states of the graph explicitly. Unfortunately, as the number of state variables grows, an algorithm that considers every state individually quickly becomes infeasible. Symbolic algorithms [111] manipulate sets of states via their characteristic functions. They derive their efficiency from the fact that in many cases of interest large sets can be described compactly by their characteristic functions. In contrast to explicit algorithms, an advantage of symbolic algorithms, which typically rely on breadth-first search, is that the difficulty of a search is not tightly related to the size of the state space, but is more closely related to the diameter of the graph and the size of the symbolic representation.

Several symbolic algorithms have been proposed that use breadth-first search and compute a set of states that contains all the fair SCCs, without enumerating them [126]. In this case, the typical and standard approach to fair cycle detection is the one of Emerson and Lei [57]. In the last decade, variants of this algorithm and an alternative method based on strongly connected component decomposition have been proposed. In [126], Ravi et al have presented a taxonomy of these techniques using some fix-point logic representations and compare representatives of each major class on a collection of real-life examples. Their main result indicates that the Emerson-Lei procedure is the fastest, but other algorithms tend to generate shorter counter-examples [126]. Based on this result, the algorithm implemented in our model checking tool is the one of Emerson and Lei [57].

7.5 Experimental results

In this section we present some experimental results applying our implementation based on the concepts introduced in the previous sections. Our model checking tool is implemented in C++ and compiled with GCC. The reported results are achieved on a Pentium 4, 2.53 GHz, 4GB RAM with an Ubuntu operating system. We use JINC [3] binary decision diagram library in our tool. In the following case studies, for the Reo channels we suppose that the data set is $\{d\}$ and we apply the ABAR models depicted in Figures 7.9.

7.5.1 Dining philosophers

The classical dining philosophers problem can be described as a coordination system by Reo specifications [14]. This system can be designed as a set of pairs of instances of two components: philosopher and chopstick instances. The externally observable behavior of a

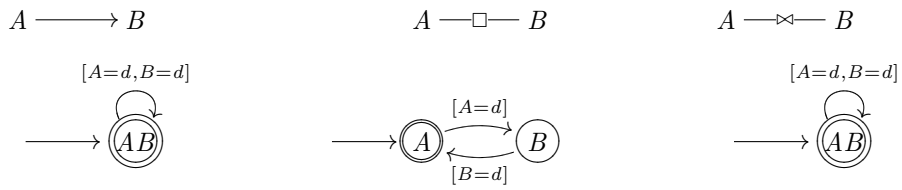

Figure 7.9: ABAR models of some Reo channels where $\mathcal{D} = \{d\}$.

Table 7.1: State space generation results for the dining philosophers problem

n	Time(sec)	Number of generated states	Number of BDD nodes for transition relation
2	0.236	36	245
3	0.533	216	473
4	3.810	1296	1043
5	92.515	7776	1956

philosopher component is as follows. After some period of thinking, it decides to eat, attempts to obtain its two chopsticks by issuing requests on its TL_i and TR_i ports. We assume that it always issues a request through its left chopstick before requesting the one on its right. Once both chopsticks take requests are granted it proceeds to eat for some time, at the end of which the philosopher then issues requests to free its left and right chopsticks by writing tokens on its RL_i and RR_i ports.

A chopstick is modeled by a FIFO1 channel and a synchronous drain [14]. The coordination pattern for this problem is represented in Figure 7.10 for the special case where the number of philosophers is 2. The coordination scenario and its graphical representation can simply be expanded for more than two philosophers [14]. Considering the philosophers as the active components in our system that communicate through this network, the behavioral aspects of such components can be modeled as Figure 7.11. In fact, q_1, q_2 and q_3 are respectively the $thinking$, $waiting$ and $eating$ states of each philosopher.

In Table 7.1, we see the number of states and the BDD nodes for different numbers of philosophers. For each case the state space generation time, number of generate states in the corresponding ABAR, and the number of BDD nodes in the symbolic representation of the transition relation (as it dominates other parts of our encoding considering the space complexity) are illustrated.

Let ϕ_1 and ϕ_2 be two ρLTL formulas in the context of dining philosophers problem such that:

$$\phi_1 = \Box\neg(eating_1 \wedge eating_2)$$
$$\phi_2 = \Box(TR_i \rightarrow \langle TR_i\rangle(RL_i \wedge RR_i)).$$

The property ϕ_1 asserts that in all instance of time, it is not the case that both philosophers are eating simultaneously. For the case of more than two philosophers, the inner conjunction in the formula ϕ_1 is expanded by all $eating_i$'s. The property ϕ_2 that is completely specified in terms of the port names, says that always if the philosopher i is waiting to take the right

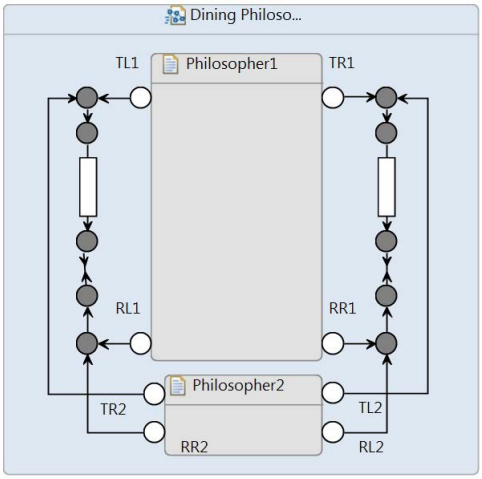

Figure 7.10: Coordination pattern for two philosophers in the dining philosophers problem

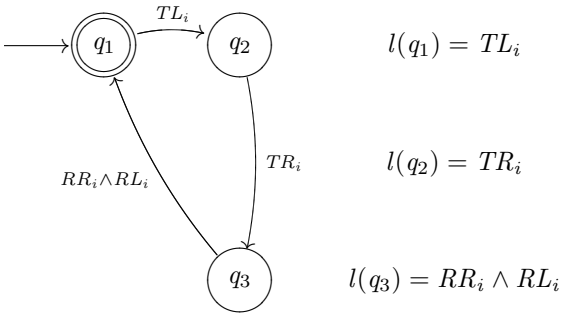

Figure 7.11: Behavior of a philosopher in ABAR terms

chopstick, it immediately takes it and goes to the eating state. obviously, the first property, ϕ_1, is satisfied by the presented coordination scenario but the second one, ϕ_2, is not satisfied since the readiness of a philosopher for taking its right chopstick does not lead to the $eating$ state for it exactly in the next state of the whole system.

Table 7.2 reports the model checking time for various number of philosophers for properties ϕ_1 and ϕ_2. For model checking the formula ϕ_1, the algorithm terminates after checking the whole state space, while for the case of ϕ_2, it terminates after finding a counterexample.

7.5.2 Mutual Exclusion

As another case study we consider a special variant of the mutual exclusion problem. Let n be the number of active processes in a system. Assuming $k \leq n$, in this variant of the mutual exclusion problem, at each time instance at most k processes can perform actions in

Table 7.2: Model checking time (sec) for n dining philosophers

n	ϕ_1	ϕ_2
2	0.260	0.258
3	0.611	0.603
4	4.246	4.272
5	96.098	96.009

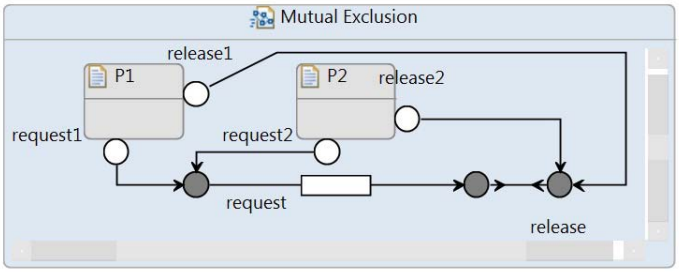

Figure 7.12: Coordination pattern for two processes in mutual exclusion for $k = 1$

the critical section. Figure 7.12 presents the communication pattern for this problem for a special case where $n = 2$ and $k = 1$.

The active processes in the system are considered as components that communicate through this connector. Figure 7.13 models the behavior of a process with an ABAR. Table 7.3 reports some results on state space generation in mutual exclusion problem for different numbers k of processes in the critical section, and active processes n, ($k \leq n$). For model checking purposes the labeling function of the ABAR depicted in Figure 7.13 must be extended to show the state of a process (executing critical/non-critical actions). Obviously, the state labeled by $release_i$ represents a configuration where a process executes actions in the critical sections. Let ϕ_3 and ϕ_4 be two ρLTL formulas such that:

$$\phi_3 = \Box\neg(critical_1 \wedge critical_2 \wedge critical_3)$$
$$\phi_4 = \Box(request_1 \rightarrow \langle request_1\rangle(\neg request_1))$$

Property ϕ_3 specifies that three processes cannot execute their critical actions simultaneously, a property which is true as long as $k<3$. Property ϕ_4 expresses that for a process ready to enter its critical section, always the next action is a request to execute its critical actions and this request is always accepted. This property is not always true.

Table 7.4 represents the model checking time for properties ϕ_3 and ϕ_4 for various values of n and k.

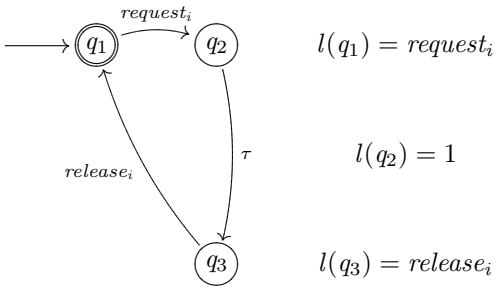

Figure 7.13: Behavior of a process in ABAR terms

Table 7.3: State space generation results for the mutual exclusion problem

n	k	Time(sec)	Number of generated states	Number of BDD nodes for transition relation
5	2	0.378	972	4421
6	2	0.868	2916	9830
7	2	1.134	8748	21767
8	2	3.105	26244	47912
9	2	30.318	236196	227690
10	3	60.090	472392	532209

7.5.3 Discussion

For model checking of the desired properties of coordination models specified by Reo, in addition to our above mentioned tool, there is another implemented tool called Vereofy [7]. The two above introduced case studies have also been considered by the authors of Vereofy and their results have been reported in [98, 99]. Similar to our implementation, they use OBDD as their main data structure to store and process their models. However, there are some essential differences between our approach to model checking of Reo nets and the work of Baier et al reported in [98, 99]:

- The modeling formalism that they use is constraint automaton while our models are

Table 7.4: Model checking time (sec) for the mutual exclusion problem

n	k	ϕ_3	ϕ_4
5	2	0.539	0.532
6	2	1.270	1.252
7	2	3.283	3.425
8	2	10.460	10.887
10	2	105.115	107.285

Büchi automata of records and their augmented versions.

- The property specification language that they use is an extension of the *branching time* temporal logic CTL called *BTSL* while our proposed logic is an action based *linear time* temporal logic called ρLTL.
- The algorithm of model checking used in the work of Baier et al is an extension of the symbolic CTL model checking algorithm [48] which by an iterative approach that computes the set of states satisfying each subformula of the desired property. Our model checking algorithm is based on the checking of the emptiness of the accepted language of an automaton that is reduced to the problem of detecting fair cycles in the graphs of automata.