COMPACTING A RULE BASE INTO AN AND/OR DIAGRAM FOR A GAME AI

Samuel Manier[1, 2] and Olivier Sigaud[1]

(1) Institut des Systèmes Intelligents et de Robotique
CNRS FRE 2507 Université Pierre et Marie Curie – Paris 6
4, place Jussieu, 75005 Paris, France

(2) Cyanide Studio
65, boulevard des bouvets,
92000 Nanterre, France

Samuel.Manier@gmail.com, Olivier.Sigaud@isir.fr

KEYWORDS
Game AI, Rule base, Decision Diagram, And/Or Diagram, Behavior Tree, Cycling simulation.

ABSTRACT

A game Artificial Intelligence defined by if-then rules can become hard to maintain and expand. In this paper we present a method to transform in a compact way a rule base into an And/Or Diagram, a hierarchical structure easier to manage over time. Our method is illustrated in a commercial cycling simulation video game.

INTRODUCTION

In game Artificial Intelligence (AI), simple if-then rules are often preferred over more powerful architectures such as Hierarchical Finite State Machines (HFSMs) (Harel 1987) or Behavior Trees (BTs) (Champandard 2008), because they do not require developing any complex engine or editor. Thanks to their simplicity, developers can quickly compose simple behaviors for game agents. However, their poor maintainability and scalability make them inappropriate when the complexity of these behaviors exceeds a certain degree. But the growth in complexity is not always easy to anticipate, especially for game series, such as sport games, which evolve in sequels in a more and more complex manner. In such a context, developers are often constrained to update their AI by rewriting it from scratch in a better formalism when it becomes too complex, depriving them of the possibility of re-using the knowledge contained in the original AI.

In this paper, based on our experience with a commercial sport video game, we propose a method to transform and compact semi-automatically a rule base in what we call an And/Or Diagram (AOD), whose formalism is close to the one of a BT. This method is divided into successive steps of transformation from a formalism to another:

- the semi-automatic transformation of the rule base into a decision diagram base;
- the automatic conversion of the decision diagram base into an AOD;
- the automatic refining (factorization and simplification) of the AOD;

In the following sections, we will detail those steps, present an application of our method to the Pro Cycling Manager game developed by Cyanide and discuss it, before concluding and giving future work directions.

TRANSFORMING A RULE BASE INTO AN AOD

Our starting point is an action selection system defined by a rule base where each rule has a premise, a conclusion and a priority (see Figure 1 for a small example). A premise is a conjunction of conditions on variables, a conclusion is a conjunction of discrete actions, whereas priorities define the order of rules testing, forbidding the activation of multiple rules at the same time.

r_1: if V1 = true and V2 < 50 and V3 < 10 then do Action1
r_2: if V1 = true and V2 < 50 and V3 > 20 then do Action2
r_3: if V1 = true and V4 = true then do Action3

Figure 1: Three if-then rules forming a rule base, ordered by decreasing priority. V1, …, V4 are variables.

The successive steps to transform and compact a rule base into an AOD are presented in the following subsections.

From A Rule Base To A Decision Diagram Base

The rule base we described is a flat structure that cannot handle common properties of rules without calling upon redundancy (for example, in Figure 1, the condition *V1 = true* is repeated in each rule). This can slow down considerably the understanding, the modification and the evolution of the AI. In this section, we explain how we reduce this redundancy by representing the rule base into a decision diagram base.

A decision diagram (Akers 1978) is a rooted Directed Acyclic Graph (DAG) composed of:

- <u>decision nodes</u>. A decision node represents a test on the value of a variable;
- <u>edges</u>. An edge coming from a decision node represents a condition on its variable;
- <u>leaves</u>. A leaf represents a solution to the decision diagram problem (for an action selection problem, a leaf is an action or a set of actions).

A decision diagram is executed from the root. At each decision node, the next visited node is designated by the edge whose condition is satisfied, until a leaf is reached.

We propose an algorithm to transform a rule base into what we call an Ordered Multi-Terminal Algebraic Decision Diagram (OMTADD) (designation deriving from Ordered Binary Decision Diagrams (OBDDs), Multi-Terminal Binary Decision Diagrams (MTBDDs) [Drechsler and Sieling, 2001] and Algebraic Decision Diagram (ADD) [Bahar et al. 1993]). An OMTADD is:

- <u>Algebraic</u>, because we use non binary variables (they can be Boolean, integer, float or defined by enumeration);
- <u>Multi-Terminal</u>, because we use non binary answers (in the leaves);
- <u>Ordered</u>, because our algorithm needs an order on variables.

In the rest of this paper, DD stands for OMTADD for short.

Our algorithm is a modified version of the DPLL procedure proposed by Huang and Darwiche (Huang & Darwiche 2004). The DPLL procedure is originally designed to transform a Conjunctive Normal Form (CNF), i.e. a conjunction of disjunctive conditions, into an OBDD. We adapted it to transform a rule base into a DD.

This algorithm requires an ordering on the variables. But, finding a good ordering for many variables is a hard problem. The resulting decision diagram can be huge if this ordering is not carefully chosen. We narrow down this problem by dividing the rule base into smaller rule bases, each one concerning a small enough subgroup of variables. It boils down to do the intuitive operation of regrouping rules using the same variables, or almost the same variables, together. However, this reorganization is constrained to respect rules priorities: the resulting rule bases are ordered by priority, each small rule base getting the priority of its first rule, and the following property is imposed: let r_1 and r_2 be two rules with compatible premises, r_1 belonging to the rule base R_1 and r_2 to the rule base R_2 (with $r_1 \neq r_2$ and $R_1 \neq R_2$). R_1.priority > R_2.priority implies that r_1.priority > r_2.priority.

The transformation of each resulting rule base into a DD is done separately by the *omtadd* algorithm presented below. We first give a brief summary and some definitions.

In this recursive algorithm, variables are instantiated successively with all their possible values. This makes premises of some rules becoming satisfied, creating decision nodes (corresponding to variables instantiations) and leaves (corresponding to rules conclusions). When several nodes are equivalent, only one is kept. The use of cutsets allows anticipating equivalence between nodes, avoiding computing them all.

Our definition of a cutset is the same as in (Huang & Darwiche 2004), with "clause" replaced by "rule": the $\underline{i^{th}}$-cutset of variable order $\pi = v_1,\ldots,v_n$ for the rule base $\{r_1,\ldots r_m\}$, denoted $\text{cutset}^i_R(\pi)$ or cutset_i for short, is defined as $\{r \in R : \exists j \leq i < k$ such that rule r mentions variables v_j and v_k in its premise$\}$.

A <u>cutset value</u> is a bit vector, each bit representing the state (satisfied or simplified) of the premise of a rule of this cutset. When two ore more configurations are found to have the same cutset value, only one of them will be computed, its DD cached, and others will simply generate a cache hit and have their DD immediately returned.

In a DD, the <u>fail action</u> is the one activated when all the rules represented by the DD fail. In that case, the next DD (with a lower priority) is evaluated.

Algorithm *omtadd*(Rules *R*, int *i*) : return the root node of the DD corresponding to the rule base *R*, whose rules are ordered by priority, according to a fixed order on variables. *i* is the current variable index.

```
if at least the premise of one rule is true then
    let r1 be the rule of highest priority among rules with a true premise;
    if there is no rule $r_2$ with simplified premise such that $r_2$.priority > $r_1$.priority and $r_2$.actions ≠ $r_1$.actions then
        return $r_1$.actions;
if all variables are instantiated then
    return the fail action;
if (node_already_computed = $cache_{i-1}[value(cutset^{i-1})]$) ≠ nul then
    return node_already_computed;
result = create_node (R, i);
$cache_{i-1}[value(cutset^{i-1})]$ = result;
return result;
```

Function *create_node*(Rules *R*, int *i*) : create a node, or return an already existing node, where the variable of index *i* is tested.

```
let N be an empty set of nodes;
for each possible value val of the variable $v_i$, do
    N.insert(omtadd($R|v_i$ = val, i+1));
if all the nodes in N are equals then
    return the first one;
else
    let Node[i] be the set of nodes representing the variable of index i;
    for each node n of Node [i], do
        if the children of n are exactly the nodes in N then
            return n;
    result = Node(i, N);
    Node[i].insert(result);
    return result;
```

Function *Node(i, N)* : create a node having *i* as variable index and the nodes contained in *N* as children.

The choice of the variable order in a BDD is hard [Tani 1993]. This is especially true for an MTADD, which is more complex than a BDD. Therefore, we do not automate that part: we proceed manually, by trial and error, to the choice of the variable order of each rule base, until we obtain a DD with a low enough degree of redundancy.

Figure 2 shows a DD resulting from the transformation of the rule base extracted from the rule base of Figure 1. We can see that the redundancy of conditions *V1 = true* and *V2 < 50* is avoided.

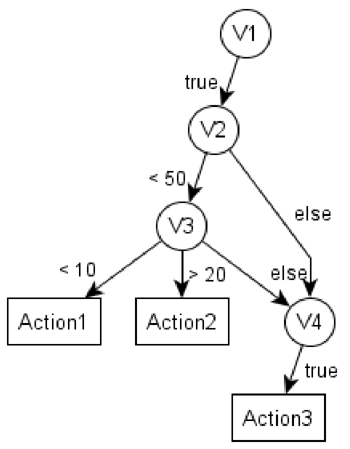

Figure 2: The DD resulting from the transformation of the rule base in Figure 1, according to the following variable order: V1, V2, V3, V4. The Fail action and edges that lead to it are not represented.

From A Decision Diagram Base To an AOD

DDs can represent disjunctions (see *else* edges in Figure 1 and, for another type of disjunctions, left side of Figure 3). But when these disjunctions involve numerous conditions, some nodes may have to be accessed from many edges. This contributes to slow down the editing of DDs (e.g. in Figure 3, replacing the node N by a node N' would require reconnecting to N' all edges leading to N). In this section, we show how we convert DDs into AODs. AODs represent disjunctions by OR nodes which, combined with AND nodes, make the AI modular and easier to reconfigure. The connection of all AODs into a single AOD is also explained.

An AOD is a rooted DAG where nodes can be either leaves (conditions, actions or a named link to other AODs) or composite nodes. There are different types of composite nodes:

- AND nodes. An AND node succeeds only if all its children succeed;
- OR nodes. An OR node succeeds if at least one of its children succeeds. Its children are ordered by priority;
- If-Then-Else nodes. An If-Then-Else node succeeds if both its If and Then children succeed or if its If child fails and its Else child succeeds. An equivalent of an If-Then-Else can be constructed with one OR node and two AND nodes.

The execution of an AOD starts from the root in a depth-first search. When a node fails, we backtrack to its parent and continue its execution (e.g. an OR node will execute its next child whereas an AND node will fail). Leaves representing actions always succeed.

We call successful sub-diagram the result of the execution of an AOD. It is composed of nodes that succeeded and that have a parent (except for the root) belonging to this sub-diagram. Actions contained in the successful sub-diagram form together the final decision.

The conversion of a DD into an AOD is done by searching recursively for patterns upon which we apply transformation rules in a priority order. These rules are explained in the following figures (from Figure 3 to **Figure 6**) presented in decreasing priority order. In these figures, nodes N, N1, N2 and N3 are not necessarily leaves, and nodes V and V1 are not necessarily roots.

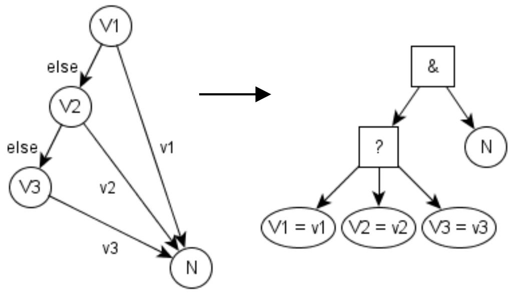

Figure 3: Conversion of a DD with disjunctions into an AOD.

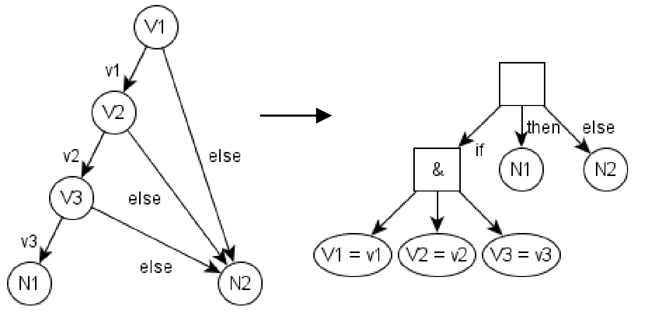

Figure 4: Conversion of a DD with another type of disjunctions into an AOD.

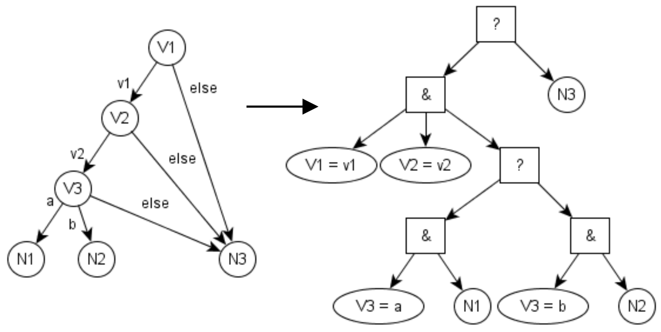

Figure 5: The general case, from the preceding rule, when the last decision node (here V3) has more than two children

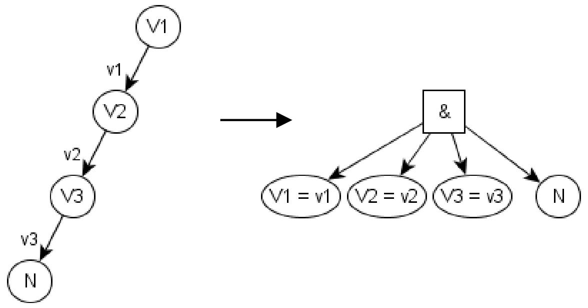

Figure 6: Conversion of a conjunction of conditions leading to the node N into an AND node responsible for the execution of these conditions and N.

AODs benefit from their hierarchical organization (see Figure 7 for an example): a sub-diagram can be seen as a high level condition (if all its leaves are conditions) or a high level action (if at least one of its leaves is an action). Thanks to this property, the complexity of an AOD (or a sub-AOD) can be hidden by representing only its root or a named link to its root. We use this property to link all generated AODs under a large OR node and thus regroup the whole action selection system into a unique AOD.

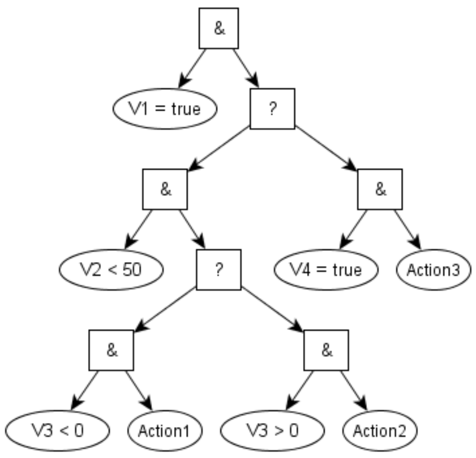

Figure 7: The AOD resulting from the transformation of the DD of the Figure 2 (equivalent to the rule base of Figure 1).

Refining Tools

We can further clarify an AOD and reduce its size with the following tools.

Factorization
From the definition of the successful sub-diagram and of the different composite nodes, we can infer that:

- when the successful sub-diagram contains an AND node, it necessarily contains all its children,
- when the successful sub-diagram contains an OR node, it contains only one of its children,
- when the successful sub-diagram contains an If-Then-Else node, it contains either its IF child and its THEN child or ELSE child.

Let N be a node with multiple parents. Let A be the common ancestor of parents of N. If all possible successful sub-diagrams coming from A contain N, then we can replace N by an always true condition and replace A by AND(A,N).

Simplification
After its construction and its factorization, an AOD may contain unnecessary nodes that can be eliminated (like true conditions created by factorization), provoking a recursive simplification of the AOD. As this procedure is intuitive, we will not detail it.

Creation of high level conditions and actions

Some too large AODs can remain hard to understand and manipulate. Their hierarchical property can be exploited to hide the complexity of some sub-diagrams into named links to those sub-diagrams. We propose a tool which first selects automatically sub-diagrams that have characteristics making them good candidates to be replaced by named links, and secondly let the user validate their replacement and choose the names of the links. The sub-diagrams proposed to the user are:

- sub-diagrams created from rules of Figure 6, Figure 4, and Figure 3, because they contain a high concentration of conditions that are susceptible, taken together, to reveal a high meaning;
- multi-parented sub-diagrams. As they are accessed from multiple contexts, they may represent relevant concepts.

APPLICATION TO A GAME

We applied the method described in this paper to the high level action selection system of the game Pro Cycling Manager. In this section, we first present the game and its original AI, then we show how we applied our method to it.

Pro Cycling Manager And Its AI

Pro Cycling Manager is a management game where the player supervises a professional cycling team through a career or a simple race in single or multi-player mode. Each race can be simulated or played in real time 3D, as shown on Figure 8. A race brings about 20 teams of 9 riders together, among which the player's team. The player controls the actions (attack, sprint, relay …) of his/her riders as well as their energy expenditure.

Figure 8: A screenshot of a race in Pro Cycling Manager

Our work focuses on the action selection system of the riders during a race. It is organized into 4 levels:

- **Level 4: Group tactic selection**. A group is composed of riders (up to 3) from the same team on the basis of a leader and whose goals are those of the leader. For example, to catch up with a dangerous opponent, the group tactic could be *Take relays* or *Attack*, with a given effort.
- **Level 3: Individual roles allocation**. This level is in charge of giving a role to each member of the group. For example, if the chosen tactic is *Attack*, only the leader will attack. On the contrary, if the chosen tactic is *Take relays*, all the group will participate to the relay except the leader.
- **Level 2: Role fulfillment**. Each role is controlled by a Finite State Machine (FSM). For example, the states of the FSM of the role *Take relays* are: *Enter the relay queue*, *Take a relay*, *Go to the end of the relay queue*.
- **Level 1: Dealing with physics**. Here are computed the forces applied on a rider, based on its role, its energy, its competences and its environmental conditions (slope percentage, wind speed and direction, obstacles …).

The player interacts with his/her riders by giving them individual roles. Levels 1 and 2 are therefore shared by player's riders and computer controlled riders.

The transformation method we described in this paper is applied on the highest and most complex level: the group tactic selection. As it was not implemented exactly as a rule base, but rather as conventional procedural programming, we needed a prior transformation stage that we will not describe because of its specificity to that game.

Application Of Our Method: Results

The starting rule base contained 427 rules bringing together 4506 conditions on 167 variables.

From A Rule Base To A Decision Diagram Base
From these 427 rules, 50 small thematic rule bases of different priorities were created by hand. Each small rule base was automatically transformed into a DD on the basis of a manually optimized variable ordering. On that occasion, the total number of conditions dropped down from 4506 to 939 (we count the number of conditions as the number of edges (1467) minus the number of *else* edges (528)).

From A Decision Diagram Base To an AOD + Refining
The automatic conversion of the DD base into an AOD eliminated the 528 *else* edges but, even after the refining of this AOD, necessitated 741 composite nodes.

Testing
We integrated the obtained AOD in Pro Cycling Manager and successfully verified its functional equivalence with the initial group tactic selection module by comparing their decisions on a very large number of race situations.

DISCUSSION

The transformation of a rule base into a DD base, when the division into small rule bases and the variables orderings are carefully chosen, significantly reduces the redundancy of conditions shared by multiple rules. Concerning the conversion of a DD base into an AOD, the diagram size is slightly increased to provide a hierarchical and modular structure allowing a large action selection system to remain easily tunable and expandable.

Our method could be improved by automating the steps that are currently processed manually, particularly the variable ordering step which was the most time consuming. This problem is computationally hard but, for rule bases of reasonable sizes, an automatic solution could help.

AODs are not the ultimate representation of an action selection system. At least they are less powerful than Behavior Trees (BTs). AODs are close to BTs since AND nodes in AODs correspond to parallels in BTs and OR nodes correspond to selectors, but BTs can also represent sequences that endow them with memory and let them behave like HFSMs. Since the AOD formalism is very close to the one of BTs, one can create a BT from an AOD by reorganizing it and adding sequences to it. Conversely, the automatic factorization and simplification tools we designed for AODs could be adapted to BTs (more precisely to sub-BTs that do not contain sequences) that were written by hand, resulting in more compact behavior definitions.

Our initial AI was hard coded. We used the opportunity of its transformation to separate its formalism from the game code. It made on-the-fly modifications possible and gave also the possibility to create tools to help analyzing and editing the AI.

CONCLUSION

In order to rewrite and compact a game AI based on if-then rules into a formalism facilitating its maintenance and its capacity to grow in size, we proposed a method to semi-automatically transform a rule base into an And/Or Diagram (AOD). On a game that continues to evolve over years, our method allows modernizing an action selection system by reusing the work of past generations of AI programmers.

In the future, we plan to integrate fuzzy conditions and composite nodes to allow defining a behavior with a lower number of nodes, and fuzzy actions to prevent crisp changes of behavior under small changes of circumstances. We also plan to develop tools to help the editing process of AODs by masking irrelevant parts to the user.

REFERENCES

Akers. S. B. 1978. “Binary Decision Diagrams”, in *IEEE Transactions on Computers*, C-27(6):509–516.

Bahar R. Iris, Frohm E. A., Gaona C. M., Hachtel G. D., Macii E., Pardo A., et Somenzi F. 1993. “Algebraic decision diagrams and their applications”. In *Intl. Conf. Computer-Aided Design*, 188-191, IEEE.

Champandard A. 2008. “Getting Started with Decision Making and Control Systems”. In *AI Game Programming Wisdom 4*, Charles River Media Inc., 257-264.

Drechsler R., Sieling D. 2001. “Binary decision diagrams in theory and practice”. In *Int. Journal on Software Tools for Technology Transfer*, 3(2):112-136.

Harel, D. 1987. “Statecharts: A Visual Formalism for Complex Systems”. In *Science of Computer Programming*, 8:231-274.

Huang, J., et Darwiche, A. 2004. “Using DPLL for efficient OBDD construction”. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, 127-136.

Tani S., Hamaguchi K., Yajima S., 1993. “The complexity of optimal variable ordering of a shared binary decision diagram”. In *Proc. 45th ISAAC*, *Lecture Notes in Computer Science, Springer*, Vol. 762:389-39.