STAIRS 2014
U. Endriss and J. Leite (Eds.)
© 2014 The Authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-421-3-61

Value Iteration for Relational MDPs in Rewriting Logic

Lenz BELZNER [a,1]
[a] *LMU Munich, PST Chair*

Abstract. Relational approaches to represent and solve MDPs exploit structure that is inherent to modelled domains in order to avoid or at least reduce the impact of the *curse of dimensionality* that propositional representations suffer from. By explicitly reasoning about relational structure, policies that lead to specific goals can be derived on a very general, abstract level; thus, these policies are valid for numerous domain instantiations, regardless of the particular objects participating in it. This paper describes the encoding of relational MDPs as rewrite theories, allowing for highly domain-specific domain encoding and abstraction. Narrowing is employed to solve these relational MDPs symbolically. Resulting symbolic value functions are simplified by Ax-matching abstract state terms. It is shown that relational state representations significantly reduce the size of state space and value function when compared to propositional representations.

Keywords. RMDP, Symbolic Value Iteration, Rewriting Logic, Narrowing

1. Introduction

The framework of *Markov decision processes* (MDPs) allows to model domains with non-deterministic action outcomes and arbitrary reward functions, thus serving well for modelling problems of *sequential decision making under uncertainty* [15]. Various exact and approximate techniques to solve MDPs exist, such as *value iteration*, *policy iteration* and *modified policy iteration* [18]. Given a reward specification (i.e. system goals), a solution of a MDP can be computed that is either a value function mapping states to their corresponding expected values (according to the reward function of the MDP) or a policy mapping states to actions that are maximizing the expected reward as the policy is executed. Algorithms for solving MDPs suffer from the *curse of dimensionality* [1], rendering them infeasible for large-scale domains. To overcome this problem, effort has been made to exploit inherent structure of domains by employing factored or relational, first-order representations of states and actions instead of propositional ones, allowing to represent structured problem domains more concisely. Thus, computation becomes feasible also for larger domains, but the additional complexity that arises from structured domain representations has to accounted for when solving the according MDP [3,12,16].

Rewriting logic is a formal logical framework that lends itself naturally to modelling non-deterministic and concurrent domains on a symbolic level [14,5]. It provides

[1]This work has been partially funded by the EU project ASCENS, 257414.

support for formally specifying structured domains through modularization and object-orientation, as well as explicitly ordered sorts and polymorphism. Previous work of the author introduced the specification of relational MDPs (RMDPs) in rewriting logic [2]. This paper shows how matching [9] and narrowing [10,4] can be employed to solve them using first-order abstraction and avoiding propositionalization. A model-based dynamic programming algorithm for relational MDPs is introduced that employs first-order reasoning and computes exact solutions with the following properties:

- It operates on order-sorted state representations, allowing for polymorphism.
- User-definable constraints assure regression of valid states.
- Only goal-relevant abstract states are regressively constructed and evaluated.
- Reasoning is performed on the first-order level wherever possible.
- The state-action space is factored, leading to concise results.
- Partial goal specifications are supported.

Using rewriting logic to encode domains as RMDPs allows to incorporate features like user-definable term syntax and equivalence classes, sort-ordering, polymorphism and object-orientated representation [6]. This enables domain encoding with only a small representational difference to human expert knowledge. E.g., as many modern software systems are modelled according to the OO paradigm, this way to specify system autonomicity may help to brigde the gap between software engineering and AI techniques.

The paper is outlined as follows: Section 2 discusses in more detail value iteration to solve MDPs exactly as well as the rewriting logic framework and the concepts of matching and narrowing. Section 3 describes how RMDPs can be encoded in terms of a rewrite theory and how rewriting logic concepts can be used to solve these relational MDPs symbolically. Section 4 discusses experimental results and a visualization approach. Finally, section 5 compares the approach to related work, summarizes the results described in this paper and hints at possibilities for further research.

2. Preliminaries

2.1. MDPs and Value Iteration

Definition 2.1. A *Markov decision process (MDP)* is a tuple (S, A, T, γ, R) with S a set of states, A a set of actions, $T : S \times A \times S \to \mathbb{R}$ a transition function, $\gamma \in [0;1]$ a discount factor and $R : S \to \mathbb{R}$ a reward function.[2]

A tuple as given in definiton 2.1 specifies the non-deterministic, discrete time dynamics of a domain in terms of a transition system. The transition function T encodes the probability that executing an action $a \in A$ in a particular state $s \in S$ will result in a state $s' \in S$; note that s and s' may be equal, indicating absence of an action effect. The discount factor γ reflects how much an agent prefers immediate over long-term rewards; the smaller γ is chosen, the more immediate rewards will impact behaviour of an agent acting according to the MDP. The reward function defines incentives that are given to the agent in particular states; in other words, it specifies which states are valuable to achieve.

[2]State-based rewards are used for the sake of simplicity. The approach described in this paper can be extended straightforwardly to allow for transition-based rewards as well.

Definition 2.2. A *value function* $V : S \rightarrow \mathbb{R}$ maps states to values. The value of a state $s \in S$ is the reward gained in s plus the expected discounted future reward when acting greedily w.r.t. V. A *policy* $\pi : S \rightarrow A$ maps states to actions. An agent *acting according to a policy* π executes action $\pi(s)$ when being in state s.

Solving a MDP means to compute either a value function V mapping states to expected values w.r.t. the given reward function, or to provide a policy π that maps states in S to actions in A that are going to maximize the expected reward of an agent acting according to π.

$$V_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A} \left(\sum_{s' \in S} T(s,a,s') V_i(s') \right) \tag{1}$$

Equation (1) shows the *value iteration* algorithm that computes V iteratively, which is guaranteed to converge to the optimal solution [1]. The general idea is that the value of a state is the sum of the reward this state will expose to the agent and the expected discounted future reward when moving on to the next state by executing an action that is assumed to be optimal w.r.t. the current value function V_i. V_0 can be arbitrarily initialized, a common approach is to set $V_0(s) = R(s)$. Iteration is performed until $|V_{i+1}(s) - V_i(s)| < \varepsilon(1-\gamma)/\gamma$ for each state $s \in S$ and a given error bound $\varepsilon \in \mathbb{R}$. This ensures that the maximum difference of V_{i+1} to the real value function V is smaller than ε for all states [18].

2.2. Rewriting Logic

Rewriting logic is suited towards the formal specification of non-deterministic, concurrent systems. System states are encoded in user-definable terms that are constructed from specified operations that can be enhanced with axioms like associativity, commutativity or idempotency. System dynamics are then described in terms of so-called rewrite rules, that allow to specify non-deterministic and concurrent behaviour. The core element to describe systems in rewriting logic are *rewrite theories*, i.e. tuples of the form $(\Sigma, E \cup A, R)$, where Σ contains sorts and operations that are used to construct state terms, E is a set of equations that define equivalence classes for these terms, A is a set of axioms like associativity, commutativity or idempotency specified for operations in Σ, and R is a set of rewrite rules that define the system dynamics. A rewrite theory represents a transition system, where states are terms in Σ, and rewrite rules in R define state transitions. A central concept in rewriting logic is *matching*, which is performed modulo axioms and with extensions (*Ax-matching*, denoted $:=_{Ax}$). Consider an infix operation $\circ$ being associative and commutative, i.e. rendering terms constructed by means of $\circ$ into a *multiset*. Then, for example, $a \circ b :=_{Ax} a \circ c \circ b$, as the latter has the same equivalence class as the term $a \circ b \circ c$ (due to commutativity) and $a \circ b$ is a subterm of $a \circ b \circ c$.

While state terms in Σ specify the static representation of a system, its dynamics are formalized in terms of *rewrite rules* of the form $label:\ t \ \rightarrow\ t'$ **if** $Conditions$ where t and t' are terms of the same kind and may contain free sorted variables. If t Ax-matches a given subject term, the subject term's matched portions are rewritten to t'. A rule's label may be omitted. Rewrite rules can optionally be conditional, in which case rewriting only is applied if all conditions evaluate to true (e.g. sort tests, boolean conditions or matching and rewriting conditions checking whether a term has a certain structure w.r.t. rewrite theory). As a rule may match different portions of a subject term, this representation of system dynamics offers a natural way to model concurrency by rewriting a term on various positions simultaneously according to one or multiple rewrite rules. On the other hand, non-determinism is expressed if (partially or completely) overlapping portions of a subject term match one or more rewrite rules, i.e. if different applications of rewrite rules are possible without being applicable concurrently. In this case, a subject term evolves non-deterministically in any possible way. For example, consider the rewrite rules $(i) : a \rightarrow a'$, $(ii) : a \rightarrow a''$ and $(iii) : b \rightarrow b'$. Then, the term $a \circ b$ rewrites to $a' \circ b'$ by applying rules (i) and (iii), and also to the form $a'' \circ b'$ (using rules (ii) and (iii)).

While rewriting treats variables in a rewriting problem universally quantified, i.e. answering a problem of the form $\forall \vec{x} : t(\vec{x}) \rightarrow_? t'(\vec{x})$, a technique called *narrowing* deals with corresponding problems where variables are treated existentially, i.e. $\exists \vec{x} : t(\vec{x}) \rightarrow_? t'(\vec{x})$, representing *symbolic reachability* problems. To answer queries of this form, instead of matching rules and subject terms as in rewriting, they are *unified* in order to perform narrowing, meaning that variables in both terms may be instantiated to achieve syntactic term unification. I.e., when narrowing, rewrite rules are applied if (one or more subterms of) the subject term can be unified with a rule's lefthand side. Note that, when narrowing, righthand sides of rewrite rules may contain variables not specified in their lefthand side, allowing rewrite rules to introduce fresh variables. For an in-depth discussion of rewriting logic and the concepts of matching, rewriting and narrowing see for example [6].

3. Value Iteration for Relational MDPs in Rewriting Logic

3.1. Relational MDPs in Rewriting Logic

In order to perform symbolic value iteration, a relational MDP (S, A, T, γ, R) is encoded as a rewrite theory $(\Sigma, E \cup A, R)$. States are represented as associative-commutative terms with user-definable syntax parametrizable with first-order variables, thus allowing to specify relations between domain objects and to avoid propositional representation where possible. Axioms and equations of the rewrite theory define equivalence classes for state (and action) terms, which in turn are representation of non-ground, abstract states. Also, state terms can be constrained by equations. This allows for a concise representation of MDPs even when there is a large number of domain objects. Also, as will be shown in section 3.2, a RMDP specified as a rewrite theory can be solved completely symbolically, only grounding variables where this is relevant for goal reachability.

For example, in a fluent-based representation, dynamically changing relations of domain objects can be encoded by a corresponding sort FLUENT. Negation of fluents (i.e. the explicit absence of a particular state property) is represented by an operation $\neg$: FLUENT $\rightarrow$ FLUENT. The state space is constructed in terms of a sort STATE (with FLUENT being a subsort of sort STATE) by an associative and commutative operation $\wedge$: STATE $\times$ STATE $\rightarrow$ STATE that is representing logical conjunction. A constant *false* is defined for sort STATE to denote constraint violations, and $\neg F \wedge F = \mathit{false}$ for all $F \in$ FLUENT. State terms that are syntactically constructed in this way may still contain semantic inconsistencies. Semantic constraints (e.g. state invariants) can be specified in terms of equations that reduce states that violate constraints to *false*. States that violate constraints (e.g. invariants) are reduced to *false*: $S \wedge \mathit{false} = \mathit{false}$ for all $S \in$ STATE. The set of equivalence classes of state terms can then be considered the set S of states of a relational MDP. Note that each equivalence class may render different instances of state terms equal according to their relational structure, thus providing a representation of abstract first-order states. Primitive actions executable by agents are encoded by a parametrizable sort ACTION. Equivalence classes on terms of sort ACTION then form the set of actions A of a relational MDP. As for state terms, free variables are allowed in action terms. The action space is thus raised to an abstract level that allows to exploit its relational structure, especially when taking into account relations between state and action space, e.g. if state and action terms share free variables.

Example 3.1. Consider a signature with sorts TRUCK, BOX and CITY and the sorts FLUENT, STATE and ACTION as above. One can then for example define a fluent $on : \text{BOX} \times \text{TRUCK} \to \text{FLUENT}$. Consider e.g. a polymorph fluent *in* defined similarly. Then $in(t,c) \wedge on(b,t) \wedge in(b',c)$ is a term of sort STATE in Σ.[3] An action representing a truck loading a box can be defined by an operation $load : \text{TRUCK} \times \text{BOX} \to \text{ACTION}$. Then, $load(t,b)$ is an ACTION-term in Σ. The constraint that a truck can only be in one city at a time can be specified by a conditional equation $in(T,C) \wedge in(T,C') \wedge S = false$ **if** $C \neq C'$. ▲

To encode the relation of states, actions (e.g. an optimal action in a particular state) and any corresponding values (e.g. a state's probability to be reached or its expected value), *SAV-tuples* (state-action-value tuples) of the form (s,a,v) are defined. When either a or v are omitted, the relation is valid for all actions or values, respectively. Note that SAV-tuples may relate state and action terms that share variables, consider for example a SAV-tuple $(s(\vec{x} \cup \vec{y}), a(\vec{y} \cup \vec{z}), v)$ where state and action share the variables $\vec{y}$. Thus, SAV-tuples allow to partition not only the state space, but the combined state-action space.

To model the transition function T of a MDP, a rewrite rule can be defined for any transition $T(s,a,s') = p$ to specify this transition in a rewrite theory[5]. The transitions in T for an action $a \in A$ are encoded as a disjunctive rewrite rule (with $\sum p_i = 1$; considering $\vee$ as disjunctive constructor for sets of SAV-tuples):

$$(s,a) \to (s'_1, p_1) \vee (s'_2, p_2) \vee \ldots \vee (s'_n, p_n) \ .$$

The state terms s and the s'_i can be considered as pre- and postconditions of action a. This representation of domain dynamics provides a solution to the *frame problem* [13], avoiding the necessity to specify all state properties unchanged by action execution.

Finally, a MDP's reward function R is represented in terms of appropriate equations, e.g. $reward(in(b,c) \wedge S) = 1.0$. Reward is considered to be zero for all other states.

Example 3.2. Consider action *load* from example 3.1. If a truck executing this action succeeds to load a box (supposing the truck is in the same city as the box) with a proba-

[3]The following notational conventions are introduced, unless stated otherwise: Lowercase letters represent terms (that may contain free variables), uppercase letters represent free variables. In particular, $t, t', T, \in$ TRUCK, $b, b', B \in$ BOX and $c, C \in$ CITY represent constants and free variables denoting domain objects; $s, s', s'_i \in$ STATE and $a, a' \in$ ACTION represent state and action terms, $S \in$ STATE denotes a free state variable; $p, p_i, v, v' \in \mathbb{R}$[4] denote probabilities of transitions and values of states, respectively.

[5]In order to match or unify a subject SAV-tuple with the rule's lefthand side in MAUDE, the rule has to be encoded in the form $(s \wedge S, a) \to (s' \wedge S, p)$. This encoding also explicitly shows the solution to the frame problem.

bility of 0.9, and fails to load it with a probability of 0.1, these transitions are expressed in terms of the following rewrite rule:

$$\begin{gathered}(in(T,C) \wedge in(B,C) \wedge S,\ load(T,B)) \rightarrow \\ (in(T,C) \wedge on(B,T) \wedge S,\ 0.9) \vee (in(T,C) \wedge in(B,C) \wedge S,\ 0.1)\ .\end{gathered}$$

Note that if there were multiple types of boxes, e.g. light and heavy ones with different dynamics, these can be specified by exploiting sort-orders and polymorphism. For example, sorts LIGHT-BOX and HEAVY-BOX could be defined as subsort of sort BOX, allowing for polymorph specification of transition dynamics' rewrite rules. ▲

Note that specifying RMDPs as rewrite theories is parametric in the underlying representation of states and actions. Thus, while the example in this section treated the encoding of RMDPs as rewrite theories for a fluent-based representation, the approach can easily be transferred to other representational paradigms, e.g. OO-MDPs [8] or object-focused MDPs [7], as rewriting logic provides user-definable syntax, equational abstraction and order-sorted, polymorph specification of objects and operations [6]. Note that the MAUDE language also implements support for these features.

3.2. Symbolic Value Iteration

Regression through Narrowing. To allow for regressive induction of state-action space abstractions from given goal states, rewrite rules that specify domain dynamics are transformed into regressive rewrite rules. The key idea is to define for a given state from which preceding states it can be reached by execution of a particular action, and with what probability this action will lead to the given state. The value of the reached state is then used to compute values of preceding states according to transition probabilities.

Consider a relational MDP (S,A,T,γ,R), a value function $V : S \rightarrow \mathbb{R}$, and a rewrite theory $(\Sigma, E \cup A, R)$ encoding the MDP as outlined in section 3.1. Then, the regressive dynamics of an action can be specified by inverting the rewrite rule that specifies an effect for this action (which is of the form $(s,a) \rightarrow (s'_1, p_1) \vee \ldots \vee (s'_n, p_n)$) for each of the effects specified in its righthand side, i.e. for each $i \in [1,\ldots,n]$:

$$(s'_i, V(s'_i)) \rightarrow (s, a, V(s'_i) * p_i * \gamma)\ .$$

Example 3.3. Let $V \in \mathbb{R}$ encode the value for particular abstract states, then the regressive rewrite rules for action *load* from example 3.2 are:

$$(in(T,C) \wedge on(B,T) \wedge S,\ V) \rightarrow (in(T,C) \wedge in(B,C) \wedge S,\ load(T,B),\ V * 0.9 * \gamma).$$

$$(in(T,C) \wedge in(B,C) \wedge S,\ V) \rightarrow (in(T,C) \wedge in(B,C) \wedge S,\ load(T,B),\ V * 0.1 * \gamma).$$

▲

For value iteration, this representation of domain dynamics serves two purposes. First, it allows to compute from a given value function V all abstract (i.e. relational) states from which a particular state $s' \in Domain(V)$ is reachable by narrowing (for a single step) a SAV-tuple (s', v) with $v = V(s')$ according to the inverted rewrite rules of a rewrite theory encoding a RMDP. Second, with regard to value iteration as outlined in equation 1, this narrowing step resembles computation of $\gamma * T(s,a,s')V_i(s')$ when computing $V_{i+1}(s)$: It produces a set of SAV-tuples $(s,a,v*p*\gamma)$ that encode the action a that, when executed in state s, would result in state s' with probability p.

As narrowing is employed, variable grounding is (only) applied where necessary (i.e. where relevant to the reachable state s') through unification of the state that is currently regressed with the lefthand sides of rewrite rules specifying domain dynamics. Also, if the state to be regressed misses any action postconditions, these are induced to regressed state terms by unification if the subject term to be regressed and action effect rules' lefthand sides contain a free state variable (i.e. are of the form $s \wedge S$). Thus, also partially specified goal states can be regressed. If system goals are specified in V_0 (e.g. by instantiating V_0 with a set of SAV-tuples $(s,R(s))$, thus resembling the MDP's reward function R), narrowing exactly grounds variables and induces fluents that are relevant for an optimal policy w.r.t. these goals. Note that regression may lead to states that violate constraints (see section 3.1), which are ignored by further computation.

Summation of Non-Deterministic Action Effects. Regressing the SAV-tuples of a given value function computes a set of SAV-tuples (s,a,v) denoting the states s from which the states in the value function domain can be reached through execution of action a. While v already incorporates transition probabilities and known state values, it does not yet take into account that an action may have multiple outcomes. In equation 1, this fact is addressed by the summation of expected values of all states that are reachable by execution of a particular action a, weighted by transition probabilities. This computation can be resembled by performing a progressive one-step rewrite of $s \times a$ according to the rewrite rules from the RMDP specification (see section 3.1), and adding up the values of the resulting states s' according to V, weighted by transition probabilities. More formally, for all regressed (s,a,v) compute v by $(s \times a) \rightarrow_1 \bigvee_i (s_i \times p_i) \Rightarrow v = \sum_i (V(s_i) * p_i)$.

Maximization through Abstract State Subsumption. To ensure that only optimal actions for each abstract state remain in the new value function V_{i+1}, only those elements that exhibit the maximal value that can be gained through action execution for each possible state are kept in the set of SAV-tuples. As states are relational, they may overlap or even subsume other states completely. To deal with subsumption, the concept of Ax-matching can directly be employed to model state subsumption, as a more general term Ax-matches a more concrete one. For maximization over the set of actions, a state s' with value v' is removed from the regressed set of SAV-tuples if it is subsumed by another state s with greater or equal value: $(s,a,v) \vee (s',a',v') = (s,a,v)$ **if** $s :=_{\mathrm{Ax}} s' \wedge v \geq v'$.

Example 3.4. Let $(on(b,t) \wedge S, a, 1.0)$ and $(on(B,T) \wedge S, a', 2.0)$ be SAV-tuples in the set to be maximized. In this case, it is clearly preferable to execute action a' when *any* box is on *any* truck, because the expected value of this action is 2.0. Thus, the former tuple can be dropped. Now suppose $(on(b,t) \wedge S, a, 3.0)$ and $(on(B,T) \wedge S, a', 2.0)$ should be maximized. Then the former should not be dropped, as action a is preferable if exactly box b is on truck t; otherwise, if another box or another truck are involved, action a' should be executed. Both tuples are necessary to deduce this behaviour. ▲

Value Iteration & Decision List Policies. To complete a value iteration step according to equation 1 after performing maximization, the currently gained reward for all states in the set of SAV-tuples has to be distributed according to the MDP's reward function R.

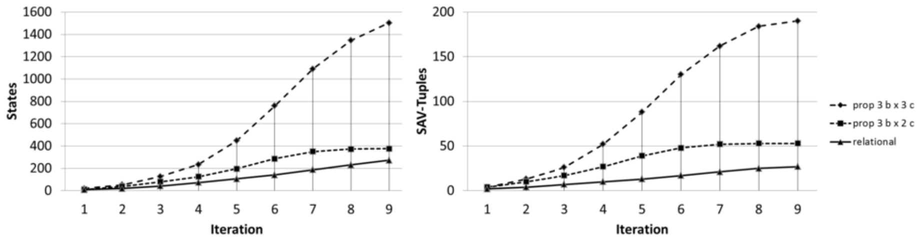

Figure 1. Number of regressed states *(left)* and value function entries *(right)* per iteration step for propositional and relational domains; reward for three boxes at destination.

New SAV-sets resembling a RMDP's value function are iteratively constructed by abstract state regression, summation of state values taking into account non-deterministic action effects, maximization of expected value for states and actions, discounting and reward distribution. For each iteration i, the resulting SAV-set exactly resembles V_{i+1} in value iteration according to equation 1 for all $s \in S$ from which states in the domain of V_i are reachable. States that are not covered by the set are assigned a value of zero, thus the SAV-set can be considered a function. Iteration stops if for all $(s,a,v) \in V_{i+1}$ there exists a $(s,a,v') \in V_i$ such that $|v - v'| < \varepsilon(1-\gamma)/\gamma$ for a given error bound $\varepsilon \in \mathbb{R}$.

The resulting SAV-set representing the converged optimal value function V (with an error bounded by ε) can then be interpreted as a decision list, sorted by values of the SAV-tuples it contains; thus, overlapping and subsuming states are dealt with. I.e., the decision list resembles a policy $\pi : S \to A$ for the MDP that was solved with the presented algorithm, considering $\pi(s) = a \Leftrightarrow (s,a,v) \in V \wedge \forall (s,a',v') \in V : v \geq v'$ and $\pi(s)$ being any action (e.g. *noop*) if $\nexists\, a,v : (s,a,v) \in V$.

4. Evaluation

The approach was experimentally evaluated with an implementation[6] of value iteration in MAUDE [6]. Experiments were conducted with the BOXWORLD domain [3], where a truck has to deliver boxes to their destination cities. A truck can either do nothing, drive from one city to another, load boxes, or unload them. Actions succeed with a probability of 0.9, and fail with no effect otherwise. Comparison was performed for a relational version of the underlying MDP (7 transition rules) and two propositional instantiations of it, one with 3 boxes and 2 cities (37 rules), the other one with 3 boxes and 3 cities (55 rules). Note that the relational rules are valid for *any* number of domain objects. This gain of scalability is not limited to specification, it holds as well for computation of solutions and for value functions themselves. Figure 1 shows the number of regressed states (and size of the value function) per iteration step for the two propositional MDPs and the relational one when a reward is specified for three boxes being in a particular city: The number of regressed states (the number of entries in the value function, respectively) grows substantially with domain size and iteration depth; even the smaller propositional MDP is outperformed by the relational one. As for the specification, the result of symbolic value iteration is valid for *any number of domain objects*. Correctness of the relational solution

[6]The implementation is available at http://www.pst.ifi.lmu.de/~belzner/odin/.

w.r.t the propositional ones was experimentally approved by checking if for each entry in the propositional value function there is a corresponding entry in the relational one.

5. Related Work & Conclusion

Related Work. The first successful approach to solve MDPs with value iteration completely on the symbolic level was achieved by *Symbolic Dynamic Programming* (SDP) [3]. It uses the situation calculus [17] to represent first-order MDPs, thus allowing for full first-order logic quantifications for variables. While the situation calculus is a very expressive specification language, the frame problem has to be addressed explicitly in the specification of domain dynamics, in contrast to specifications in rewriting logic. Another difference of SDP to the approach presented in this paper is that dynamics are defined in a regressive manner and per fluent (in terms of so-called *successor state axioms*) and not per action, diverging from modern software design paradigms as for example object-orientation, where dynamics are typically defined in terms of operations. In consequence, compilation of regressive successor state axioms from progressive, operation-oriented specifications becomes a complex transformation. Also, because of the complexity of regressed state formulas, consistency checking and simplification is a complex task. Even if these tasks are manageable automatically in theory, the authors of SDP only reported on a preliminary implementation that illustrated their approach, but simplification of results was applied manually. The *fluent calculus* [20] can be considered a progressive counterpart to the situation calculus as it represents states as associative-commutative terms of fluents. First-order value iteration for the fluent calculus (FOVIA) [11,12] can be performed in a fully automated manner due to restricted expressivity of the fluent calculus when compared to the situation calculus, as only existential quantification of variables is allowed. As the presented approach, FOVIA performs state subsumption for value function simplification and employs AC1-unification to perform regression, but it is not parametrizable in terms of state representation. In contrast to both SDP and FOVIA, using rewrite theories for symbolic value iteration leads to natural support for flexible, domain-specific representations when it comes to specification of RMDPs, allowing to include features like free choice of syntax and abstraction level, explicit sort ordering, polymorphism or object-orientation.

Summary & Further Work. This paper discussed the representation of RMDPs in rewriting logic and how to use the concepts of matching and narrowing to solve them symbolically, resulting in advantages regarding computational effort and expressivity when compared to propositional solution techniques. To this end, RMPDs were represented as rewrite theories. Regression is performed by exploiting the capabilities of narrowing, allowing for symbolic computation. Simplification of the resulting symbolic value function through state subsumption was realized by Ax-matching state terms. By relating variables in state and action terms, both state and action space are partitioned properly. The specification of RMDPs in terms of rewrite theories allows for domain-specific, user-definable representations by exploiting free syntax choice, sort-hierarchies, polymorphism and object-orientation, achieving small representational gaps between human expert knowledge and domain encoding. A clear reduction of state space and value function size was shown when comparing relational value iteration with rewrite theories to propositional value iteration. A possible direction for future research is to explore the combination of rewrite theories with algebraic representations for RMDPs (e.g. FOADDs, see [19]).

References

[1] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[2] Lenz Belzner. Verifiable decisions in autonomous concurrent systems. In Eva Kühn and Rosario Pugliese, editors, *COORDINATION*, volume 8459 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2014.

[3] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In Bernhard Nebel, editor, *IJCAI*, pages 690–700. Morgan Kaufmann, 2001.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Unification and narrowing in maude 2.4. In Ralf Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] Luis C. Cobo, Charles L. Isbell, and Andrea Lockerd Thomaz. Object focused q-learning for autonomous agents. In Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M. Jonker, editors, *AAMAS*, pages 1061–1068. IFAAMAS, 2013.

[8] Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 240–247. ACM, 2008.

[9] Steven Eker. Fast matching in combinations of regular equational theories. *Electr. Notes Theor. Comput. Sci.*, 4:90–109, 1996.

[10] Santiago Escobar, José Meseguer, and Prasanna Thati. Narrowing and rewriting logic: from foundations to applications. *Electr. Notes Theor. Comput. Sci.*, 177:5–33, 2007.

[11] Axel Großmann, Steffen Hölldobler, and Olga Skvortsova. Symbolic dynamic programming within the fluent calculus. In *Proceedings of the IASTED International conference on Artificial and Computational Intelligence*, pages 378–383, 2002.

[12] Steffen Hölldobler and Olga Skvortsova. A logic-based approach to dynamic programming. In *Proceedings of the Workshop on Learning and Planning in Markov Processes–Advances and Challenges at the Nineteenth National Conference on Artificial Intelligence (AAAI04)*, pages 31–36, 2004.

[13] John Mccarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, volume 4, pages 463–502, 1969.

[14] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, April 1992.

[15] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[16] Aswin Raghavan, Saket Joshi, Alan Fern, Prasad Tadepalli, and Roni Khardon. Planning in factored action spaces with symbolic dynamic programming. In Jörg Hoffmann and Bart Selman, editors, *AAAI*. AAAI Press, 2012.

[17] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, Massachusetts, MA, illustrated edition, 2001.

[18] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[19] Scott Sanner and Craig Boutilier. Practical solution techniques for first-order mdps. *Artificial Intelligence*, 173(56):748 – 788, 2009. Advances in Automated Plan Generation.

[20] Michael Thielscher. Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.*, 2:179–192, 1998.