ELECTRONIC NEWSLETTER ON
REASONING ABOUT ACTIONS AND CHANGE

Issue 98049 Editor: Erik Sandewall 14.6.1998

Today

Debates

David Poole has submitted an article to the ETAI; the title and abstract follows below. Also, Angelo Montanari and co-authors have answered to Peter Jonsson's invited review comments about their ETAI submitted article.

There has been a certain delay since the previous Newsletter due to the editor's participation at the KR conference and the nonmon workshop that preceded it. A report from these conferences is planned for a later newsletter issue.

ETAI Publications

Received research articles

Decision Theory, the Situation Calculus and Conditional Plans

by David Poole

This paper shows how to combine decision theory and logical representations of actions in a manner that seems natural for both. In particular, we assume an axiomatization of the domain in terms of situation calculus, using what is essentially Reiter's solution to the frame problem, in terms of the completion of the axioms defining the state change. Uncertainty is handled in terms of the independent choice logic, which allows for independent choices and a logic program that gives the consequences of the choices. As part of the consequences are a specification of the utility of (final) states, and how (possibly noisy) sensors depend on the state. The robot adopts conditional plans, similar to the GOLOG programming language. Within this logic, we can define the expected utility of a conditional plan, based on the axiomatization of the actions, the sensors and the utility. Sensors can be noisy and actions can be stochastic. The planning problem is to find the plan with the highest expected utility. This representation is related to recent structured representations for partially observable Markov decision processes (POMDPs); here we use stochastic situation calculus rules to specify the state transition function and the reward/value function. Finally we show that with stochastic frame axioms, action representations in probabilistic STRIPS are exponentially larger than using the representation proposed here.

Discussion about received articles

The Complexity of Model Checking in Modal Event Calculi with Quantifiers

by **Iliano Cervesato, Massimo Franceschet, and Angelo Montanari**

Comment from: Angelo Montanari

To: **Peter Jonsson**

> *Page 4, col 1, para 2: Please find a better name than "condition ()".*

Ok. Let us call it "singleton condition" since both the initiate and terminate maps associate each property with a singleton set (or an empty set).

> *Page 6, col 1, para 3: You write "The best approximation of /formula/ we can achieve...". In what sense is it the "best" approximation?*

Ok. Since we are not interested in providing a formal measure of the goodness of an approximation, we will rewrite the sentence as follows: "Let us consider the following approximation of $\Diamond\varphi^i$: ... which in general is not equivalent to, but only implied by, $\Diamond\varphi^i$."

> *Chapter 4 "Implementation". I agree that the implementation of QCMEC in λ-prolog is very elegant. However, due to my own experience of prolog-like languages, I wonder how efficient the implementation is? I suggest that you include a small empirical investigations as follows: First, implement a QCMEC algorithm in some fast, imperative language such as C. This is straightforward since you have already proved that QCMEC is in PSPACE and your way of showing this (by exhibiting a polynomial-time alternating TM) immediately suggests a recursive algorithm for the QCMEC problem. (Even though such a naive algorithm may not be the fastest algorithm, it is probably efficient enough.) Then, compare your implementation in λ-prolog with the C-implementation on a number of test examples (such as the diagnosis example). Even though this experiment does not prove anything, it can very well give us an idea of the (in)efficiency of the implementations. I will not be surprised if the C-implementation completely outperforms the λ-prolog-implementation. I will be extremely surprised if the λ-prolog-implementation outperforms the C-implementation.*

We presented a lambda-Prolog implementation of QCMEC for two reasons: first to show that this formalism is implementable, and quite easily so; second to promote in the AI community languages and a style of programming that permit direct encodings of the problem at hand, to the point that it is feasible to provide rigorous mathematical proofs of correctness (see below).

We did not consider efficiency issues. We would ourselves be very surprised if our lambda-Prolog implementation of QCMEC were to do better than even a naive C implementation. However, it is not our goal to compare the efficiency achievable with different programming languages.

> *Page 8, before Th. 4.1: You write "We only show the statement of our soundness and completeness result since a fully worked out proof would require a very detailed account of the semantics of λ-prolog". So why implement QCMEC in λ-prolog at all? By implementing a straightforward QCMEC algorithm using a language with a more intuitive semantics, the soundness/completeness proof would be more or less trivial (once again, compare with your PSPACE-membership proof of QCMEC).*

We strongly disagree. A correctness proof requires showing that the computation induced by the execution of an algorithm achieves the effects dictated by its specification, and nothing else. If we want to be serious about this, we must in particular relate the individual steps in the computation to aspects of the specification. In the case we consider, this is quite immediate since the representation language (λ-Prolog) is a logic, and the specification of QCMEC is given by means of logical definitions. Relating lambda-Prolog derivations (i.e. execution traces) to these specifications is almost trivial; however, a fully formal proof requires examining all possible cases and all possible rules, making it long and tedious. A proof of this kind can be found in [1][[4] (to be published in the Journal of Logic Programming)]. Had we relied on an implementation in an imperative language such as C, we believe that a similar proof, although theoretically possible, is simply not feasible in practice (at least in a reasonable amount of time).

This was the logician's answer. The vast majority of the people who are likely to cast their eyes on our work will be interested in the more intuitive aspect of a correctness proof that we call immediacy: how far apart is the implementation from the formalism it implements? Here, our lambda-Prolog implementation is a direct transliteration of the specification of the semantics of QCMEC. It would not be so had we used C.

> *Chapter 5 "Complexity Analysis" Why not rearrange the section as follows: Begin the section by proving that QCMEC is in PSPACE. Then there is no need for first showing that CMEC is in PSPACE, extending the result to EQCMEC and finally realize that the proof for EQCMEC also works for QCMEC.*

You propose a top-down organization of the chapter; we arranged it in a bottom-up fashion, which gives us for free some of the hardness results (e.g., QCMEC-hardness immediately follows from EQCMEC-hardness). We prefer to keep things as they stand.

> *Page 8, col 2, proof of Th. 5.3.: You write "...the algorithm enters in an AND (resp. OR) state. It nondeterministically choose one among α and β and evaluates it in w." This should be rephrased. If you are in an OR state, the sentence is correct but if you are in an AND state, no nondeterministical choice takes place. Instead, the ATM checks that all paths starting in the AND state is an accepting path.*

In the paper, we describe an alternating polynomial time algorithm. The algorithm neither checks that all paths starting in an AND state are accepting paths, nor searches for an accepting path starting from an OR state. If

[1] Ref: http://www.stanford.edu/~iliano/papers/udmi-37_96.ps.gz

it were to, it would not be polynomial time-bounded. In both cases (AND and OR states), the algorithm simply guesses a path and follows it.

By definition, the alternating machine accepts an AND-state configuration (resp., an OR-state configuration) if and only if all (resp., al least one) of its successor configurations are accepting configurations. Thus, the distinction bewteen AND states and OR states comes into play in the definition of acceptance.

It is worth noting that given a problem, a nondeterministic algorithm for this problem does not solve it. Given a "yes" instance of the problem and a succint certificate for it, the nondeterministic algorithm only checks that the instance is indeed a "yes" instance of the problem by exploiting the certificate. For example, in the case of SAT, given a satisfiable propositional formula F and a truth assignment for the variables in F that makes F true, a nondeterministic algorithm for SAT checks that the assignment satisfies F.

Notice also that AP is equal to PSPACE, that is, finding an alternating polynomial time algorithm for a given problem allows us to conclude that the problem can be solved in polynomial space. Thus, it is not surprising that the behaviour of the algorithm in an AND and in an OR state is the same, since the space requirements in both states are the same.

See [12] (chapter 16) for further details.

Page 10, col 1, end of proof of Th. 5.3. Please give the proof of why $w \vdash F_1$ iff G is true. Or, at least, give the reader a hint...

While answering this comment, we noticed a minor imprecision in the definition of the formula defining F_k. The correct formulation is as follows:

```
      | \Diamond \hat{F}                                 if k = n
      |
F_k = | \Diamond (\psi_{k+1} \And \Box \hat{F})          if k = n-1
      |
      | \Diamond (\psi_{k+1} \And
                  \Box (\psi_{k+2} \Impl F_{k+2}))       otherwise
```

Coming back to your question, we will now provide further details in order to help the reader to understand the proof.

Let G be a QBF with variables in $V = \{x_1, x_2, ...x_n\}$ and H be an EC-structure as defined in the proof.

A knowledge state w in W_H induces an assignment t_w of the variables in V: for every variable x in V, let $t_w(x)$ be true if and only if $w \models p_x(e_x, e_{\neg x})$. Otherwise, that is, if $w \models \neg p_x(e_x, e_{\neg x})$, then $t_w(x)$ is false. Notice that different knowledge states may induce the same assignment. Indeed, only the relative order of the events e_x and $e_{\neg x}$ for every variable x in V is relevant. This many-to-one correspondence between knowledge states and assignments establishes an equivalence relation on the set of all the knowledge states for H such that the truth of a propositional CMEC-formula on H (i.e. a CMEC-formula on H without modalities) is invariant within each equivalence class.

If F is a propositional formula (the matrix of some QBF G), the corresponding (propositional) CMEC-formula $\hat{F}$ is obtained by replacing every occurrence of the variable x in F with $p_x(e_x, e_{\neg x})$. Given a knowledge state

w in W_H, $w \models \hat{F}$ iff t_w satisfies F. Hence, $w \models \Diamond\hat{F}$ (resp. $w \models \Box\hat{F}$) iff F is satisfiable (resp. valid).

In the proof, we exploit this correspondence in order to map the propositional quantifiers of a QBF into the modal operators of a CMEC-formula. However, there is a technical problem with this solution: the scope of a quantifier is limited to the variable that it quantifies, whereas the scope of a modal operator in the formula $\Box F$ or $\Diamond F$ extends to the whole formula F. In the proof, we take advantage of the formulas ψ_k in order to restore the correct context for the evaluation of $\hat{F}$.

We invite the skeptical reader to make some small examples with two or three nested quantifiers.

> *Page 11, col 1: You write "In Section 4 we have transliterated the definition of QCMEC and its subcalculi in the higher-order logic programming λ-prolog. The directness of the implementation allows checking easily that the complexity of the implemented algorithms coincide with the bounds proved in this section for the problems they implement". One should be extremely careful when justifying complexity results with implementations in logic programming languages. Such languages often introduce large overheads and it is difficult to know the exact complexity of the different operations (unification,...) that are performed by the systems. Your assertion may very well be correct, but it needs further justification.*

This observation is true in general of the implementation of any programming language. We do not have enough insight about the internals of the implementation we used to answer it in one way or another. However, because of the relatively simple structure of our programs, we do not expect the overhead introduced by the interpreter we used to alter the bounds we obtained.

This newsletter is issued whenever there is new news, and is sent by automatic E-mail and without charge to a list of subscribers, in plaintext format. The present postscript issue is a bona fide rendering of the same information in alternative typography. Instruction for contributors and other additional information is found at the following URL:

`http://www.ida.liu.se/ext/etai/actions/njl/`