$PDL_{||}$ and its relation to PDL

Fahad Khan
University of Nottingham
afk@cs.nott.ac.uk

Abstract

In this report we examine results pertaining to Karl Abrahamson's $PDL_{||}$, namely PDL with an interleaving operator, $||$, with respect to an agent programming point of view. We first establish its usefulness in such contexts, before defining a syntax and semantics for the logic, looking at its relation to the regular expression shuffle operator and to PDL itself. We also look at the practical implications of this relation between PDL and$PDL_{||}$and of its $PDL_{||}$'s relation to BPDL another logic defined by Abrahamson over a quarter of a century ago.

1. Introduction

Interleaved PDL, denoted as $PDL_{||}$, was first defined by Karl Abrahamson in his 1980 PhD thesis, *Decidability and expressiveness of logics of processes*[2], as an extension of PDL (propositional dynamic logic) with a new operator, the interleaving operator $||$. In fact, the $||$ operator makes a useful addition to the regular four PDL operators[1]allowing the easy expression of the interleaving of two or more $PDL_{||}$programs – and hence facilitating reasoning about the effects of such interleavings. We give more details of the interleaving operator below, but in order to illustrate why reasoning about the interleaving of representations of programs is of particular importance, at least from an agent programming point of view, consider the example of SimpleAPL, a programming language explored in *A logic of agent programs*[3] and used to implement a particular model of basic agents with beliefs, goals, and plans[2].

In SimpleAPL an agent has beliefs, whose role it is to encode various aspects of its environment, and goals, which encode representations towards the realisation of which the agent works by adopting plans which are selected in turn via planning goal rules. Both beliefs and goals are represented by literals; plans on the other hand are composites built up from a set of basic actions via sequencing, conditional choice and conditional iteration operators. In [3] Alechina et al. detail two execution strategies for executing an agent program, the first of which allows either for an agent with no plan to select a planning goal rule and choose a single plan, or for an agent with a plan to execute the next step in the single plan which it carries; in the second an agent can amass a number of plans at any single juncture, consequently interleaving the execution of these plans or selecting another planning goal rule. So for example, with the first strategy, an agent would have to carry out the plans 'make coffee' and 'make toast' one after the other – potentially leaving it with a cold cup of coffee or piece of toast – whereas the second strategy would allow for multitasking as it were, allowing it to carry out actions associated with either of these plans in their correct order within the plan, but otherwise in whatever order was preferred.

Furthermore, Alechina et al., develop a sound and complete variant of PDL with which they are able to reason about the safety and liveness properties of agent programs in SimpleAPL. Crucially it turns out that the interleaved strategy admits of a straightforward formulation through the use of the $||$ operator – which, as we will show, can ultimately be eliminated, thereby allowing any formula of $PDL_{||}$to be equivalently formulated in PDL.

It is clear, at least from the foregoing example, that the usefulness of $PDL_{||}$in the context of agent programming and modelling lies, among other things, in the fact that it allows for the succint expression of agent program execution strategies that incorporate the interleaving of agent plans (incidentally Abrahamson's own original motivation for defining $PDL_{||}$in [2] related to modelling and formally verifying claims about the behaviour of concurrent computer programs).

Given this agent based motivation, the purpose of this report is to collect and elucidate some important relevant technical results involving $PDL_{||}$. To summarise the rest of the paper, we begin by elaborating on the syntax and semantics of $PDL_{||}$as well as on the correspondence between the

[1]Recall that regular PDL programs are built up using nondeterministic union($\cup$), concatenation(;), iteration(*) and query(?) operators.

[2]Note that SimpleAPL is, as the name suggests, a simplified fragment of the more extensive agent programming language, 3APL. See [3] for more details.

shuffle regular expression operator and the interleaving operator and how we can use this equivalence to eliminate the interleaving operator from PDL programs, before describing how we can improve on the size of the resulting formula by using BPDL – another logic defined by Abrahamson in [2]. Finally in the conclusion we briefly consider some of the possible future directions for research suggested by the preceeding results.

2. An Inductive Definition of PDL$_\|$Syntax and a PDL$_\|$Semantics

The following series of definitions serve to define the syntax and semantics of PDL$_\|$.

Definition 2.1 Given a fixed set of proposition symbols, $\Phi = \{p, q, ...\}$, we can inductively define the set of PDL$_\|$formulae as follows:

- each proposition symbol $p \in \Phi$ is a formula,
- if ϕ and ψ are formulae, then so too are $\neg\phi$, $\phi \wedge \psi$ and $\langle\rho\rangle\phi$ where $\rho \in \Psi$ is a program.

Assuming a fixed set of basic programs $\Psi_0 = \{a, b, ...\}$ we inductively construct the set of programs, Ψ, used in the previous definition, in the following manner:

- each basic program $a \in \Psi_0$ is a program,
- if α and β are programs, then so too are $\alpha \cup \beta, \alpha; \beta, \alpha^*$ and $\alpha \parallel \beta$, where the latter is the interleaving operator,
- if ϕ is a formula of PDL$_\|$then $\phi?$ is a program.

◁

We now come to define models for PDL$_\|$:

Definition 2.2 Let M be a structure, $M = (W, \tau, V)$, then M is a model for PDL$_\|$if:

- W is a set of states,
- $V(p) \subseteq W$ is a function that for each $p \in \Phi$ gives us the set of states in W at which p holds. We can extend this in the obvious way so that $V(\phi)$ gives us the set of states where the PDL$_\|$formula ϕ holds: given the PDL$_\|$formulae ϕ, ψ and the program ρ, $V(\neg\phi) = W - V(\phi)$, $V(\phi \vee \psi) = V(\phi) \cup V(\psi)$ and $V(\langle\rho\rangle\phi)$ equals the set $U \subseteq W$ consisting of all the states of $u \in W$ such that there exists a computation sequence $\sigma \in \tau(\rho)$ (we define τ below) where either $(u, u_1), ..., (u_n, v)$ where $v \in V(\phi)$ and σ is a legal sequence, or where $\sigma = \epsilon$ and $u \in V(\phi)$. Note that by legal computational sequences we are referring to sequences ρ such that whenever $(s_1, s_2)(s_3, s_4)$ is a subword of ρ, then $s_2 = s_3$,
- $\tau(a) \subseteq (W \times W)$ gives us the set of state transitions for a. We can extend this inductively to give us a set of paths $\tau(\rho) \subseteq (W \times W)^*$ corresponding to any PDL$_\|$program expression ρ in M:
  - $\tau(\phi?) = \{(u, u) : u \in V(\phi)\}$,
  - $\tau(\rho_1 \cup \rho_2) = \{z : z \in \tau(\rho_1) \cup \tau(\rho_2)\}$,
  - $\tau(\rho_1; \rho_2) = \{z_1 \circ z_2 : z_1 \in \tau(\rho_1),\ z_2 \in \tau(\rho_2)\}$, where $\circ$ is a concatenation of paths operator,
  - $\tau(\rho^*)$ is the set of all paths consisting of zero or finitely many concatenations of paths in $\tau(\rho)$,
  - $\tau(\rho_1 \parallel \rho_2)$ is the set of all paths obtained by interleaving atomic actions and tests from $\tau(\rho_1)$ and $\tau(\rho_2)$.

  Note that the set of paths $\tau(\rho) \subseteq (W \times W)^*$ may contain non-legal sequences – in fact, to do otherwise would be to place a severe restriction on our ability to interleave sets of paths.

◁

3. Shuffling and Interleaving

From the foregoing series of definitions it is easy to see that the program constructors $\cup, ;$, and * correspond to the regular expression (RE) operators $+, \cdot$, and *, respectively. However, given that under our definition of PDL$_\|$basic programs are indivisible, we are in a position to define another RE operator, shuffle, which corresponds to our interleaving operator and which we will also denote using $\|$.

Let $x, y \in \Sigma^*$, where Σ is a finite alphabet, and x, y are strings over Σ. Then the shuffle of x and y, namely, the set $x \parallel y$, is defined (in for example, [5]) as:

- $\epsilon \parallel y = \{y\}$,
- $x \parallel \epsilon = \{x\}$,
- $xa \parallel yb = (x \parallel yb) \cdot \{a\} \cup (xa \parallel y) \cdot \{b\}$.

Furthermore we define the shuffle of two languages X, Y as follows:

$$X \parallel Y = \bigcup_{\substack{x \in X \\ y \in Y}} x \parallel y.$$

Since for any two RE's α and β, we intend the language $L(\alpha \parallel \beta)$ to accept all strings x such that x belongs to the shuffle of the languages $L(\alpha)$ and $L(\beta)$ – where $L(\alpha)$ and $L(\beta)$ are the languages of α, β respectively – we define $L(\alpha \parallel \beta)$ as $L(\alpha) \parallel L(\beta)$.

As an example, take the shuffle of the two sets $\{ab\}$ and $\{cd\}$, namely $\{ab\} \parallel \{cd\}$, which gives us the set $\{abcd, acbd, acdb, cabd, cadb, cdab\}$ or the shuffle of the

two RE's a^* and $(b;c)$, $a^* \parallel (b;c)$, which results in the set of strings of arbitrary length including the string bc in which b and c are inserted within a series of one or more repetitions of the character a: in other words the set of strings satisfying the RE $a^*(b)a^*(c)a^*$.

Given the correspondence of shuffle with our interleaving operator (indeed we will use the terms 'shuffle operator' and 'interleaving operator' interchangeably from hereon in) and the fact, which we will presently demonstrate, that any instance of the shuffle operator in a given regular expression can be eliminated, it is clear that we can translate any PDL$_\parallel$formula ϕ into a PDL formula ϕ'. And we do this by simply replacing each program ρ that occurs in a subexpression $\langle\rho\rangle\chi$ of ϕ with its equivalent RE which we consequently translate, using the method we summarise below, from a shuffle RE into a shuffle free RE before translating it back into a program ρ' and replacing the subexpression $\langle\rho\rangle\chi$ with its equivalent $\langle\rho'\rangle\chi$.

In fact, as we now show, we can directly translate any formula of PDL$_\parallel$containing no instances of the * operator into PDL using RE equivalences – the other method which we detail below and which can be applied to any formula of PDL$_\parallel$involves a detour into automata theory. Along with the usual RE equivalences this direct translation requires the following regular expression equivalences (the proofs are trivial and are therefore omitted):

Proposition 3.1

- *(i) For all regular expressions α,β, and γ, we have that $\alpha \parallel (\beta + \gamma) \equiv (\alpha \parallel \beta) + (\alpha \parallel \gamma)$ and $(\alpha + \beta) \parallel \gamma \equiv (\alpha \parallel \gamma) + (\beta \parallel \gamma)$.*
- *(ii) For all strings $x, y \in \Sigma^*$ and $a, b \in \Sigma$, where Σ is some alphabet we have that $xa \parallel yb \equiv (x \parallel yb)a + (xa \parallel y)b$.*

Now given a formula χ in *-free PDL$_\parallel$we can use the following algorithm to remove the instances of shuffle embedded in χ:

- Step 1: list all of the subformulae ϕ of χ,
- Step 2: let ϕ' be a maximal such subformula of χ, maximal in that it does not contain a subformula of the form $\lambda_1 \parallel \lambda_2$ and there is no shuffle free subformula of χ of which it is a proper subformula,
- Step 3: rewrite ϕ' in the form $\phi_1 + \phi_2 + ... + \phi_n$, where each ϕ_i is a concatenation of characters. We can do this through repeated application of the regular expression equivalences $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$ and $(\alpha + \beta)\gamma \equiv \alpha\gamma + \beta\gamma$. Replace each such maximal subfomula ϕ' with its rewriting,
- Step 4: now we rewrite each subformula ψ of the form $\psi = x \parallel y$ where x and y are shuffle free (and which thanks to our previous operations are in the form we require) in terms of its equivalent of the form $\psi' = \psi_1 + ...\psi_k$ where each ψ_i is of the form $x_i \parallel y_i$ where x_i, y_i contain only concatenations of symbols via repeated applications of the the equivalences proved in Proposition 3.1 (i). Next we get rid of shuffle from each ψ_i by rewriting ψ_i using the equivalence proved in Proposition 3.1 (ii),
- we end up with a new formula χ' with which we can repeat the previous steps until we've gotten rid of all instances of shuffle.

Note that each rewriting of a subformula ψ_i of size n using the equivalence proved in Proposition 3.1(ii) in Stage 4 results in a formula ψ_i' of size $O(2^{p(n)})$ – meaning that the method we will now detail for the elimination of shuffle in any formula of PDL$_\parallel$and which guarantees us a double exponential bound on the size of the formula resulting from the translation is preferable in most cases.

In fact this next method (also known as the "brute force" method) seems to be the most straightforward means of translating any regular expression, α, containing one or more instances of the shuffle operator into an equivalent RE constructed solely in terms of the RE operators $\cup$,* and ;, and it proceeds in two steps. We begin by translating α into a nondeterministic finite automaton (NFA) M using a special cross-product construction, this is the first step; the second step consists of translating M back into an RE. Unfortunately the combination of these two operations entails, in the worst case, a double exponential blowup in the size of the resulting RE. We now describe in greater detail both of the steps comprising this translation method.

For the first step, we proceed inductively starting with an instance of a regular expression $\alpha = \alpha_1 \parallel \alpha_2$ consisting of a shuffle operator applied to two shuffle free RE's α_1 and α_2, as our base case.

Now, we can convert the two shuffle free RE's α_1 and α_2 into two NFA's $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$, respectively where $L(\alpha_1) = L(M_1)$ and $L(\alpha_2) = L(M_2)$. Note that each of these conversions gives us an NFA that is linear in the size of our original RE, since the conversion algorithm we will use – and which can be found in, for example, Hopcroft and Ullman's famous Automata textbook [4], and in Kozen's textbook on the subject [5] – will only add 2 states for each subexpression of our original RE.

The important thing for us now is to be able to show that the set $L(M_1) \parallel L(M_2) (= L(\alpha_1) \parallel L(\alpha_2))$ can be converted into an NFA M such that $L(M) = (M_1) \parallel L(M_2)$ which we will then convert back into an RE. Obviously it would be simpler – though perhaps not preferable in terms of the size of the resulting formula – if we had a way of generating $L(\alpha_1) \parallel L(\alpha_2)$ directly via an RE as we did with the

translation of the $^*-$free fragment of PDL$_\|$, rather than by taking this circuitous route.

To generate a machine whose language is $L(M_1) \parallel L(M_2)$ we use the following cross-product construction on M_1 and M_2, the result of which is an NFA M_3 which accepts a string x if and only if $x = x_1 \parallel x_2$, where $x_1 \in L(M_1), x_2 \in L(M_2)$:

$M_3 = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F)$, where the transition function δ is defined as $\delta((q_1, q_2), a) = (\delta_1(q_1, a) \times \{q_2\}) \cup (\{q_1\} \times \delta_2(q_2, a))$, and the set of accepting states F is defined as $F = \{(q_1, q_2) \in Q_1 \times Q_2 : q_1 \in F_1, q_2 \in F_2\}$.

A proof of the correctness of the construction can be easily produced on the template of the proofs given for the equivalence of NFAs and DFAs in terms of the class of languages accepted by either, in for example Kozen [5].

Now, given an RE α containing a number of nested $\|$ operators, we can iterate through the subexpressions of α until we reach subexpressions α_i that match our base case, building up a series of NFA's which we can combine either using our cross product constructor or via the rules for building NFA's inductively from RE's – again as set out in for example [4] in the algorithm for converting an RE into an NFA. The upshot is that we have defined an NFA M_3 such that $L(M_3) = L(M_1) \parallel L(M_2) = L(\alpha_1) \parallel L(\alpha_2) = L(\alpha_1 \parallel \alpha_2)$.

Sadly in the worst case this means our NFA M is exponential in the size of our original RE α, i.e., the size of M_3 is $2^{(O|r|)}$ where $r = |\alpha|$. To understand why this is so consider that each basic program constituent, a of α can be translated into an NFA of size 2 and given $r = |\alpha|$ where by necessity $r > 3$, we may potentially need to use the cross product construction kr times, where $1 \leq k \leq \frac{r}{2}$. (Meyer and Stockmeyer use this fact to prove an upper bound for the complexity of PDL$_\|$'s satisfiability problem [6]).

Worse is to come. It seems that the best known algorithms we have for translating an n state NFA into an RE entail, in the worst case, an exponential blow up in the size of the resulting RE, e.g., the algorithm given in [4] gives us an RE of size $O(n^3 4^n)$. This means that after having translated our original RE α of size r into an NFA of size $O(2^r)$ we then end up with an RE of size $O(2^{2^r})$.

4. BPDL

Abrahamson, who first defined PDL$_\|$in his PhD thesis [2], writes in the self same that "[a]ny axiom system for PDL$_\|$which ultimately relies on reducing away concurrency by expressing it in terms of $\cup$,;, or *... is misguided." He suggests introducing auxiliary variables into PDL in order to improve on the double exponential size of the formula that results from the brute force method. In fact, it is relatively simple to see how we can do this in relation to BPDL, an extension of normal PDL that incorporates boolean variables and which is also defined by Abrahamson in his PhD thesis.

BPDL structures feature an additional set, Q, of boolean variables, which we refer to when defining the set of well-defined BPDL formulae – and note that these boolean variables are treated completely separately from the propositional symbols. The definition of the syntax of BPDL is similar to that for PDL$_\|$.

Definition 4.1 Given a fixed set of proposition symbols, $\Psi_0 = \{p, q, ...\}$ and a fixed set of boolean variables $Q = \{x, y, ...\}$, we can construct the set of BPDL programs as follows:

- each basic program $a \in \Psi_0$ is a program,
- for each variable $x \in Q$, $\uparrow x$ and $\downarrow x$ are programs,
- if α and β are programs, then so too are $\alpha \cup \beta, \alpha; \beta$, and α^*,
- if ϕ is a formula of BPDL then $\phi?$ is a program.

$\triangleleft$

The set of formulae of BPDL are defined as for PDL$_\|$, again with reference to a set Φ of propositional variables. We can now define models for BPDL.

Definition 4.2 Let M be a structure such that $M = (W, V_B, \tau_B, Q)$, then M is a model for BPDL if

- W is a set of states,
- Q is a set of boolean variables,
- $V_B(p) \subseteq \wp(W \times \wp(Q))$, where $\wp(Q)$ denotes the power set of Q, is a function that gives us, for each $p \in \Phi$, the cross product with the power set of Q of the set of states in W at which p holds, namely, the set $V(p)$ as defined in Definition 2.2, i.e., $V_B(p) = V(p) \times \wp(Q)$. Additionally, for each $x \in Q, V_B(x) = W \times \{S \subseteq Q : x \in S\}$.

  We extend this function to all formulae of BPDL inductively: given the formulae ϕ, ψ and the program ρ with $V_B(\phi), V_B(\psi) \subseteq W \times \wp(Q)$ and $\tau_B(\rho) \subseteq (W \times \wp(Q))^2$ (we will define τ_B below), the definition runs as follows:

  - $V_B(\neg\phi) = (W \times \wp(Q)) - V_B(\phi)$,
  - $V_B(\phi \vee \psi) = V_B(\phi) \cup V_B(\psi)$,
  - $V_B(\langle\rho\rangle)\phi = \{(u, S) \in (W \times \wp(Q)) :$ there exists $v \in W, T \subseteq Q$ such that $((u, S), (w, T)) \in \tau_B(\rho)$ and $(w, T) \in V_B(\phi)\}$

- $\tau_B(a) \subseteq \wp((W \times \wp(Q))^2)$, is a function that gives us, for each $a \in \Psi_0$, the set $\{((u,S),(w,S)) : (u,w) \in \tau(a), S \subseteq W\}$ where $\tau(a) \subseteq (W \times W)^*$, the set of state transitions of a is defined similarly to the function τ of Definition 2.2.

  Additionally, for each $x \in Q$ we have that $\tau_B(\uparrow x) = \{(u,S),(u,S') \in (W \times \wp(Q))^2 : S' = S \cup \{x\}\}$ and $\tau_B(\downarrow x) = \{(u,S),(u,S') \in (W \times \wp(Q))^2 : S' = S - \{x\}\}$. For programs α and β we define $\alpha \cup \beta, \alpha;\beta$ and α^* as in Definition 2.2. For any BPDL formula ϕ we define $\tau_B(\phi?)$ as $\{((u,S),(u,S)) \in (W \times \wp(Q))^2 : (u,S) \in V_B(\phi)\}$.

◁

It turns out that adding Boolean variables to PDL gives a strong boost to the expressiveness of the resulting language. For example, we can represent any integer in the range $0,...,2^{(n-1)}$ using just n Boolean variables. It is also routine to write programs of length $O(n)$ that add, subtract or compare two such "n-bit" integers. However for our purposes the most important consequence of adding Boolean variables to PDL is that we are able to drastically improve on the double exponential overhead incurred by the shuffle translation method given above.

To see how this is possible note that any NFA M consisting of n nodes can be converted to a BPDL program of length $O(n \log n + c)$ where c is the combined length of the tests on the outgoing edges of each node in M[3]. Obviously, in many cases, this allows us to improve on the exponential size of the PDL program resulting from the usual method of translating NFA's to RE's. The translation proceeds by assigning a numbering to the states of the NFA and constructing a program of the form $S;(\bigcup_i T)^*;F?$ where the subprogram S sets a counter to the number of the initial state; T_i performs the action associated with state numbered i if the counter is in the state numbered i; and finally $F?$ checks whether the counter is in an accepting state.

The easiest way to see how this works is by recourse to an example, here the NFA M illustrated below as Figure 1.

We can easily model the action of M via the following program:

$$
\begin{array}{l}
I := 1; \\
((I=1)?;(a?;I:=2) \cup (b?;I:=3); \\
(I=2)?;(a?;I:=4) \cup (b?;I:=2); \\
(I=3)?;(a?;I:=3) \cup (d?;I:=5); \\
(I=4)?;(c?;I:=6))^*; \\
(I=5)? \cup (I=6)?
\end{array}
$$

[3]Note that we can exploit the nondeterminism of PDL (and hence BPDL) to model the nondeterminism of M.

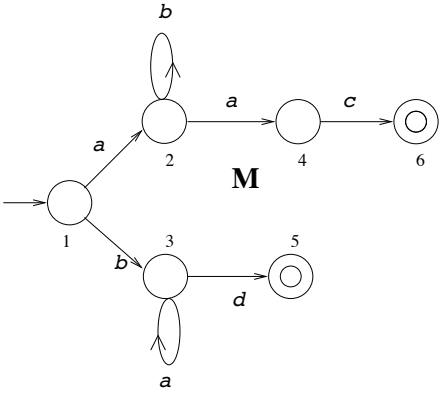

Figure 1. The NFA M

So given a program α, effectively an RE, of length n containing one or more instances of the $\|$ operator we can generate, as detailed above, an NFA M whose size is exponential in n such that $L(M) = L(\alpha)$. If we're willing to translate M into BPDL instead of PDL we end up with a formula of size $O(n2^n + d)$, where d is the length of all the tests in M. Usually this will offer a substantial improvement on the double exponential size of the formula resulting from the translation of M into PDL. Sadly we can't improve on our previous translation of PDL$_\|$ formulae into PDL formulae by inserting an intermediate stage in which we translate our cross product NFA into a BPDL formula before further translating this into a PDL formula: Abrahamson determined in [1] that the translation of a formula from BPDL into PDL has an double exponential lower bound.

4.1. Conclusion

So in summary, we have surveyed a number of the main results concerning PDL$_\|$, most notably the fact that the addition of the $\|$ operator to PDL affords no increase in the expressiveness of the resulting language – however it does seem to give important, and indeed dramatic benefits in terms of the succintness of the formulae we can devise to describe the interleaving of two or more programs. We have looked at a conceptually straightforward method of translating the * free fragment of PDL$_\|$ into PDL – straightforward in that it only makes use of regular expression equivalences – the original contribution of this report; and we have detailed the aptly named "brute force" method. However the double exponential size of the formulae resulting from the brute force method presents a substantial practical obstacle to the application of the interleaving operator in, for example, an agent programming context as described in the introduction. Abrahamson's BPDL – PDL enriched with binary variables – seems to provide one solution to this blow up in complexity, and indeed a further direction for investigation here is the possible use and development of BPDL tools for

the verification of agent programs.

In fact we could go further and keep the interleaving operator as a primitive: future work could investigate the effects of different axiomatisations of $\mathrm{PDL}_{\|}$, or the creation of efficient $\mathrm{PDL}_{\|}$ tools for verification purposes. An investigation into the kinds of regular expression featuring instances of the interleaving operator that admit of a more compact translation into a shuffle free regular expression would also yield useful practical results (it would be also interesting to see if there were other means of translating $\mathrm{PDL}_{\|}$ into PDL than those we have described) . Of course further investigations could also centre on other kinds of agent execution strategies and the various logics which could be developed to describe them, including other extensions of PDL.

References

[1] K. R. Abrahamson. Boolean variables in regular expressions and finite automata. Technical report, Department of Computer Science, University of Washington, 1980.

[2] K. R. Abrahamson. *Decidability and expressiveness of logics of processes*. PhD thesis, Department of Computer Science, University of Washington, 1980.

[3] N. Alechina, M. Dastani, B. Logan, and J.-J. C. Meyer. A logic of agent programs. In *AAAI*, pages 795–800, 2007.

[4] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[5] D. Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997.

[6] A. J. Mayer and L. J. Stockmeyer. The complexity of PDL with interleaving. *Theoretical Computer Science*, 161(1&2):109–122, 1996.