Characterising and Matching Iterative and Recursive Agent Interaction Protocols

Tim Miller
Dept. of Computer Science and Software Eng.
University of Melbourne
Victoria, 3010, Australia
tmiller@unimelb.edu.au

Peter McBurney
Department of Computer Science
University of Liverpool
Liverpool, L69 7ZF, UK
mcburney@liverpool.ac.uk

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Algorithms, Languages

Keywords

multi-agent systems, interaction protocols, characterisation

ABSTRACT

For an agent to intelligently use specifications of executable protocols, it is necessary that the agent can quickly and correctly assess the outcomes of that protocol if it is executed. In some cases, this information may be attached to the specification; however, this is not always the case. In this paper, we present an algorithm for deriving *characterisations* of protocols. These characterisations specify the preconditions under which the protocol can be executed, and the outcomes of this execution. The algorithm is applicable to definitions with infinite iteration, and recursive definitions that terminate. We prove how a restricted subset of non-terminating recursive protocols can be characterised by rewriting them into equivalent non-recursive definitions before characterisation. We then define a method for *matching* protocols from their characterisations. We prove that the complexity of the matching method is less than for methods such as a depth-first search algorithm. Our experimental evaluation confirms this.

1. INTRODUCTION

Research into interaction protocols for multi-agent systems is focused mainly on the documentation of interaction protocols, which specify the set of possible interactions for a protocol in which agents engage. Agent developers use these specifications to hard-code the interactions of agents. We identify three significant disadvantages with this approach: 1) it strongly couples agents with the protocols they use — something which is unanimously discouraged in software engineering — requiring agent code to change with every change in a protocol; 2) agents can only interact using protocols that are known at design time, a restriction that seems out of place with the goals of agents being intelligent and adaptive; and 3) agents cannot compose protocols at runtime to bring about more complex interactions, restricting them to protocols that have been specified by human designers — again, this seems out of place with the goals of agents being intelligent and adaptive.

Executable protocols for goal-directed agents are often specified using action languages [8, 13]. Such languages tend to provide support for specifying actions with preconditions and postconditions, and allow actions to be composed to create more complex behaviour. For agents to become intelligent and adaptable, it is desirable that agents learn new protocols from each other, and are able reason about the outcomes of new protocols.

Miller and McBurney [9] have specified a set of rules for producing *characterisations* of protocols. A characterisation contains information about the precondition under which a protocol is applicable, and the postconditions that are possible if that protocol is executed. The weakness of the rules is that they did not consider iterative or recursive definitions. In this paper, we build on Miller and McBurney's work.

Section 4 presents an algorithm for characterising executable protocols specified in an action language (defined in Section 2) — so called, *first-class protocols* [7]. This algorithm is applicable to iterative protocols, and recursively-defined protocols that terminate. Characterisations are expressed in the form of theorems in propositional dynamic logic [5], a logic for reasoning about the behaviour of programs. In Section 5, we prove that a restricted subset of non-terminating protocols can be characterised using this algorithm by replacing the recursion with iteration, allowing the application of the characterisation algorithm. Section 6 defines and verifies a method for *matching* protocols from their characterisations. That is, given a protocol library, with each protocol annotated with its characterisation, an agent can determine which protocols in the library achieve a given goal. Section 7 proves that the complexity of this method is less than a depth-first search algorithm, and discusses experimental evaluation that confirms this.

2. SPECIFYING PROTOCOLS

In this section, we present a brief overview of a generic action language that can be used to define a state transition system. This language is defined to suit our purposes, but is general enough to ensure the results of the paper have wide applicability to other languages, such as that in [13]. We

Cite as: Characterising and Matching Iterative and Recursive Agent Interaction Protocols, T. Miller and P. McBurney, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.
Copyright © 2010, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

also discuss a logic for reasoning about outcomes of protocols specified in this language.

2.1 Action Languages

We assume that the language used to model protocols is an action language. The language should support atomic actions (messages sent between agents), sequencing, and choice. The language we describe also supports recursion.

Specifications manipulate a state, which is set of positive literals. These literals may represent the environment, or more likely, a shared set of commitments or norms. Each action (atomic protocol) is specified as a triple containing a precondition, an action identifier, and a postcondition. We represent this using the format:

$$\psi \xrightarrow{p} \psi',$$

in which ψ is the precondition, p the action identifier, and ψ' the postcondition. The action p can be performed only if ψ holds in the current state. The precondition and postconditions are propositions that are made up positive and negative literals, and conjunction of propositions. The action identifier is a positive literal. In the case of protocols, the action identifier specifies the message that is sent between agents, as well as the sender and receiver. We will use ϕ and ψ subscripted with numbers (e.g. ϕ_0) to represent propositions, p, q, and r to represent positive literals, $\neg p$ to represent the negative literal of p, and $\wedge$ as the conjunction operator, which represents a set of literals. The $\supseteq$ operator represents entailment; that is, $\phi \supseteq \psi$ holds if and only if the literals in ϕ are a superset of the literals in ψ.

Actions can be composed to make a compound protocols specifications. If α and β are both protocols, then the sequential composition $\alpha;\beta$ represents the protocol in which α executes fully, followed by β. The starting state of β is the end state of α. The choice $\alpha \cup \beta$ represents a non-deterministic choice between the two protocols, in which the participants must execute α or β, but not both. The iteration α^* represents zero or more iterations of α. The test operator, $\psi?$, represents the protocol that does nothing, but can execute only if ψ holds. We will use α, β, and γ to represent protocols.

Specifications consist of a set of definitions of the form $N(x) \mathrel{\hat{=}} \alpha$, in which N is a name, and x a variable list. Protocols can be referenced via their names in other protocols. We will omit the variables for brevity. Using this, α^* becomes shorthand for the name N, in which $N \mathrel{\hat{=}} \text{true}? \cup \alpha; N$.

2.2 Propositional Dynamic Logic

Propositional dynamic logic (PDL) can be used to reason about this language. PDL is used to represent the characterisations of protocols.

Given a protocol α, the proposition $[\alpha]\phi$ specifies that the proposition ϕ holds in every final state of the protocol α. This logic extends the propositional logic on which the action language is built by the addition of this operator.

We can define a second operator $\langle\alpha\rangle\phi$, which means that ϕ holds for *at least one* final state of α. This can be defined using the shorthand $\langle\alpha\rangle\phi$ iff $\neg[\alpha]\neg\phi$. That is, ϕ holds in some outcome if ϕ does not hold in all outcomes. This implies the law of excluded middle: $\neg\phi$ is true only if ϕ cannot be proved.

A deductive proof system for this logic can be defined using the following axioms:

$$
\begin{array}{lcll}
[\psi?]\phi & \leftrightarrow & \psi \rightarrow \phi & \\
[\psi \xrightarrow{p} \psi']\phi & \leftrightarrow & \psi \dagger \psi' \rightarrow \phi & \\
[\alpha;\beta]\phi & \leftrightarrow & [\alpha][\beta]\phi & \\
[\alpha \cup \beta]\phi & \leftrightarrow & [\alpha]\phi \wedge [\beta]\phi & \\
[N]\phi & \leftrightarrow & [\alpha]\phi & \text{if } N \mathrel{\hat{=}} \alpha
\end{array}
$$

and an inference rule for *modus ponens*. In the second axiom, the $\dagger$ operator defines *overriding*. That is $\psi \dagger \psi'$ defines the proposition that results by combining ψ and ψ' while removing all literals whose negation occurs in ψ'.

The final axiom above is sound, but incomplete for recursive definitions, because the unfolding of N will occur an infinite number of times. Instead, we use an *induction* inference rule, based on Scott induction [12]:

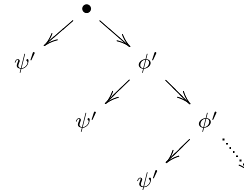
$$\frac{[N]\phi \vdash [\alpha]\phi}{[N]\phi} \quad \text{where } N \mathrel{\hat{=}} \alpha$$

This rule states that, to prove $[N]\phi$, first try to prove that $[\alpha]\phi$ is valid under the assumption that $[N]\phi$. If this is provable, then it must be that $[N]\phi$. The assumption that $[N]\phi$ is the inductive step, and prevents the proof from being infinite.

The idea behind this rule is straightforward to show. Consider the protocol

$$N \mathrel{\hat{=}} \psi \xrightarrow{p} \psi' \ \cup \ \phi \xrightarrow{q} \phi'; N.$$

This protocol can be summarised using the following infinite tree structure



in which the dashed arrow indicates that the unfolding of N continues infinitely. In this tree, we can see that the only terminating states are those at the nodes labelled ψ'. To prove $[N]\psi''$, we need to prove that ψ'' is satisfied in all terminating states. To prevent the infinite unfolding, we assume at the top level (the $\bullet$ node) that ψ'' holds in every end state below ϕ', and prove this for the remaining end states, of which there is only one: ψ'. One can see that this is sound, because if we unfold N, the only terminating states are the ψ' states. These states are able to be deduced directly from the *finite* definition of N, and the proof is finite.

Miller and McBurney [10] have demonstrated soundness and completeness of a proof system, including Scott induction, for a propositional dynamic logic over the $\mathcal{RASA}$ language.

3. DEFINITIONS

For the rest of this paper, the term *protocol* will refer to a protocol defined using the action language from Section 2.1 that does not have the *stuckness* property: a protocol is stuck if and only if, at any point during the execution of the protocol, the protocol has not terminated, and the specification prevents any action from occurring; for example, all of the preconditions of available actions do not hold.

The *weakest precondition* of a protocol is the weakest (or most general) proposition from which a protocol cannot become stuck. The *maximal postcondition* is the strongest

proposition that results from a protocol being executed under its weakest precondition.

A *goal* is a state of the world that an agent would like to bring about, or maintain. In this paper, we assume that a goal is represented as a proposition in the underlying language.

Given a goal, ϕ_G, and an initial state, ψ_I (the state of the world from which an agent wants to achieve the goal – generally the current state), a *weak matching* protocol is a protocol, α, that achieves the goal ϕ_G from the initial state ψ_I for at least one outcome. Formally:

$$\psi_I \rightarrow \langle\alpha\rangle\phi_G$$

A *strong matching* protocol is a protocol that achieves a goal for all outcomes, assuming that there exists at least one outcome[1]. Formally:

$$\psi_I \rightarrow [\alpha]\phi_G.$$

All strong matching protocols are also weak matching protocols. We distinguish between the two because an agent would want a protocol that achieves its goal for at least one outcome, but would likely prefer a protocol that achieves it for all outcomes.

To find all matches for a goal ϕ_G from the state ψ_I, the agent could simply use the PDL proof system discussed in Section 2.2. That is, for every protocol α, if the proof $\psi_I \rightarrow \langle\alpha\rangle\phi_G$ is successful, then α is a weak match. For a large protocol library, this is an expensive operation to perform each time an agent wants to find a protocol that achieves a certain goal. Instead, we summarise the preconditions and outcomes of the protocol using characterisations, and then search these. It is out hypothesis that this is more efficient than performing a PDL proof for each protocol.

4. CHARACTERISING PROTOCOLS

Characterisations for a protocol are derivable directly from the protocol specification itself. In this section, we present a straightforward algorithm for characterising protocols, which is based on symbolic execution. This algorithm can characterise any iterative protocol, and any recursively defined protocol that always terminates.

4.1 Representing Characterisations

Characterisations are represented as theorems in the logic presented in Section 2.2. For example, the characterisation

$$\psi_0 \rightarrow [\alpha]\phi_0$$

specifies that, if executed from any state that satisfies the weakest precondition ψ_0, the protocol α is guaranteed to achieve the outcome ϕ_0. Our goal is to characterise, for each protocol in a protocol library, not the outcomes that it achieves, but the outcomes of the *paths* of the protocol.

As an example, consider the protocol $P; (Q \cup R)$, in which $P \mathrel{\widehat=} p \xrightarrow{p} p'$, $Q \mathrel{\widehat=} \text{true} \xrightarrow{q} q'$, and $R \mathrel{\widehat=} \text{true} \xrightarrow{r} r'$. Figure 1 shows the abstract syntax tree of the protocol, and the characterisations that we derive. This protocol contains two paths: (1) P followed by Q; and (2) P followed by R.

Characterising paths, rather than entire protocols, is necessary, otherwise our characterisation is incomplete. Consider the protocol $A \cup B$, in which A and B are defined as

[1]This assumption is subsumed by our assumption that a protocol is free from stuckness.

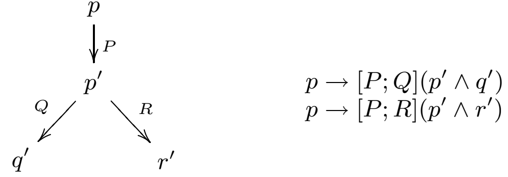


Figure 1: An abstract syntax tree for a protocol, and its characterisations.

$A \mathrel{\widehat=} a \xrightarrow{p} a'$ and $B \mathrel{\widehat=} b \xrightarrow{q} b'$. Our algorithm will generate the characterisations $a \rightarrow [A]a'$ and $b \rightarrow [B]b'$. If we were to merge these characterisations into one, we could write $a \vee b \rightarrow [A \cup B](a' \vee b')$. However, this loses vital information: that of the relationship between a and a', and the relationship between b and b'. That is, a' is only achievable from a state in which a holds. One can not infer this from the general characterisation, therefore, an agent could conclude that a' may achievable from b, which is not the case.

4.2 Characterising Outcomes

A straightforward symbolic execution algorithm is used to characterise paths.

Recall from Section 4.1, that protocols are characterised by the outcomes of their paths. To do this, each path in the protocol is symbolically executed, with the initial state being its weakest precondition. When the algorithm reaches the end of a path, the symbolic state that is left is the maximal postcondition of that path. The characterisation can be derived from this maximal postcondition, the weakest precondition, and the path.

The `characterise` function defines how to symbolically execute a protocol. As inputs, it takes a protocol and a proposition representing the current symbolic execution state. Initially, this state is the weakest precondition of the path that is being executed. `characterise` returns an characterisation of the form $\psi_I \rightarrow [\alpha]\phi_G$, in which ψ_I is the weakest precondition, and ϕ_G the maximal postcondition.

```
characterise(ψ0 -p-> ψ0', φ0) ≙
    if φ0 ∧ ψ0 ⊇ false then return {}
    else return {φ0 ∧ ψ0 → [ψ0 -p-> ψ0'](φ0 ∧ ψ0) † (ψ0' ∧ p)}

characterise(α; β, φ0) ≙
    S := characterise(α, φ0)
    for φ1 → [α]ψ1 ∈ S do
        S' := characterise(β, ψ1)
        for φ2 → [β]ψ2 ∈ S' do
            R := R ∪ {φ2 † φ1 → [α; β]ψ2}
    return R

characterise(α ∪ β, φ0) ≙
    return characterise(α, φ0) ∪ characterise(β, φ0)

characterise(ψ0?, φ0) ≙
    if φ0 ∧ ψ0 ⊇ false then return {}
    else return{φ0 ∧ ψ0 → [ψ0?](φ0 ∧ ψ0)}

characterise(N, φ0) ≙
    if N ≙ α then return characterise(α, φ0)

characterise(α*, φ0) ≙
    return characterise(true?, φ0) ∪ characterise(α, φ0)
```

With the exception of atomic and iterative protocols, the

characterisation algorithm is straightforward. Calculating the characterisation for an atomic protocol is as follows: if the symbolic state ϕ_0 is compatible with the precondition ψ_0 – that is, their conjunction is satisfiable (does not entail false) – then the precondition is their conjunction. If they are incompatible, no characterisations are generated because this represents a termination condition for the protocol. The maximal postcondition is simply the overriding (denoted using the binary operator †) of the symbolic state and precondition with the postcondition.

The iterative operator defines a protocol of infinite size: α^* can iterate of α an unbounded number of times. Unfolding this an infinite number of times is not possible. Instead, we calculate the fixed points of the protocol's end states. First, we break the protocol into two cases: the case in which zero iterations occur, and the case in which one or more iterations occur. The zero iteration case is equivalent to true?. The postcondition for the one-or-more iterative case is the just the postcondition of one iteration, and similarly for the weakest precondition (see Theorem 4.2 later in this section). Therefore, to characterise this, we simply need to characterise α.

4.3 Termination, Soundness, and Completeness

The `characterise` function defined in this section terminates when applied to any protocol that terminates, provided that the underlying proposition system is over a finite domain. It will not terminate for a non-terminating protocol. The algorithm is also sound and complete.

THEOREM 4.1. *For terminating protocol α, and finite proposition domain, the function* `characterise` *will terminate under the protocol's weakest precondition.*

PROOF. For non-recursive protocols, recursive calls to `characterise` are made only on sub-protocols, and name references will only be unfold once (because there is no recursion in the protocol). Therefore the algorithm terminates.

For recursive protocols that terminate, there are a finite number of terminating paths, which are each of a finite length. The `characterise` algorithm uses weakest preconditions as the initial states, of which there are a finite number. With a finite number of paths executed on a finite number of inputs, the algorithm will terminate. □

For each definition of `characterise`, soundness and completeness follow directly from the definition of the protocol specification language, with the exception of iterative protocols. We prove the soundness of the iterative case here. Characterisation of iterative protocols relies on the following theorem.

THEOREM 4.2. *If ψ_0 is the weakest precondition of α, then the following holds: $\psi_0 \rightarrow [\alpha]\phi$ iff $\psi_0 \rightarrow [\alpha;\alpha^*]\phi$*

PROOF. The right-to-left case holds by noting that all outcomes in α are a subset of the outcomes in $\alpha;\alpha^*$. For the left-to-right case, if ψ_0 holds, then 1 iteration of α will result in ϕ. At this point, one of two properties hold: $\neg(\phi \rightarrow \psi_0)$, or $\phi \rightarrow \psi_0$; that is, either the precondition holds, or it does not. If the former, then α cannot iterate again, because ψ_0 is its weakest precondition, and the precondition is not satisfied. Therefore, the postcondition is ϕ. If the latter, then α can either terminate, leaving the postcondition as ϕ, or it can iterate again. If it iterates again, we know that the result will be ϕ, because ψ_0 holds, and we know from the premise that α preserves ϕ under its weakest precondition. Therefore, the resulting postcondition is ϕ. Applying this argument inductively, we see that the strongest postcondition of $\alpha;\alpha^*$ is ϕ. □

For the case of α^* (Theorem 4.2 considers only $\alpha;\alpha^*$), we note that the zero iteration case is equivalent to true?, which can always be executed and never changes the state.

5. CHARACTERISATION OF RECURSIVE PROTOCOLS

The difficulty of characterising recursive protocols is the infinite unfolding of name references. The approach for dealing with this in Section 4 is of limited use, because it cannot handle recursive, non-terminating protocols, and it requires the unfolding of name references an unknown number of times, which can quickly exhaust memory.

The approach taken in this section is to characterise recursive protocols by re-writing them into a form that removes the recursion; that is, removing name references, such that the abstract syntax tree is finite. For example, consider a protocol definition of the form $N \mathrel{\widehat{=}} \epsilon \cup (\alpha;N)$. The definition of N is equivalent to α^* (directly from the definition of *). The `characterise` algorithm can then be used to characterise N.

DEFINITION 5.1. *Choice Normal Form.* We say that a protocol is in *choice normal form* if and only if it is a choice between one or more protocols that do not contain other choice protocols. That is, for a protocol $\alpha \cup \ldots \cup \alpha_n$, each of α to α_n does not contain a choice. This is analogous to disjunctive normal form in Boolean logic.

THEOREM 5.1. *Any protocol can be reduced to choice normal form.*

PROOF. Any protocol that fits our definition from Section 2 can be reduced to choice normal form by using the property of sequential composition distributing over choice:

$$\alpha;(\beta \cup \gamma) \quad \equiv \quad (\alpha;\beta) \cup (\alpha;\gamma),$$

and commutative and associate rules. Names are not unfolded when re-writing into choice normal form. Iteration operators are pushed inwards over choice protocols using

$$(\alpha \cup \beta)^* \quad \equiv \quad \alpha^*;(\beta;\alpha^*)^*,$$

which is an axiom direct from Kleene algebras [6]. □

DEFINITION 5.2. *Linear choice normal form.* We say that a protocol definition, $N \mathrel{\widehat{=}} \alpha \cup \ldots \cup \alpha_n$ is in *linear choice normal form* if and only if $\alpha \cup \ldots \cup \alpha_n$ is in choice normal form, and *both* of the following hold:

1. for all protocols in $\{\alpha, \ldots, \alpha_n\}$, there is at most, one reference to N; and
2. for at least one protocol in $\{\alpha, \ldots, \alpha_n\}$, there is no reference to N.

This means that 1) there exists only one recursive call in each branch of that protocol; and 2) at least one branch does not contain a recursive call, and is therefore guaranteed to terminate.

It is our theorem that any protocol in linear choice normal form can be characterised by removing the recursive calls, and replacing them with equivalent non-recursive definitions.

THEOREM 5.2. *For protocols of the format $N \mathrel{\widehat{=}} \alpha \cup \beta; N; \gamma$, in which the definition is in linear choice normal form, the following formula holds:*

$$[N]\phi \quad \textit{iff} \quad [\alpha \cup (\beta^+; \alpha; \gamma^+)]\phi,$$

in which $^+$ defines non-empty iteration; that is, $\alpha^+ \equiv \alpha; \alpha^*$.

PROOF. From PDL, we know that a program of the form $\alpha \cup (\beta; N; \gamma)$ is equivalent to a program of the form $\beta^n; \alpha; \gamma^n$, which represents a set of programs: for all $n \geq 0$, execute β exactly n times, then execute α, and then execute γ exactly n times. It is known that such a program is not expressible in PDL [5].

However, for the purpose of characterisation, we can bring protocols of such form back into the class of PDL programs by using the results on iterative protocols from Section 4.

If we divide protocols of the form $\beta^n; \alpha; \gamma^n$ into the cases in which $n = 0$ and $n > 0$, then we get the following:

$$\alpha \cup (\beta^{n+1}; \alpha; \gamma^{n+1}),$$

assuming that $\alpha^0 \equiv \epsilon$, and using the property $\epsilon; \alpha \equiv \alpha \equiv \alpha; \epsilon$.

From Theorem 4.2, we know that characterising α^+ is equivalent to characterising α, and that the number of iterations of α is not relevant to its strongest postcondition. If this is the case, then the number of iterations in the expression β^{n+1} is not relevant either, so we can characterise β^{n+1} by characterising β, and similarly for γ^{n+1}. □

DISCUSSION POINT. An obvious question is how useful the above theorem is, because it is applicable only to a restricted class of protocols.

All protocols can be re-written into choice normal form (Theorem 5.2). This leaves us with the two classes that do not fit into Definition 5.2, and cannot be re-written: 1) those protocols in which one of the branches is non-linear, for example, $N; N; \alpha$; and 2) those protocols in which ALL paths fail to terminate.

The outcomes of protocols in class 2 are unable to be characterised in any case because all paths fail to, and therefore have no outcomes. This does not imply that such protocols are worthless, but just that other types of characterisation may be required. That leaves us with protocols in class 1: non-linear recursive branches. Our theorem does not apply to these protocols, and these will be considered in future work. However, we have not identified a protocol with this property in the multi-agent systems literature, which leads us to believe our theorem has wide applicability.

We note that, while Theorem 5.2 assumes a protocol definition of the form $N \mathrel{\widehat{=}} \alpha \cup (\beta; N; \gamma)$, any protocol in linear choice normal form can be expressed in this way. That is, α may be a choice protocol itself that contains a reference to N, however, as long as one of the branches of that protocol terminates, this is still in linear choice normal form, and Theorem 5.2 still holds. The protocols α, β and γ are not restricted to be actions; they can be compound protocols.

The recursive branch, $\beta; N; \gamma$, is expressive enough to handle head or tail recursion. For example, to handle tail recursion, simply substitute ϵ in for β and note that $\epsilon; \alpha \equiv \alpha$.

6. MATCHING PROTOCOLS VIA CHARACTERISATIONS

Recall the definitions of weak and strong matches from Section 3. In this section, we present methods for identifying weak and strong matching protocols using their characterisations.

6.1 Strong Matching

The relation for a strong match is more straightforward than for a weak match. To prove that a protocol, α, is a strong match for goal ϕ_G and initial state ψ_I, one must prove the following:

$$\forall a \in chrs(\alpha) \bullet \texttt{strong_match}(a, \psi_I, \phi_G)$$

in which `strong_match` is defined as the relation

$$\texttt{strong_match}(\psi_0 \rightarrow [\alpha]\phi_0,\ \psi_I,\ \phi_G) \leftrightarrow \\ \psi_I \supseteq \psi_0 \ \rightarrow\ \phi_0 \supseteq \phi_G$$

and *chrs* returns the set of all characterisations for a protocol. The above states that, given a set of outcome characterisations for a protocol, if for all of these outcomes, when the initial state satisfies the precondition, the postcondition satisfies the goal state, then we have a strong match.

The implication ($\rightarrow$) in this definition states that we only care about the paths in which the precondition is satisfied because the other paths cannot be executed, so are not relevant to the agent's ability to achieve the goal from the initial state.

THEOREM 6.1. *The strong matching method is sound and complete.*

PROOF. For soundness, if, for all paths (characterisations), the weakest precondition is satisfied by the initial state, and the outcome satisfy the goal, then it must be that all outcomes of all paths satisfy the goal. To show completeness, we note that, for any true formula, $\phi_I \rightarrow [\alpha]\phi_G$, it holds that ϕ_I must satisfy the precondition, ψ_0, and that the maximal postcondition, ϕ_0, must satisfy ϕ_G. □

6.2 Weak Matching

Recall from Section 2.2 that $\langle\alpha\rangle\phi$ is defined as shorthand for $\neg[\alpha]\neg\phi$. Using this symmetry, we define weak matching as:

$$\neg\forall a \in chrs(\alpha) \bullet \texttt{strong_match}(a, \psi_I, \neg\phi_G).$$

This is equivalent to:

$$\begin{aligned} & \exists a \in chrs(\alpha) \bullet \neg(\psi_I \supseteq \psi_0 \ \rightarrow\ \phi_0 \supseteq \neg\phi_G) \\ \equiv\ & \exists a \in chrs(\alpha) \bullet \psi_I \supseteq \psi_0 \wedge \phi_0 \supseteq \phi_G \end{aligned}$$

It is straightforward to see that this captures our requirements for a weak match: at least one characterisation (path in the protocol) can be executed and satisfies the goal.

7. EVALUATION

In this section, we prove that the time complexity of our method is less than using a depth-first search algorithm to match a goal. First, we perform a theoretical complexity analysis that proves this. Second, we perform an experimental analysis to confirm the complexity analysis.

7.1 Theoretical Analysis

To study the complexity, we treat protocols as trees, as in Figure 1. Similarly, we treat characterisations as trees, but only of depth 1, with each characterisation forming one branch. Figure 2(a) illustrates the tree format for the protocol $\phi_0 \xrightarrow{m_1} \phi_1; (\phi_1 \xrightarrow{m_2} \phi_2 \ \cup\ \phi_1 \xrightarrow{m_3} \phi_3)$ and its characterisations.

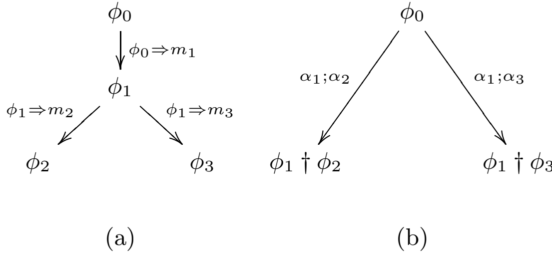


Figure 2: (a) A tree representation of a protocol; and (b) A tree representation of the characterisations of the protocol. For characterisations, the tree will always be of depth 1.

A protocol is mapped to a tree by taking the abstract syntax tree (AST) of the structure. Nodes represent states, and edges represent the message templates. To map a protocol to its AST, the weakest precondition of the protocol forms the root node, choice protocols are represented by branches in the tree, and sequential compositions are represented as concatenation of paths down the depth of the tree.

A set of characterisations is mapped to a forest structure by mapping each characterisation to a single tree with two nodes and an edge: the root node being the precondition of the annotation, the edge being the protocol, and the leaf node being the most general outcome. Several trees with the same root node can form a tree with one root node. Figure 2(b) illustrates this.

One can see that the number of characterisation is equivalent to the number of paths throughout the protocol, therefore, the protocol AST and the characterisation tree both have the same number of end nodes. As the length of the paths in the protocol increase, the height of the protocol AST increases, while the height of the annotation tree remains the same: exactly one.

Using these tree structures, we can now simplify our problem to the analysis of a depth-first search algorithm. The problem is slightly different to the standard analysis of depth-first search because the goal states must be located at the leaf nodes.

We consider the expensive operations in this search to be the following:

1. the operation that checks whether a goal is satisfied by an end state or a precondition is satisfied by the start state (entailment); and
2. the operation that traverses a vertex in the tree (e.g. applying an axiom in a theorem prover). The end state of a protocol cannot be assessed until the entire path has been evaluated, so the total cost of this for an entire tree is the number of edges, e.

The total number of paths through a protocol is denoted n, and this is equivalent to the number of end states and the number of characterisations.

In this section, we compare our matching method with that of a depth-first search (DFS) algorithm. We choose a DFS because this is an alternative way to match a goal, and the complexity would be similar to that of using a theorem prover or model checking to find a matching protocol, as both theorem provers and model checkers would need to assess the same edges nodes as a DFS algorithm.

Case		Time complexity: DFS	Time complexity: Matching Method
Strong matching	Average	$\frac{e+2n}{n-m+1}$	$\frac{2n}{n-m+1}$
	Worst	$e+2n$	$2n$
Weak matching	Average	$\frac{e+2n}{m+1}$	$\frac{2n}{m+1}$
	Worst	$e+2n$	$2n$

Table 1: Average and worst-case complexities for matching a protocol again a goal using DFS and our matching method.

7.1.1 Finding Strong and Weak Matches

The time complexities of finding strong and weak matches using both methods are shown in Table 1. In this section, we prove these complexities.

THEOREM 7.1. *The worst-case complexities of both strong and weak matching are* $e+2n$ *for DFS and* $2n$ *for our matching method.*

PROOF. For either method, the worst case complexity for weak and strong matching are equivalent. In the worst case, the entire tree will have to be searched, which has complexity $e+2n$, in which e is the number of vertices in the AST, and n is the number of paths in the tree. The expression $2n$ is derived because we must check the precondition of each path, as well as the outcome, so for each of the n paths, two entailment operations are required.

The complexity of a complete traversal of a characterisation tree is $2n$, in which n is the number of characterisations. Again, we must check both the precondition and outcome of all paths, but in this case, the vertices are not relevant, because all are of a constant height (one). □

With the height of every path in a characterisation tree being one, we may consider not using a tree at all, and replacing this with a list of characterisations. The advantage of using a tree is that some characterisations will share preconditions. Each precondition is evaluated once, so we can avoid repeated evaluations by using a tree. However, to simplify our analysis, we do not consider this case, and instead assume a list of characterisations.

From an asymptotic view, the worst case time complexity for matching a protocol takes time complexity $O(e+n)$ for DFS, and $O(n)$ using our matching method.

THEOREM 7.2. *The average case complexity for weak matching is* $\frac{e+2n}{m+1}$ *for DFS, and* $\frac{2n}{m+1}$ *for our matching method; and the average case for strong matching is* $\frac{e+2n}{n-m+1}$ *for DFS, and* $\frac{2n}{n-m+1}$ *for our matching method.*

PROOF. The average case complexity is slightly more complicated than the worst case because we have to consider that the search does not have to assess the entire tree. In the case of a weak match, the search will terminate once a match is found. In the case of a strong match, the search will terminate once a single path is shown not to hold.

If only one end state satisfies the goal, then the average complexity is $\frac{e+2n}{2}$ for DFS and n for matching, because we

would have to search half of the tree on average to find a match. For multiple matches, the complexity analysis is less straightforward.

Assume that m end states satisfy the goal, where $0 \leq m \leq n$. For any individual end state, the probability of that end state satisfying the goal is $\frac{m}{n}$. For a weak match, the average number of nodes we will have to search to find a match is therefore $\frac{1}{m+1/n+1}$, which is just $\frac{n+1}{m+1}$. The total number that need to be assessed is $\frac{n+1/m+1}{n+1}$, or $\frac{1}{m+1}$. Therefore, the time complexity is $\frac{e+2n}{m+1}$ for DFS, and $\frac{2n}{m+1}$ for matching. This is consistent with the worst-case complexity, which is when $m = 0$.

For a strong match, the probability of a path being *non*-matching is $\frac{n-m}{n}$. Calculating the average case for a strong match is the same as for a weak match, except we substitute $n - m$ for m. Therefore, we obtain a time complexity of $\frac{e+2n}{n-m+1}$ for DFS, and $\frac{2n}{n-m+1}$ for matching. Again, this is consistent with the worst-case complexity when $m = n$. □

From an asymptotic view, we know that $m \leq n$, so the worst case time complexity for matching a protocol takes time complexity $O(e + n)$ for a DFS, and $O(n)$ using our matching method.

The results from Theorems 7.1 and 7.2, tell us the asymptotic complexities of both methods increase in linear time with the number of end states, or exponentially related to the size of the protocol (for example, given an average height, h, and average branching factor, b, we have that $n = b^h$, so n increases exponentially).

7.1.2 Deriving Characterisations

Using DFS to match protocols against a goal does not require any pre-processing, whereas our matching method first requires us to characterise the protocol. The time complexity for this is straightforward: we have to symbolically execute the entire protocol, so it is the same as the worst-case time complexity of DFS: $e + 2n$. However, we note that the expensive operations for this algorithm are different than for matching; instead of performing an entailment to see if a goal holds at an end state, we calculate the strongest postcondition, and propagate this information back up the tree. As a result, we may wish to use different variables to distinguish this; for example, $f + 2c$, in which f is the number of edges, and c the number of end states.

To assess whether our method of characterisations and matching is useful, we have to assess the impact of characterisations. While characterisations have to be calculated only once for every protocol, matching may be done an indefinite number of times. However, given that the characterisation algorithm is the worst-case coverage of the protocol tree, as is DFS, it is only worth using our method if we expect matching to be performed, on average, more than once per protocol. This assumes that the complexity of the various expensive operations is the same; that is, $e \approx f$ and $c \approx n$.

One additional advantage of our method is that the characterisation can be performed "off line".

7.2 Experimental Evaluation

The complexity results from Section 7.1 compare the theoretical complexity of our method with that of DFS. However, the average case complexity is based on the number of end states that match in the protocol. Furthermore, the complexity is simplified by treating an entire path as one calculation, whereas in practice, as path is only evaluated if the precondition holds true. In this section, we discuss an experimental evaluation that compares our method with a DFS algorithm.

7.2.1 The Experimental System

The characterisation algorithm from Section 4, the matching method from Section 6, and a PDL proof system (Section 2.2) have been implemented using Prolog. For the underlying language, we have used the CLP bounds solver, an integer constraint solver with variables, although we aimed for the implementation to be general enough to support most constraint solvers.

7.2.2 The Experiment

To experiment with the system, we implemented a random protocol generator. First, a set of constraints is generated, then, a protocol of a random height is generated, using the constraints as preconditions and postconditions of atomic protocols. The protocol is then characterised. Using these characterisations, the two methods were applied (DFS and our matching method), while recording the time complexity. We varied the height of the protocols incrementally from 1 to 20. In total, we generated 100 random protocols of each height.

The independent variable in the experiment is the method being used to match protocols: a DFS algorithm, or our matching method. The dependent variables measured were: 1) the average time taken for each method to check for strong and weak matches; 2) the average number of inferences performed by Prolog by each method for strong and weak matches; 3) the average number of entailment operations performed for strong and weak matches. The constants were the protocols on which the experiments were performed, and the initial and goal states.

A strong match was simulating by searching for the goal true, while a weak match was simulated by searching for a random goal that was present in the set of characterisations.

7.2.3 Results

Figure 3 shows the execution time for the three algorithms: depth-first search for strong matching, our matching method for strong matching, and our characterisation method. The x-axis is the number of end states in the protocol (n). The graphs for number of entailment operations and number of inferences are similar, so are omitted. The results for weak matching are also similar, except the weak matches take less for DFS and our matching method, and much the same for characterisation.

One can see from these graph that our matching method is a more efficient way of matching compared to depth-first search, and the time difference between the two increases linearly with the number of end states. This confirms our theoretical analysis from Section 7.1.

The other interesting thing to note is that, for the randomly generated protocol, the relations discussed at the end of Section 7.1 that $e \approx f$ and $c \approx n$ do not hold. For these protocols, it appears that the actual complexity of characterisation is approximately 4 times that of the depth-first search, so a protocol would need to be matched at least 4 times for the characterisation to pay off.

8. RELATED AND FUTURE WORK

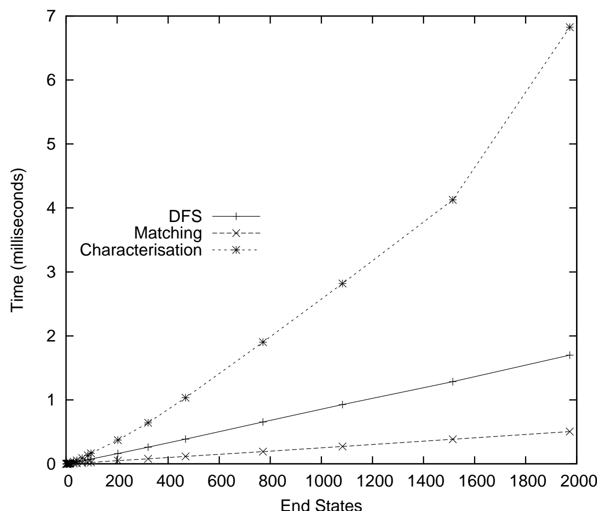


Figure 3: Execution time in milliseconds for strong matching using the three algorithms: depth-first search, our matching method, and our characterisation algorithm; plotted against the number of end states

In this paper, we have presented a method for characterising protocols, and a method for matching a protocol against a given goal, using those characterisations. We considered iterative and terminating recursive protocols, and presented an algorithm for characterising these. We prove that a restricted subset of recursive protocols (including some non-terminating protocols) can be re-written into an equivalent non-recursive form using iteration, so that the characterisation algorithm could be used. Theoretical and complexity analysis demonstrates that, for matching a protocol against a goal, our matching method is more efficient than a depth-first search, although the initial characterisation is more expensive.

Bussman et al. [1] discuss characterisation and matching of agent interaction protocols. Their approach characterises properties other than outcomes, such as number of participants and number of joint commitments. The matching is performed by developers at design time, rather than by agents at runtime. Clement and Durfee [2] present a method for characterising precondition and postconditions of hierarchical task network plans. However, they do not discuss characterising iterative or recursive plans. Planning and temporal projection methods [4, 11] are similar to our techniques, however, they aim to construct a plan that achieves a goal from a specific start state, whereas we aim to characterise the preconditions and postcondition of a given protocol.

Our concept of protocol libraries is similar to plan libraries, such as those found in many BDI and goal-oriented agent framework, such the Procedural Reasoning System [3]. However, plan preconditions and postconditions are typically specified manually, rather than calculated, and a plan matches if the postcondition unifies with the goal, rather than entailing it. However, we believe our characterisation method can be easily adapted to automatically calculate plan preconditions and postconditions.

The research reported here is an important step towards achievement of the vision of intelligent, goal-directed software agents able to decide at run-time which communications protocols to use. In other work, we are investigating how protocol libraries can be stored and searched for efficient protocol matching.

9. REFERENCES

[1] S. Bussmann, N. Jennings, and M. Wooldridge. Re-use of interaction protocols for agent-based control applications. In *Agent-Oriented Software Engineering III*, volume 2585 of *LNCS*, pages 73–87, 2002.

[2] B. Clement and E. Durfee. Theory for coordinating concurrent hierarchical planning agents using summary information. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pages 495–502. AAAI, 1999.

[3] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 677–682, 1987.

[4] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[5] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[6] D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *Proceedings of Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 26–47. Springer, 1990.

[7] J. McGinnis and T. Miller. Amongst first-class protocols. In A. Artikis, G. O'Hare, K. Stathis, and G. Vouros, editors, *Engineering Societies in the Agents World VIII*, volume 4995 of *LNAI*, pages 208–223, 2007.

[8] T. Miller and P. McBurney. Using constraints and process algebra for specification of first-class agent interaction protocols. In G. O'Hare, A. Ricci, M. O'Grady, and O. Dikenelli, editors, *Engineering Societies in the Agents World VII*, volume 4457 of *LNAI*, pages 245–264, 2007.

[9] T. Miller and P. McBurney. Annotation and matching first-class agent interaction protocols. In L. Padgham, D. Parkes, J. P. Mueller, and S. Parsons, editors, *Proceedings of the Seventh International Conference on Autonomous Agents and Multi-Agent Systems*, pages 805–812, Estoril, Portugal, May 2008.

[10] T. Miller and P. McBurney. Propositional dynamic logic for reasoning about first-class agent interaction protocols. *Computational Intelligence*, 2010. *(Forthcoming)*.

[11] H. Prendinger and G. Schurz. Reasoning about action and change: A dynamic logic approach. *Journal of Logic, Language, and Information*, 5(2):209–245, 1996.

[12] D. Scott and J.W. de Bakker. A theory of programs: Notes of an IBM Vienna seminar, 1969. Unpublished.

[13] P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and AI*, 42(1–3):227–253, 2004.