Efficient storage and retrieval in agent protocol libraries using subsumption hierarchies

Tim Miller
Department of Computer Science and Software Engineering, University of Melbourne, Australia,
tmiller@unimelb.edu.au

Peter McBurney
Department of Informatics, King's College London, U.K., peter.mcburney@kcl.ac.uk

Abstract—To be truly adaptive in dynamic environments, agents must be able to interact in ways that were not conceived at design time. One way to achieve this is to use executable protocol specifications, in which participants can share a protocol specification at runtime, and interact using the specified rules. In such a model, it is necessary that an agent can quickly and correctly locate a protocol that achieves its goals. Techniques such as model checking or theorem proving can be used to assess whether a protocol achieves a goal; however, for resource bound agents, this approach may be inefficient. Building on previous work, we present a method for characterising and matching protocols, and for storing, structuring, and searching these characterisations in protocol libraries. Libraries are represented using subsumption hierarchies, in which the vertices represent characterisations of protocols, and edges record a relation between two characterisations if one characterisation subsumes the other; that is, the subsumed protocol achieves all of the same outcomes. In addition, information about the domain of characterisations in the graph is propagated up the subsumption hierarchy, which prevents searching of irrelevant parts of the library. An experimental analysis demonstrates that this approach has an average-case time complexity of $O(n)$, which is the same as a standard linear search, however, the measured time is significantly lower. Despite this, the cost of structuring the library has an average-case time complexity $O(n^2)$, so the ratio of searches to insertions must be high to gain a benefit.

1 INTRODUCTION

In the distributed environments of multi-agent systems, interaction protocols are seen as a promising approach to coordination in multi-agent systems. However, many practitioners view interaction protocols as rigid specifications that are defined a priori, with agents being hard-coded to follow the protocol rules. 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 — therefore requiring agent code be changed 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, therefore 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.

An important corollary of these points is that the protocol is internalised within the individual agents. There is no possibility to communicate, inspect or verify the protocols by the agent or others, not to mention a duplication of effort for each agent engineer as each agent must be encoded with the same protocol.

There are a range of solutions to this problem, such as not using defined interaction protocols, instead designing agents to interact unconstrained. Planning and reasoning mechanisms can be used to determine what information needs to be communicated, and when. However, this can be considered a longer-term goal of adaptive agents research. Furthermore, equipping adaptive agents with such mechanisms assumes they have the resources to perform such planning, which some do not. The BDI agent paradigm was designed to deal with such situations. Interaction protocols and BDI agents are highly compatible, as interaction protocols provide context for participating parties, which can be implemented in BDI plans, thereby, reduce thing the complexity required to communicate with other agents.

To bridge the gap between hard-coded protocols and highly adaptive agents, some researchers and practitioners advocate the treatment of agent interaction protocols as first-class computational entities, allowing agents to select, reference, share, compose, invoke and inspect protocols at runtime [13], [20], [24], [29], [32], [33]. Such an approach would enable a higher level of adaptation than hard-coded protocol, by allowing agents to assess which protocols achieve their goals, and to learn the rules and effect of new protocols at runtime, while also providing context and rules against which the agent can reason.

A major goal of research into first-class protocols is for agents to maintain a library of interaction protocols, and to be able to select the protocol that best suits the goals that it wants to achieve at a given time and in a given environment. For this, agents must be able to quickly and correctly determine the outcomes that can
result for an interaction protocol, and compare protocols in their library. Such libraries are particularly applicable in dynamic domains, such as systems of competing marketplaces, where both the market participants and the market owners change their behaviours in response to, and in anticipation of, each other’s behaviours, as if they were co-evolving populations.

In previous work [17], [19], we presented methods to characterise the possible outcomes of a first-class protocol, so that an agent does not have to calculate the outcomes each time it is trying to find a suitable protocol, and to match a protocol that achieves a goal against these characterisations. In that work, we assume that agents have access to a library of protocols, but we made no assumptions about the structure of such libraries, nor how the libraries are searched. The characterisation and matching method are summarised in Section 4.

In this paper, we improve that method by outlining a method for structuring and searching a library of protocols using subsumption hierarchies [30] — directed graphs that record the relationship between characterisations. By structuring the characterisations as a subsumption hierarchy, rather than, for example, a list, we take advantage of the subsumption relationship to reduce the search space for particular searches. In Section 5, we present an informed depth-first search algorithm for locating matching protocols in a hierarchy, and in Section 6, we present an experiment comparing the complexity of the search algorithm to an uninformed linear search on an unstructured library. The structuring and searching approach is general enough to be applied to other aspects of multi-agent systems, such as plan libraries, but our motivation is to improve adaptive agent interaction.

The results of the experimental analysis indicate that our approach has an average-case time complexity of \( O(n) \), which is the same as a standard linear search but the measured time is significantly lower. However, this is offset by the cost of structuring the library as a subsumption hierarchy, which has an average-case time complexity \( O(n^2) \), implying that the ratio of searches to insertions must be high to gain a benefit.

Section 7 of the paper discusses the most closely related research to ours, and Section 8 discusses future work and concludes the paper.

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, and is flexible enough to specify executable interaction protocols. 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 [36]. We also discuss a logic for reasoning about outcomes of protocols specified in this language, which is used to represent characterisations.

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 a set of ground first-order terms. We assume a first-order logic with quantifiers is used to represent the state, as well as an underlying communication language. That is, a propositional logic extended with terms that contain unground variables. The logic contains terms, conjunction \((\land)\), negation as failure \((\neg)\), and entailment \((\implies)\). We will use subscripted \( \phi \) and \( \psi \) (e.g. \( \phi_0 \)) to represent formula in this logic, and \( p \), \( q \), and \( r \) to represent terms.

In our projects that use executable protocol specifications [16], [18], [20], [19], we have used constraint languages as the underlying language. However, in this paper, we use a a first-order language to generalise the results. This first-order language is sufficient to generalise the results to BDI programming languages, logic programming languages, and action formalisms such as the situation calculus [28] and event calculus [11].

The state under manipulation 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 \( \psi \) is the precondition, \( p \) the action identifier, and \( \psi' \) the postcondition. The action \( p \) can be performed only if \( \psi \) holds in the current state. The precondition and postconditions may contain variables, however, at execution time, all variables become ground. The action identifier is a positive term. In the case of protocols, the action identifier specifies the message that is sent between agents, as well as the sender and receiver.

As well as atomic actions, actions can be composed to make a compound protocols specifications using the following grammar:

\[
\alpha ::= \phi \xrightarrow{P} \phi \mid \alpha;\alpha \mid \alpha \cup \alpha \mid \phi? \mid \alpha^* \mid N(x).
\]

in which \( \phi \) represents predicates.

If \( \alpha \) and \( \beta \) are both protocols, then the sequential composition \( \alpha;\beta \) represents the protocol in which \( \alpha \) executes fully, followed by \( \beta \). The starting state of \( \beta \) is the end state of \( \alpha \). The choice \( \alpha \cup \beta \) represents a non-deterministic choice between the two protocols, in which the participants must execute either \( \alpha \) or \( \beta \), but not both. The iteration \( \alpha^* \) represents zero or more iterations of \( \alpha \). The test operator, \( \phi? \), represents the protocol that does nothing, but can execute only if \( \psi \) holds. The protocol \( \alpha^* \) represents zero or more iterations of \( \alpha \). We will use \( \alpha, \beta, \gamma \) as variable names representing protocols.

Specifications consist of a set of definitions of the form \( N(x) \equiv \alpha \), 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.

The $\dagger$ operator is used to define the semantics of language, specifically, to define how state updates occur when messages are sent. The operators is defined as:

\[
M \dagger p = M \cup \{p\} \\
M \dagger \neg p = M \setminus \{p\} \\
M \dagger (\psi \land \psi') = M \dagger \psi \cup M \dagger \psi'
\]

in which $M$ represents a set of terms, $p$ is a ground term, and overriding this adds/removes positive/negative terms respectively, similar to asserting and retracting terms to/from a constraint store. To override with a conjunction, both operands are evaluated simultaneously, so asserting a positive term and its negation will add the term to the store.

Using the above definitions, we can now define a denotational semantics for the action language. In this paper, we are concerned only with the final states, or outcomes. Assuming that $M$ is a model (a set of ground terms) representing the initial state, the semantics of the language are as follows:

\[
\begin{align*}
\llbracket \psi \vee \psi' \rrbracket (M) &\equiv \{M' | M \supseteq \psi \land M' \supseteq M \dagger (\psi' \land \psi)\} \\
\llbracket \alpha; \beta \rrbracket (M) &\equiv \{M'' | M' \in [\alpha] (M) \land M'' \in [\beta] (M')\} \\
\llbracket \alpha \lor \beta \rrbracket (M) &\equiv \llbracket \alpha \rrbracket (M) \lor \llbracket \beta \rrbracket (M) \\
\llbracket \psi? \rrbracket (M) &\equiv \{M | M \supseteq \psi\}
\end{align*}
\]

We discuss these briefly. Actions override the state with new postconditions, but can only be performed if their precondition holds. Sequential composition $\alpha; \beta$ uses the end state of protocol $\alpha$ as the initial state of protocol $\beta$. The set of end states of a choice protocol is simply the union of the end states of both operands. The test operator leaves the state unchanged, but is only applicable if the term in the test holds in the model.

Name references are less straightforward, due to the possibility of recursive definitions. To handle this, we use a second parameter, $e$, representing an environment that maps names to their semantic definitions. We split the definition into two cases: 1) when the name is in the environment, in which case we return the semantics for that name; and 2) when the name is not in the environment, in which case the semantics are added to the environment:

\[
\llbracket N \rrbracket (M, e) \equiv e(N) \quad \text{if} \quad N \in \text{dom}(e) \\
\mu F \quad \text{if} \quad N \not\in \text{dom}(e)
\]

where $F(H) = [\alpha]([e \dagger \{N \rightarrow H\}]$ and $\mu F$ is the least fixpoint of a function $F$ with respect to the partial order $\subseteq$; that is $\mu F = F(\mu F)$. So, in the second case, the least fixpoint is defined as the semantics, and any subsequent calls to $N$ in the protocol will return this via the first case.

Finally, the iteration operator is defined as a shorthand operator:

\[
\llbracket \alpha^* \rrbracket (M, e) \equiv \llbracket N \rrbracket (M, e),
\]

in which $N \equiv \text{true?} \lor \alpha; N$. That is, $\alpha^*$ is equivalent to a choice between terminating (true?) or executing $\alpha$ once, and then making this choice again.

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 $\alpha$, the proposition $[\alpha]\phi$ specifies that the proposition $\phi$ holds in every final state of the protocol $\alpha$. This logic extends the propositional logic on which the action language is built by the addition of this operator. For clarity, the variables $\phi$ and $\psi$ without a subscript refer to modal propositions, while those with subscripts (e.g. $\phi_0$) refer to propositions in the action language.

A PDL proposition is interpreted under a model, $M$, where $M$ is a set of positive terms. The semantics for the logic is defined as follows:

\[
\begin{align*}
M \models p &\iff M \supseteq p \\
M \models \phi \land \psi &\iff M \models \phi \land M \models \psi \\
M \models \neg \phi &\iff \text{not } M \models \phi \\
M \models [\alpha]\phi &\iff \forall M' \in [\alpha] (M) \cdot M' \models \phi
\end{align*}
\]

From this, one can see that a term $p$ holds if it is entailed by $M$, $\neg \phi$ holds if $\phi$ does not, and $\phi \land \psi$ holds if $\phi$ holds and $\psi$ holds. For the dynamic operator, $[\alpha]\phi$ holds if and only if $\phi$ holds in every final state, $M'$ of $\alpha$, as defined by the semantics. Other propositional operators such as implication and disjunction can be defined using these basic operators.

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

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

\[
\begin{align*}
\llbracket \psi? \rrbracket \phi &\iff \psi \rightarrow \phi \\
\llbracket \psi \rightarrow \psi' \rrbracket \phi &\iff \psi \dagger (\psi' \land \phi) \\
\llbracket \alpha; \beta \rrbracket \phi &\iff [\alpha][\beta] \phi \\
\llbracket \alpha \lor \beta \rrbracket \phi &\iff [\alpha] \phi \land [\beta] \phi \\
\llbracket N \rrbracket \phi &\iff [\alpha] \phi \quad \text{if} \quad N \equiv \alpha
\end{align*}
\]

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 $\psi$ and $\psi'$ while removing all terms whose negation occurs in $\psi'$.

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 [31]:

\[
\frac{[N] \phi \dagger [\alpha] \phi}{[N] \phi} \quad \text{where} \quad N \equiv \alpha
\]

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
Therefore, the weakest precondition is 
and \( q \) must hold: 
\[ p \land q \land p' \land q' \]

This protocol can be summarised using the following infinite tree structure:

\[
\begin{array}{c}
\psi' \\
\psi \quad \phi' \quad \phi \\
\end{array}
\]

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

Miller and McBurney [20] have demonstrated soundness and completeness of a proof system, including Scott induction, for a propositional dynamic logic over the \textit{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 \textit{weakest precondition} of a protocol is the weakest (or most general) proposition from which a protocol cannot become stuck. For example, consider the following protocol, in which \( p, p', q, \) and \( q' \) are literals and \( p \land q \land p' \land q' \) is satisfiable:

\[
(p \xrightarrow{\phi'} p') \land (q \xrightarrow{\phi} q')
\]

For this to successfully execute, the proposition \( p \land q \) must hold: \( p \) so that the first atomic protocol can execute, and \( q \) so that the second atomic protocol can execute. Therefore, the weakest precondition is \( p \land q \).

The \textit{maximal postcondition} is the strongest proposition that results from a protocol being executed under its weakest precondition. In the above example, the maximal postcondition that results when \( p \land q \) (the weakest precondition) holds is \( p \land q \land p' \land q' \).

In previous work, we have specified a method for proving the absence of stuckness in a protocol [18]. This method involves using the weakest precondition and maximal postconditions of protocols. A formal definition of stuckness, weakest precondition, and maximalpost-condition, is available in referenced article.

A \textit{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, \( \phi_G \), and an initial state, \( \psi_I \) (the state of the world from which an agent wants to achieve the goal – generally the current state), a \textit{weak matching} protocol is a protocol, \( \alpha \), that achieves the goal \( \phi_G \) from the initial state \( \psi_I \) for at least one outcome. Formally:

\[
\psi_I \rightarrow (\alpha)\phi_G
\]

A \textit{strong matching} protocol is a protocol that achieves a goal for all outcomes, assuming that there exists at least one outcome. 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 \( \phi_G \) from the state \( \psi_I \), the agent could simply use the PDL proof system discussed in Section 2.2. That is, for every protocol \( \alpha \), if the proof \( \psi_I \rightarrow (\alpha)\phi_G \) is successful, then \( \alpha \) 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 our 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 \( \psi_0 \), the protocol \( \alpha \) is guaranteed to

1. This assumption is subsumed by our assumption that a protocol is free from stuckness.
achieve the outcome $\phi_0$. 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 $P \equiv p \xrightarrow{\alpha} p'$, $Q \equiv true \xrightarrow{\beta} q'$, and $R \equiv true \xrightarrow{\epsilon} r'$. 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 $R$.

\[
\begin{array}{c}
p \quad \xrightarrow{\alpha} p' \quad p \rightarrow [P; Q](p' \land q') \quad p \rightarrow [P; R](p' \land r')
\end{array}
\]

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

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 $A \equiv a \xrightarrow{\alpha} a'$ and $B \equiv b \xrightarrow{\beta} b'$. 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 $a \lor b \rightarrow [A \cup B](a' \lor b')$. 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 be 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 $\psi_I$ is the weakest precondition, and $\phi_G$ the maximal postcondition.

\[
\text{characterise}(\psi_0 \xrightarrow{\alpha} \psi_1, \phi_0) \equiv \\
\begin{cases}
\text{false} & \text{if } \psi_0 \land \psi_0 \text{ false then return } \{} \\
\text{return } \{\phi_0 \land \psi_0 \rightarrow [\psi_0 \xrightarrow{\alpha} \psi_1](\phi_0 \land \psi_0) \land p\} & \text{else return} \\
\end{cases}
\]

\[
\text{characterise}(\alpha; \beta, \phi_0) \equiv \\
S := \text{characterise}(\alpha, \phi_0) \\
\text{for } \phi_1 \rightarrow [\alpha]\psi_1 \in S \text{ do} \\
S' := \text{characterise}(\beta, \psi_1) \\
\text{for } \phi_2 \rightarrow [\beta]\psi_2 \in S' \text{ do} \\
R := R \cup \{(\phi_2 \downarrow \phi_1 \rightarrow [\alpha; \beta] \psi_2)\} \\
\text{return } R
\]

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 $\phi_0$ is compatible with the precondition $\psi_0$ – 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 $\uparrow$) of the symbolic state and precondition with the postcondition.

The iterative operator defines a protocol of infinite size: $\alpha^*$ can iterate of $\alpha$ 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 [19]). Therefore, to characterise this, we simply need to characterise $\alpha$.

In [19], we present proofs of termination, soundness, and completeness of this algorithm.

### 4.3 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.

#### 4.3.1 Strong Matching

The relation for a strong match is more straightforward than for a weak match. To prove that a protocol, $\alpha$, is a strong match for goal $\phi_G$ and initial state $\psi_I$, one must prove the following:

\[
\forall a \in \text{chrs}(\alpha) \bullet \text{strong_match}(a, \psi_I, \phi_G)
\]

in which strong_match is defined as the relation

\[
\text{strong_match}(\psi_0 \rightarrow [\alpha]\phi_0, \psi_I, \phi_G) \iff
\]

\[
S := \text{characterise}(\alpha, \phi_0) \\
\text{for } \phi_1 \rightarrow [\alpha]\psi_1 \in S \text{ do} \\
S' := \text{characterise}(\beta, \psi_1) \\
\text{for } \phi_2 \rightarrow [\beta]\psi_2 \in S' \text{ do} \\
R := R \cup \{\phi_2 \downarrow \phi_1 \rightarrow [\alpha; \beta] \psi_2\} \\
\text{return } R
\]

\[
\text{characterise}(\psi_0 \xrightarrow{\alpha} \psi_1, \phi_0) \equiv \\
\begin{cases}
\text{false} & \text{if } \psi_0 \land \psi_0 \text{ false then return } \{} \\
\text{return } \{\phi_0 \land \psi_0 \rightarrow [\psi_0 \xrightarrow{\alpha} \psi_1](\phi_0 \land \psi_0) \land p\} & \text{else return} \\
\end{cases}
\]
and \( \psi_I \trianglerighteq \psi_0 \rightarrow \phi_0 \trianglerighteq \phi_G \)

and \( \text{chrs} \) 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.

### 4.3.2 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:

\[ \neg\forall a \in \text{chrs}(\alpha) \bullet \text{strong\_match}(a, \psi_I, \neg\phi_G). \]

This is equivalent to:

\[ \exists a \in \text{chrs}(\alpha) \bullet \neg(\psi_I \trianglerighteq \psi_0 \rightarrow \phi_0 \trianglerighteq \neg\phi_G) \]

\[ \equiv \exists a \in \text{chrs}(\alpha) \bullet \psi_I \trianglerighteq \psi_0 \land \phi_0 \trianglerighteq \phi_G \]

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.

In [19], we present soundness and completeness proofs for these matching methods.

### 4.4 Evaluation

Figure 2 from [19] shows the execution time (in milliseconds) for three algorithms: our characterisation algorithm, our matching method, and a complete depth-first search method that proves whether a protocol matches a goal or not. A depth-first search algorithm was used in the comparison because the complexity would be similar to that of using a theorem prover or model checker to find a matching protocol, as both theorem provers and model checkers would need to assess the same edges nodes as a DFS algorithm.

One can see from this 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. 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.

### 5 Structuring and Searching Protocol Libraries

In this section, we present our method for structuring protocol libraries for efficient searching. This method takes advantage of the partial ordering of predicate logic; that is, it relies on the entailment operator, \( \trianglerighteq \), and that protocol outcomes may often be related via this partial order.

#### 5.1 Protocol characterisations and posets

The definition of our first-order language from Section 2 satisfies the properties of a partially-ordered set (poset): a set paired with a partial order. In this case, the poset is \((\text{Prop}, \trianglerighteq)\), where \(\text{Prop}\) is the set of all predicates.

In this work, we are interested in structuring protocol outcomes. From Section 4, we know that protocol outcomes are specified using predicates; that is, for any characterisation, \([\alpha]\phi_0\), of protocol \(\alpha, \phi_0\) is a predicate.

From the theory of posets, we know that, given a poset \((S, \leq)\), then, for any set, \(S_0\), such that \(S_0 \subset S\), \((S_0, \leq)\) is also a poset. From this, we make the following observation:

The set of all possible characterisations, \(A\), forms a poset with the operator \(\leq\), \(\langle A, \leq \rangle\), in which the relation \([\alpha]\phi_0 \leq [\beta]\psi_0\) holds if and only if \(\psi_0 \supseteq \phi_0\). Any library of characterisations is a subset of \(A\), therefore, any library forms a poset with the inverse of the entailment operator.

This observation means that, given the set of protocols available to an agent, and their characterisations, we can form a poset from these characterisations by using the entailment operator as the partial order.

It is important to note that edges in the graph are relations between characterisations, and are based on the protocol outcome. That is, the edges do not represent any relation defined by the dynamic logic in which the characterisations are represented.

#### 5.2 Protocol libraries as subsumption hierarchies

A subsumption hierarchy is a directed, acyclic graph (DAG), in which the nodes of the graph represent values,
and the relationships between nodes represent that the source node subsumes the destination node. Given a poset, \((S, \leq)\), one can represent that poset in a subsumption hierarchy by taking \(S\) as the nodes of the graph, a creating an edge, \((s, t)\), between the two nodes, \(s\) and \(t\), if and only if \(s \leq t\). Thus, we can represent the poset \((A, \leq)\) using a subsumption hierarchy.

Example 5.1: Consider an example of a collection of outcomes using a constraint language, in which lower case letters are variables, and upper case letters are protocol names. Characterisations represent nodes, and, using the entailment operator as the partial order, one creates the DAG of characterisations shown in Figure 3.

It is straightforward to see that the predicate at each node subsumes the predicate at all nodes below it. The node on the top-left represents a protocol with an outcome in which an item is bought, while its two leftmost children by specific items, and its rightmost child buys an item under a particular price. Note that it is not the case that every graph is connected to every other graph. In this example, there are three distinct subgraphs.

To improve the efficiency of storage and retrieval, the library is stored as a transitive reduction.

Definition 5.1: Transitive reduction A transitive reduction of a graph \(G = (V, E)\) is a relation \(E^-\) such that the transitive closure of \((V, E^-)\) is equivalent to the transitive closure of \((V, E)\). In other words, it is the minimal set of edges such that for any pair of nodes that are linked by \(E\) (either directly or transitively) are linked by \(E^-\).

Formally, this can be defined as follows:

\[
E^- = E - (E \circ E^+),
\]

in which \(E^+\) is the transitive closure of \(E\) and \(\circ\) is functional composition.

We define the relation \(TR(G)\) for a graph \(G\) as:

\[
TR(G = (V, E)) \iff E = E - (E \circ E^+).
\]

Most work on transitive reductions focusses on the calculation of a transitive reduction given a graph, however, we define an algorithm for inserting a new node into a graph, which calculates the new edges and maintains the property of a transitive reduction.

There are several algorithms for maintaining transitive reductions under insertion, however, all of these consider either the addition of an edge to the graph [14], [12], or the insertion of the node along with edge information [5], whereas we insert a node and compute the new edges on the fly.

Our algorithm is outlined in Algorithms 1 and 2. Algorithm 1 is split into two parts: The first part traverses the graph from top to bottom, inserting edges \(v \mapsto n\) whenever \(n\) entails a node \(v\) but does not entail its children. The second part traverses the graph from bottom to top, adding a node when \(n\) is entailed by a node \(v\), but not by any of its children.

Algorithm 1 presents the top-down algorithm, \(dfs\_insert\), which is a depth-first traversal. The algorithm visits only nodes that have not been visited in the top-down search previously. Line 4 terminates the search if the new node \(n\) does not entail the current node. If this is the case, \(n\) cannot be an ancestor of \(n\). Lines 6 to 15 attempt to insert the new node as a child of the current nodes children. If this succeeds \((c\_inserted\) is true), then \(n\) cannot be a child of the current node, because this would violate the transitive reduction property. If it does not succeed, but the child node implies the new node, then the new node is inserted in between the current node and the child.

Lines 16 to 19 specify that if the new node was not added as an ancestor of any of the current node’s children, then it must be a new child of the current node. The node \(v\) is labelled as “inserted” to inform the search later in the case that \(v\) is visited again. This saves us from traversing the same nodes.

The algorithm for bottom-up traversal is similar to Algorithm 2, except that it considers edges in which the entailment is reversed.

Algorithm 1 insert\((G, n)\): insert a new node, \(n\), into DAG, \(G = (V, E)\), returning a graph \(G' = (V', E')\)

Require: \(TR(G)\)

Ensure: \(V' = V \cup \{n\}\) and

\[
\begin{align*}
&\text{for all } v \in V', n \not\supseteq v \text{ implies } n \mapsto v \in E^+ \text{ and } \\
&\text{for all } v \in V, v \not\supseteq n \text{ implies } v \mapsto n \in E^+ \text{ and } \\
&TR(G').
\end{align*}
\]

1: for all source nodes \(v\) in \(G\) do
2: \(dfs\_insert(G, n, v)\)
3: end for
4: for all leaf nodes \(v\) in \(G\) do
5: \(rdfs\_insert(G, n, v)\)
6: end for

5.3 Termination, Soundness, and Completeness

The algorithms are depth-first search algorithms on acyclic graphs, so termination is implied.

Theorem 5.1: Algorithm 1 is sound and complete.

Proof: To prove soundness, we must show that any edge added is valid and preserves the transitive reduction property.

The algorithm only adds an edge \(v \mapsto n\) if \(n \not\supseteq v\), which implies the edge is valid. Preserving the transitive reduction is done in lines 8 to 13. To show that this preserves the property of transitive reduction, we note the following property of transitive reductions:

\[
(x, y) \in E \text{ implies } \neg \exists n \in V \mid (x, n) \in E^+ \land (n, y) \in E^+
\]

Informally, if there is an edge between two nodes, there can be no other path between them.

If \(\neg c\_inserted\) and \(c \not\supseteq n\) holds at Line 8, then \(n\) was not inserted as an ancestor of child \(c\), however, \(c \not\supseteq n \supseteq v\).
Algorithm 2 $dfs_{\text{insert}}(G, n, v)$: insert a new node, $n$, into $DAG$, $G = (V, E)$

Ensure: $V' = V \cup \{ n \}$ and for all $v \in V$, $n \geq v$ implies $n \leftrightarrow v \in E^+$ and $TR(G)$

1: $\text{result} \leftarrow \text{false}$
2: if $v$ is not visited then
3: label $v$ as visited
4: if $n \geq v$ then
5: $\text{inserted} \leftarrow \text{false}$
6: for all children $c$ of $v$ do
7: $c_{\text{inserted}} \leftarrow dfs_{\text{insert}}(c, n)$
8: if $\neg c_{\text{inserted}}$ and $c \geq n$ then
9: remove edge $v \leftrightarrow c$
10: add edge $n \leftrightarrow c$
11: add edge $v \leftrightarrow n$
12: $\text{inserted}(v) \leftarrow \text{true}$
13: end if
14: $\text{inserted} = c_{\text{inserted}}$ or $\text{inserted}$
15: end for
16: if $\neg \text{inserted}$ then
17: add edge $v \leftrightarrow n$
18: $\text{inserted}(v) \leftarrow \text{true}$
19: end if
20: end if
21: else if $v$ is visited then
22: $\text{result} \leftarrow \text{inserted}(v)$
23: end if
24: return $\text{result}$

Therefore, we must have the edges $v \leftrightarrow n$ and $n \leftrightarrow c$, while removing the old edge. From the property above, this preserves the transitive reduction, because there can be no other path between the two nodes.

To prove completeness, we must show that any edge that needs to be added is added. For a node $n$ to be an ancestor of $v$, it must be that $n \geq v$. In this case, there are three possibilities: 1) $n$ is an ancestor of one of $v$’s children; 2) $n$ is between $v$ and one or more of its children; 3) or $n$ is a child of $v$ but entails none of its children.

Case 1 is added via the recursive call to $dfs_{\text{insert}}$. Case 2 is added in lines 8 to 13. Case 3 is added between lines 16 to 19.

5.4 Searching Structured Libraries

In this section, we present in outline an algorithm for searching libraries structured as subsumption hierarchies, as discussed in Section 5.2. In Section 6, we discuss the complexity of such an algorithm, comparing this to searching an unstructured library.

5.4.1 Unstructured Libraries

If an agent’s library is unstructured, and it wants to find all protocols that achieved a certain goal, it would need to try matching every characterisation in the library. We assert that, if an agent structures its library as outlined in Section 5.2, this can be improved upon.

5.4.2 Structured Libraries

We first note the following three-part theorem:

Theorem 5.2: If $[\alpha] \phi_0 \leq [\beta] \psi_0$ is an edge in a subsumption hierarchy, and $\phi_G$ is a goal, then the following propositions hold:

1) if protocol $\alpha$ achieves the goal $\phi_G$, then protocol $\beta$ will also achieve that goal;
2) if all outcomes of protocol $\alpha$ are inconsistent with the goal $\phi_G$ — that is, $\phi_G \land \phi_0 \models \text{false}$ — then all outcomes of protocol $\beta$ will also be inconsistent with that goal;
3) if protocol $\beta$ does not satisfy the goal $\phi_G$, then protocol $\alpha$ will also fail to satisfy the $\phi_G$.

Proof: Parts 1 and 2 follow directly from the entailment operator. Part 3 also follows from entailment, but requires a few steps that we outline here.

This is proved via contradiction. If $\beta$ does not satisfy the goal, then $\psi_0 \nvdash \phi_G$. We know that $\psi_0 \nvdash \phi_0$, from the fact that $([\alpha] \phi_0, [\beta] \psi_0)$ is an edge in the subsumption hierarchy. If the theorem is false, then it must be possible for some $\phi_0$ that $\phi_0 \nvdash \phi_G$. If we assume this is the case, then we have the following three facts:

(a) $\psi_0 \nvdash \phi_G$;
(b) $\psi_0 \supseteq \phi_0$; and
(c) $\phi_0 \supseteq \phi_G$. 

Fig. 3. An example library organised as a subsumption hierarchy using the entailment operator as a partial order.
However, from facts (b) and (c), and the transitivity of the $\supseteq$ operator, we infer a new fact $\psi_0 \supseteq \phi_G$, which directly contradicts fact (a). Therefore, the proposition holds.

From these theorems, we know that, given a subsumption hierarchy, the following three properties hold:

1) If an agent is attempting to match a goal, $\phi_G$, and it evaluates a node, $\phi_A$, such that $\phi_A \supseteq \phi_G$, then every node that is a transitive child of $\phi_A$ in the hierarchy must also satisfy the goal.

2) If an agent is attempting to match a goal, $\phi_G$, and it evaluates a node, $\phi_A$, such that $\phi_A \land \phi_G \supseteq \text{false}$ — that is, $\phi_A$ and $\phi_G$ are inconsistent with each other — then every transitive child of $\phi_A$ must also be inconsistent with $\phi_G$.

3) If an agent is attempting to match a goal, $\phi_G$, and it evaluates a node, $\phi_A$, such that $\phi_A \not\supseteq \phi_G$, then all transitive parents of the node $\phi_A$ in the hierarchy must also not satisfy the goal.

This can cut our search time down quite effectively, depending on the hierarchy and the goal. Consider the hierarchy from Figure 3, and the goal $x \in 0..30$. The agent matches the node $x \in 0..20$, because this entails the goal, the agents knows that every protocol at that node and at all four nodes transitively related to it, satisfy the goal, and are implicitly matched. Alternatively, if the goal is $x = 50$, then the proposition $x \in 0..20 \land x = 50$ is unsatisfiable, so there is no need to search the children of the node $x \in 0..20$, because none of them will be consistent with $x = 50$ either.

5.4.3 Depth-first search

If we were to search a structured library using a depth-first search, we can use the above properties to reduce the search. Given a goal, the search continues deepening until one of four properties holds for a node:

1) the node has no children (a dead end);
2) each of the nodes’ children has been visited already;
3) the characterisation does not imply the goal.

The first and second properties are standard for a depth-first search, and in those instances, the algorithm retreats to the parent from which it came and continues. However, the third and fourth properties remove any transitive children from the search by marking them as “visited”. In the third case, all nodes subsequent in the search are added as matches, and in the fourth case, no nodes are added. As with the first two cases, the algorithm retreats to the parent from which it came and continues.

5.4.4 Reverse depth-first search

If we were to search a structured library using a “reverse” depth-first search — that is, starting at the leaf nodes and search up the hierarchy —, we can use property 3 above reduce the search. Given a goal, the search continues up the hierarchy until one of three properties holds for a node:

1) the node has no parents (a dead end);
2) each of the nodes’ parents has been visited already; or
3) the characterisation does not imply the goal.

As above, the first and second properties are standard for a depth-first search. The third case, all parents are removed from the search by being marked as “visited”. In each case, the algorithm retreats to the child from which it came and continues.

In fact, this “reverse” depth-first search is a standard depth-first search, except the edges in the graph are reversed; that is, an edge $([a]\phi_0, [\beta]\psi_0)$ exists if $\phi_0 \supseteq \psi_0$, instead of the other way around.

5.5 Improving search efficiency using domain information

One downfall of this method is that an entire hierarchy may be searched despite the characterisations being about an unrelated domain. Consider the example graph in Figure 3. The largest subgraph in this graph contains a source node representing a protocol labeled $A$ with the characterisation $[A]x \in 0..20$, which has a collection of (transitive) descendants, each of which further constrains the value of $x$, or the values of $x$ and $y$. Now, consider that an agent has a goal $z = 0$. When evaluating the source node, the agent will note that $x \in 0..20$ and $z = 0$ are consistent with each other, but that $x \in 0..20$ does not achieve the goal $z = 0$. As such, it will continue deepening its search until the entire graph has been traversed, finding no matches. Assuming that $z$ is an unrelated domain, this search is wasteful.

In Figure 3, each of the nodes in the left-most hierarchy (i.e., that sub-graph containing the six protocols labeled $A$ through $F$) refer only to the variables $x$ or $y$, therefore, none can constrain the value of $z$. If we consider an agent with a large protocol library, it is likely that for an arbitrary goal, most of the protocols will not refer to variables in that goal. In this example, we are using a constraint system as the underlying first-order language, however, in a system such as Prolog, the term functors could be used to represent the domain instead. In the remainder of this section, we will refer to these as the domain variables.

Using the search process outlined above will result in the agent assessing most protocols in the library — an approach which is not much more efficient than simply assessing every protocol.

5.5.1 Improving depth-first search using domain variable propagation

To improve the standard depth-first search algorithm, we alter both our process of structuring libraries and searching them. Structuring is altered by recording the set of domain variables in a characterisation, as well as all domain variables in all descendants. In Figure 3, all
nodes in the left-most hierarchy would be paired with the set of variables \(x, y\), the unconnected node would be paired with \(z\), while the final two nodes would have \(y, z\). We refer to this as variable propagation, from the fact that the domain variables propagate up the graph.

We add an additional condition for terminating the deepening of our depth-first search algorithm: the cases in which all variables in the goal are not annotated to the current node. In our above example, if an agent has a goal containing only the variable \(z\), then it calculates that \(\{z\} \not\subseteq \{x, y\}\), therefore eliminating the largest graph in Figure 3 from our search. Assuming that the domain variables in characterisations is more efficient than the entailment operator in our constraint language, which we do not believe is unreasonable, this can reduce the overall complexity of our search.

5.5.2 Improving reverse depth-first search using domain variable information

To improve the reverse depth-first search is more straightforward. First, the structuring process does not require altering, as we know that if \(\psi_0 \supseteq \phi_0\) for an edge \((\alpha|\phi_0, \beta|\psi_0)\), then the domain variables in \(\psi_0\) are a subset of those in \(\phi_0\).

To terminating the deepening of the our reverse depth-first search algorithm, we check whether the domain variables in the goal are a subset of those in the current node. If not, then performing an expensive entailment operation will not yield a positive result. Furthermore, the parents of the current node will also not entail the goal, which we know from Theorem 5.2.3.

5.6 Termination, Soundness, and Completeness

Theorem 5.3: Both depth-first search algorithms terminate.

Proof: Termination of these improved algorithms are straightforward to show. As with any depth-first search, deepening terminates when they are no children at the current node, which is guaranteed to terminate because the subsumption hierarchy is both finite and acyclic. The current node, which is guaranteed to terminate because the subsumption hierarchy is both finite and acyclic. The straightforward to show. As with any depth-first search, we check whether the domain variables propagate up the graph.

Theorem 5.4: Both depth-first search algorithms are sound, and are complete for goals containing no tautologies.

Proof: The method for matching a single protocol has been proved sound in earlier work [19]. For the forward depth-first search, the algorithm includes all of the current node’s children as matches as well. This is sound from Theorem 5.2(1). For the backward depth-first search, the algorithm only includes matches if the entailment holds, which is sound from our earlier work [19].

For completeness, we have to demonstrate that those nodes in the graph that are omitted from the search cannot possibly achieve the goal. However, this is not the case, but it is almost the case.

Firstly, we identify a counter-example showing that neither algorithm is complete. Consider a characterisation \([N]x = 1\), and a goal \(x = 1 \land y = y\). Clearly, the proposition \([N](x = 1 \land y = y)\) holds, because \(x = 1\) holds for all end states of \(N\), and \(y = y\) is a tautology. However, if the characterisation is in a graph containing the variables \(x\), the proposition \([N](x = 1 \land y = y)\) will never be proved, because the deepening will terminate at the source node in the forward depth-first search, for which the propagated set of variables, \(\{x\}\), is not a superset of the domain variables in the goal, and will terminate at the leaf nodes to the reverse depth-first search.

If we assume that agents do not contain tautologies in their goals, then our algorithms are both complete. To prove this, we must show that characterisations that omitted from the search cannot possibly achieve the goal.

In the forward depth-first search, there are two cases in which characterisations are omitted: if the goal and current node are incompatible, and if the domain variables in the goal are not in the propagated variables at the current node. Theorem 5.2.2 shows that the first case is correct. For the second case, if we consider the characterisation, \((\alpha|\phi_A, \beta|\psi_0)\), then it is not possible that \((\alpha|\phi_G, \beta|\psi_0)\), for goal \(\phi_G\). We know that the domain variables in the goal are not a subset of the propagated variables at the current node. All children of the current node have at most, the same domain variables as the current node. The definition of \(\supseteq\) specifies that an entailment is true if and only if the possible bindings for all variables on the left is a subset of the possible bindings for all variables on the right. If there is a variable on the right that is not on the left, then the possible binding on the left includes all bindings for that variable, therefore, the only time this could be entailed is when the possible bindings on the right contains all bindings. From this, we conclude that constraints on that variable must be a tautology. Since we assume that there are no tautologies, then our forward depth-first search algorithm is complete.

In the reverse depth-first search, there is on which is which characterisation are not evaluation: when the current node is not a match, all transitive parents are excluded. This is valid from Theorem 5.2(3), so our reverse depth-first search algorithm is complete.

Regarding the issue of tautologies in goals, we offer two items of discussion. Firstly, the algorithm can be modified to check for tautologies; that is, prove that the constraints on domain variables not in the protocol are universally true. Alternatively, the algorithm can be left as is, because we believe the approximate solution is better than the above alternative. It would be unusual for a rational agent to contain a tautology in its goals — in fact, we believe this would indicate an agent with far greater problems than matching protocols — and testing for tautologies in a goal would be unnecessarily increasing the overhead of the search, just to avoid a problem that is unlikely to occur.
6 Analysis

In this section, we present an analysis of the structuring and searching of protocol libraries, and compare this to the case of unstructured libraries. We focus on the number of entailments that the must be performed, because we believe that this would be the most expensive part of the search process. Our assertion is that using structured libraries significantly reduces the number of entailments. For the purpose of analysis, we treat all entailment operations the same; that is, they have the same complexity. This is not the case, as predicates have no restrictions, however, there is no other basis to use.

A brief complexity analysis is presented that outlines the best- and worst-case time complexities of the linear and depth-first search techniques. The average-case complexity is somewhat more complicated, so an experimental evaluation is performed to estimate the average-case time complexity.

6.1 Informal analysis

We assume that an unstructured library would be implemented using a list-like structure. For such an implementation, the insertion of a new characterisation into the library would have time complexity $O(1)$: the protocol’s characterisation is simply added to the end of the list.

For search, we assume that an agent would like to find every possible match in its protocol libraries, and then deliberate over these further. Such a search requires an agent to check every characterisation, each requiring two entailments (one for the precondition and one for the postcondition), so for an unstructured library, the best and worst cases are both in $O(n)$.

Assuming a subsumption hierarchy-based library, and using the algorithm presented in Section 5.4, an agent can significantly improve on its search time. The downfall to our method is that insertion time is increased. Inserting a node would require an algorithm that searches each connected graph in the hierarchy until either a correct place is found, or until the entire hierarchy has been completely traversed. Assuming that the agent uses a standard depth-first search without analysing the variables at nodes, the complexity of the insertion is $O(n)$ for a single node. Considering that $n$ nodes must be inserted, the worst-case complexity of generating an entire library is in $O(n^2)$.

Searching the libraries is where we see an improvement. The best and worst-case complexities for the forward depth-first search are straightforward: the best case is that every protocol in the library forms a single connected graph with exactly one source node, and the goal is entailed by that source node. For a reverse depth-first search, the best case is a single connected graph with one leaf node, with the node failing to satisfy the goal. Each of these cases has the complexity of $O(1)$. The worst cases are the same as for a linear search: when the goal is entailed by no characterisations, but compatible with all for a forward depth-first search; and when the goal is entailed by all characterisations for a reverse depth-first search. Both of these is therefore $O(n)$.

For the average-case, we have a number of considerations. Firstly, the structure of the subsumption hierarchies can range anywhere from those in the best-case analysis to those in the worst-case analysis. At present, we have little idea as to the relationships between protocol outcomes, or the probability of how many protocols will be entailed by others. In addition, these properties will vary depending on the underlying logic that is used, the domain of the protocols, the types of the protocols, and the nature of the interactions the protocols define. Therefore, instead of a formal analysis, we performed an experimental analysis, which gave us some idea of the complexity of our depth-first search approach.

6.2 Experimental evaluation of average-case complexity of search

In this section, we define an experiment to measure the average-case complexities of our search algorithm and the linear search algorithm using randomly generated subsumption hierarchies. This evaluation is generic; that is, the results hold for any subsumption hierarchy over logic formula, and are not specific to protocol libraries. The experiment is used to establish the average-case complexity of the algorithms, not as a representation of actual protocol libraries.

6.2.1 Independent variable

The independent variable in the experiment is the search algorithm used to find matching outcomes. We used the following six search algorithms:

1) Linear search (LS): this is a standard linear search algorithm, using a list as the data structure.

2) Linear search with variable checking (VLS): another linear search algorithm, with the exception that this algorithm uses a technique similar to variable propagation. Instead of simply testing if an outcome entails a goal, the algorithm first checks that all variables in the goal are present in the outcome. This is included under the assumption that checking the variables would require less time and space than performing an entailment operation, similar to variable propagation.

3) Depth-first search with variable propagation and compatibility checking (CVDFS): the modified depth-first search algorithm with variable propagation described in Section 5.4.

4) Depth-first search with variable propagation (VDFS): the modified depth-first search algorithm with variable propagation described in Section 5.4, however, without using the compatibility check to terminate the deepening. This was included after the first experimental run due to some unusual results regarding CVDFS.

5) Reverse depth-first search (RDFS): the reverse depth-first search algorithm described in Section 5.4.
6.2.2 Measures
There are two measures taken in this experiment:
1) The number of entailment operations performed in the search.
2) The number of variable checks performed in the search.

These measures are taken because they are considered to be the most expensive operations in the search algorithms. The second measure, variable checks, is only relevant for the VLS, VDFS, and CVDFS algorithms, as the other algorithms to not perform variable checks.

6.2.3 Experiment setup
For the purpose of experimentation, we use an integer-based constraint solver, and we restrict the maximal value of the range\(^2\). This provides us with an easily calculatable, finite set of constraints representing outcomes.

Several experiments were performed, each with the following parameters modified:
1) The size of the library, relative to the number of outcomes generated. This is incremented by 10%, starting at 0%, up to 100%.
2) Single vs. multiple matches. That is, whether the algorithm is required to find the first matching outcomes, or all matching outcomes.
3) The disjointedness of the constraint space. That is, if one adds characterisations for all possible outcomes, how many subgraphs would result? We simulate only two values for this: one subgraph (the entire library is connected) and five subgraphs. This is simulated by having distinct sets of variables for each constraint space.
4) The goal for which we are searching. We vary this from having searches in which the goal is not present in the library, the goal is present in the library, and the goal is randomly selected from the constraint space.

We randomly selected the required number of nodes from the constraint space, and constructed libraries from these (both lists and subsumption hierarchies). For each simulation, 100 different libraries were generated, and for each library, 10 different goals were searched for.

6.2.4 Threats to validity
The main threat to validity of this experiment is related to the protocol libraries that are generated. We generate a constraint space of a certain size, and then randomly select characterisation outcomes from this constraint space to form a library. This results in libraries with certain relationships between them; for example, a graph consisting of 100% of the constraint space forms a complete lattice. We have little idea whether these libraries and the relationships they comprise reflect those that we would see in real protocol libraries, therefore, generalising the results to all possible protocol libraries is not possible. However, an investigation into real protocols, their characterisations, and the relationships between them is only worth tackling if the current experiment produces positive results.

The second major threat to validity relates to the method for generating protocols libraries. It would be possible to use different methods for random generation of libraries, and these may produce different results.

6.2.5 Results
Results of the experiments are summarised in Figure 4 (results of searching for all matches) and Figure 5 (results of searching for one match), which plot the number of entailment operators performed against the number of characterisations in the library for the different search algorithms, and in Figure 6, which shows the results of the number of variable checks.

First, we discuss the results in Figure 4. Unsurprisingly, both linear searches have the average case complexity of \(O(n)\), where \(n\) is the number of characterisations in the library. However, VLS is considerably more efficient, using half as many entailment operations for a fully connected library, and approximately one-tenth for a library containing five disjoint subgraphs. The difference in these results is due to the fact that, for disconnected subgraphs, the nodes in the different subgraphs have different sets of variables in their characterisations, so the VLS algorithm will detect this before performing an entailment operation, even though it searches the list-based library.

The DFS-based algorithms all have the average case complexity of \(O(n)\). The CVDFS algorithm performed the worst out of these, performing more entailment operations than a standard linear search for loosely connected libraries (Figures 4(a)-4(c)). This is due to the fact that performing a compatibility check must be done using the entailment operator, so for many nodes, two entailment operations are performed; thus, the worst-case is \(2n\) entailments for a library of \(n\) characterisations. Once the library becomes highly connected (that is, moving along the x-axis), the compatibility check ends the search deepening more often. This is shown by the fact that CVDFS significantly out-performs LS in Figures 4(d)-4(f).

In cases when the goal is not present in the library, VDFS and VLS are equivalent, as both must assess every characterisation in the library, only performing an entailment if the variable check passes. However, for cases in which the goal is present, VDFS performs marginally better.

RDFS is the most efficient for all graphs in Figure 4. It performs especially well for cases in which the goal is not present. This is because, if a goal does not match any entry in a library, only the leaf nodes will be evaluated. If a leaf node does not match, the deepening terminates.

Overall, the results for all matches indicate that any search method that takes the domain information into account will outperform those that do not, as demonstrated
by the VLS algorithm’s performance. More importantly, the results indicate that RDFS is superior to the other assessed search techniques.

For the single search methods, the results in Figure 5 are somewhat different from those for searching all matches. For the goal being not present (Figures 5(a) and 5(d)), the results are the same to that of matching all characterisations, due to the fact that the search will not terminate for a match being found if there is no goal present.

For the goal present and random goals, the results are different. First, all algorithms have the time complexity in $O(\log n)$. This is due to the fact that there may be more than one characterisation that matches the goal. Therefore, as the size of the list increases, the likelihood of finding a match earlier increases. For example, if the list contains 10 characterisations, of which 5 match, we are likely to find a match within 2 characterisations. If the list contains 100 characterisations of which 50 match, we are still likely to find a match within 2 characterisations.

Second, the increase in time for RDFS is almost indistinguishable on this graph. This is because, if we are searching for a single match, it is more likely that a “strong” constraint will match a goal as opposed to a “weak constraint”. For example, the constraint $x \in 1..10$ entails any constraint $x \in A..B$ in which $A \leq 1$ and $10 \leq B$, whereas the stronger constraint $x = 10$ will entail these, as well as constraints in which $A \leq 10 \leq B$. The reverse depth-first search will start at the lead nodes in the subsumption hierarchy, which are more likely to achieve the goal than those higher up, therefore, the search is more likely to terminate early.

The difference between VLS, VDFS, and RDFS are much the same as in Figure 4, with RDFS significantly more efficient than than that other two. The overall conclusions of the entailment data for single searches are the same as that for all matches: RDFS is the most efficient search algorithm.

Finally, we discuss the second measure that was taken for the experiment: the number of variable checks performed in the VDFS, CVDFS, and VLS search methods. Figure 6 outlines the results of this, although only the results for random goals and disjointedness of five are displayed; results of others follow a similar pattern.

As expected, the subsumption hierarchy-based techniques perform considerably less variable checks than the linear search, for matching both all characterisations, and a single characterisation. We do not believe this result is particularly important in relation to the other search methods, as the cost of performing a variable check would be negligible when compared to the cost of performing an entailment.

6.3 Average case complexity of constructing subsumption hierarchies

In our experiments, we also measured the amount of entailment operations required to construct a subsumption hierarchy. As discussed in Section 6.1, the worst-case complexity of inserting into a new characterisation
into a list is $O(1)$, but increases to $O(n^2)$ for a subsumption hierarchy. The question that remains is: how many searches have to be performed before using a subsumption hierarchy-based search method is worthwhile?

Figure 7 plots the number of entailments required to construct a subsumption hierarchy-based library against the number of characterisations in that library. This demonstrates the average-case complexity is in $O(n^2)$, which is not a good result. From this, we can conclude that the number of searches to be performed in order to make the subsumption hierarchy-based methods worthwhile increases with the size of the library. In Figure 7, at the highest point in the graphs, one would require just over 100 searches before the subsumption hierarchy-based techniques paid off.

It is our belief that searching would be performed a considerable amount of times more often than an insert, and therefore the variable-based DFS searches are more efficient than a linear search. In addition, insertion of protocols into a library of protocols may be undertaken offline, or by dedicated entities not engaged in seeking protocols at run-time for interaction with other agents, and thus protocol-library insertion may be separated temporally, geographically and control-wise from protocol-library search.

### 6.4 Discussion

From our analysis, we see that the overall complexity of insertion is increased for subsumption hierarchy-based methods, while the average number of entailments in a search is decreased. A trade-off must be made between these two. For agents that will be querying a protocol library more often than inserting new protocols, the trade-off would edge towards subsumption hierarchy-based methods. For agents that do not query the library often (compared to the number of inserts) and linear search and unstructured library will prove more beneficial.

From the results of the experiment, we conclude that using RDFS, the reverse depth-first search, is consistently the most effective method to use, although VLS, linear search with variable checking, performs well compared to DFS-based searches with no variable propagation.

Clearly, one of the questions that can arise is related to the formation of subsumption hierarchies. Given a set of characterisations, it is possible that none of the characterisations are related via the entailment operator, and we are therefore left with only source nodes. From a perspective of searching, this would be equivalent to a linear search. However, it is our belief that, given a set of characterisations, there will be sufficient relationships between characterisations to make our case worthwhile. After all, one of the visions of first-class protocols is that agents can choose different protocols that achieve the same goal, but which differ on other aspects.

Further improvements can be made to the above results by ordering the source nodes of the subsumption hierarchy into a subsumption hierarchy themselves, using the propagated variables at the nodes as values, and the subset operator as the partial order. Rather than
search through all of the source nodes, one can search the source nodes as a graph, only visiting the children if the source node’s variables are a superset of the variables in the goal. If the current node is not applicable, and its children have fewer variables, then the children cannot be applicable either.

7 Related Work

7.1 Protocols as first-class entities

There are many approaches to specification of executable protocols, such as Yolum and Singh’s commitment machines framework [35], Robertson’s Lightweight Coordination Calculus [29], De Silva et al.’s Petri-Net approach [8], Pham et al.’s temporal linear logic approach [24], and Leόn-Soto’s state-action space approach [13]. McGinnis and Miller [15] provide a detailed presentation of the state-of-the-art in this area, and presents the advantages and disadvantages the different approaches, include RAS.A. The general idea of annotation and matching could be applied to protocols specified in these languages, and the subsumption hierarchy method for structuring and searching protocol libraries could be employed with them.

As far as the authors are aware, there has been no investigation into the structuring and searching of either protocol or plan libraries for agents. Libraries, such as the plan libraries found in the Procedural Reasoning System [9], are searched sequentially, and offer no support for other techniques. In addition, the matching of plans to goals usually involves testing whether the plan postcondition unifies with the goal, which is different to our approach of matching, due to our use of maximal postconditions.

Characterisation of agent plan libraries and interaction protocols has been investigated by several other authors. Bussman et al. [4] 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, and storage of protocol libraries is not discussed.

Clement and Durfee [7] present a method for characterising precondition and postconditions of hierarchical task network plans. Their technique for characterising the plans is similar to the technique employed by us [19], except they support only sequential composition and choice, with no consideration for iterative or recursive plans. Their motivation for summarising plans is to allow the coordination of plans at an abstract level to prevent conflict, and as such, their approach would
Fig. 8. A plot of the number of node entailments as a percentage of the total number of nodes for each method searching all matches.

Fig. 9. A plot of the number of node entailments as a percentage of the total number of nodes for each method searching a single match.

benefit from an efficient storage and retrieval method. However, this does not appear to be considered.

Chopra et al. [6] describe a method for verifying whether an agent’s goal can be achieved by a protocol specification. Protocols are specified as sets of commitments, and Chopra et al. specify semantics for deter-
mining whether an agent’s capabilities can be combined with a protocol specification to bring about the necessary commitments for the goal to be achieved. They define a “stronger than” operator that could be used as a poset operator over a set of protocol commitments, and therefore, our structuring and searching method is compatible with their matching method.

Planning and temporal projection methods [10], [27] 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. As such, efficient storage and retrieval mechanisms would not be of use, unless one encounters the same start states many times.

Our concept of protocol libraries is similar to plan libraries, such as those found in many BDI and goal-oriented agent frameworks, such as the Procedural Reasoning System [9]. The authors are unaware of any work that describes methods for storing plan libraries to allow for efficient searching. This is likely due to fact that BDI programming languages are typically built on top of logic programming languages, which have similar mechanisms for storing and search terms. However, to achieve the full vision of first-class protocols, we believe that large protocol libraries would be useful, and efficient searching of these would become important.

Furthermore, we believe that such techniques would be useful in plan libraries. Even for relatively small libraries, any agent that consistently queries its plan library would benefit from an efficient storage and retrieval mechanism.

### 7.2 Reasoning about interaction protocols

Winikoff [34] presents a mapping from commitment machines to an abstract BDI programming language for agents called the Simple Abstract Agent Programming Language (SAAPL). The resulting SAAPL programs contain a set of agent plans for each role that execute the protocol.

This approach is somewhat different to the idea of protocols as first-class entities, as outlined in the introduction, in that Winikoff’s agents do not inspect protocols to decide their course of action, but are instead implemented as a mapping from the commitment machine into a SAAPL program. However, it is clear that agents could compile new plans at runtime when new protocols are encountered using Winikoff’s technique, making this a viable alternative to dynamic reasoning for certain applications.

Yolum and Singh [36] discuss the use of an abductive planner for agents to calculate execution paths in a commitment protocol. The planner uses a set of axioms relating communicative acts and commitments to derive all possible paths in a protocol from an initial state to an agent’s goal state. From this, decisions can be made about which path is the most preferred using standard search algorithms.

Pham et al. [25] presents the choice calculus, based on temporal linear logic (TLL). Agent designers specify adaptive interaction logic using TLL. The sequent calculus for temporal linear logic inherits the inference rules for internal choice and indeterminate possibility from linear logic. For use by an agent, this implies the following: 1) agents must prepare for all indeterminate possibilities; 2) internal choice is made with only local information; and 3) internal choices cannot be made before specified by the protocol. Pham et al. caution that while this is useful for agents that require safe behaviour, it requires more computational resources for an adaptive agent to handle all circumstances efficiently and correctly. For agents that are permitted riskier behaviour, such a calculus is unnecessary and can be improved by using predictions to improve resource efficiency.

Pham et al.’s choice calculus provides additional inference rules that take these three drawbacks into account, which allows agents to use predictions about the future. Furthermore, Pham and Harland split a formula into two sub-formula in order to turn an interaction regarding a specific goal into sub-goals, and apply the inference rules to the sub-goals in concurrent threads. The splitting itself is domain independent, so agents are able to decide which sub-goals to achieve by calculating partial achievement of goals, and which sub-goals still remain to be proved.

### 7.3 Subsumption lattices in AI

Subsumption lattices have been used to reduce search spaces in inference [30] and in logic programming languages. For example, Muggleton [22] maintains a store as a subsumption lattice, and searches only the relevant parts of the lattice to match an inductive logic programming query. A subsumption hierarchy is similar to this, although it is not required to form a complete lattice. Logic programming languages take advantage of the lattice property and the existence of least upper bounds. Considering this, it is possible that some agent platforms are built on top of logic programming languages that implement such techniques, and as such, the plan libraries may be stored using the default constraint store.

The use of subsumption information is used for search ontology bases. Baadar et al. [3] empirically assess four algorithms for classifying all classes in an ontology into a subsumption hierarchy. While the output of the algorithms is a subsumption hierarchy, the relationships between classes as defined by the ontology also contain a partial order that can be used to reduce the search space. One of the algorithms presented by Baadar et al.— the enhanced traversal method — exploits the partial order in the same way that our algorithms do, by terminating the search of branches in the graph when the current node implies the goal of the search, including all children as matches. An empirical assessment of the algorithm demonstrates that less of the class space in the ontology needs to be searched to construct a complete subsumption hierarchy, which is consistent with our results.
More recent advances in the construction of subsumption hierarchies from ontologies focus on the parallelisation of the algorithms; e.g. [2], [1]. From that work, it seems possible that our algorithms could be parallelised as well, however, we have no intention of investigating this.

8 Conclusion
In this paper, we present a method for structuring and searching protocol libraries such that the search space for protocols is reduced. Using a partial order over the outcomes of protocols, libraries are sorted into subsumption hierarchies, and a depth-first search is used to find all protocols that achieve a given goal. The deepening of the depth-first search can be terminated early if one of several conditions hold.

The results of the experimental analysis indicate that our approach has an average-case time complexity of $O(n)$, which is the same as a standard linear search but the measured time is significantly lower. However, this is offset by the cost of structuring the library as a subsumption hierarchy, which has an average-case time complexity $O(n^2)$, implying that the ratio of searches to insertions must be high to gain a benefit. The experiments demonstrated that using a depth-first search technique that starts at the leaf nodes of the subsumption hierarchy and traverses up the graph is consistently more efficient than any other search method assessed.

The work in this paper is another step towards achieving a vision of first-class protocols. However, before our full visions are realised, significant further work is required. In other work [20], [18], we have looked at issues such as verification of protocol specification, and of protocol compositions. In addition, meta-protocols are needed that allow agents to propose and negotiate which protocols are to be used, and suitable protocols for doing so will be investigated. To develop and test these ideas, we plan a prototype implementation in which agents negotiate the exchange of information using protocols specified using the RASA framework.

Applications of this work will arise wherever software agents need to consider the dynamic selection and invocation of protocols. One interesting application is in ecologies of marketplaces [21], where markets for the same or for closely-related products compete with one another for traders, as do, for example, the London, New York, NASDAQ and other Stock Exchanges. A marketplace is just a particular type of multi-agent interaction, and the rules of the market constitute an agent interaction protocol. In the case of competing marketplaces, traders develop strategies in response to particular market interaction rules, and market-owners change the rules of their markets in response to, or in anticipation of, trader behaviours and the rules of competing marketplaces. With electronic trading, one can readily envisage automated, run-time selection and adoption of marketplace rules (that is, of interaction protocols) by market owners in addition to the automated selection of strategies by traders; the two populations of markets and traders can be viewed as co-evolving [26]. These ideas have been developed through the CAT Market Design Tournament, established in 2007 to promote computer science research into automated and adaptive market interaction protocols [23].

Acknowledgments
The authors are grateful for financial support from the EC through the PIPS (IST-FP6-507019) project and from the UK EPSRC through the Market-Based Control of Complex Computational Systems project (GR/T10657/01).

References


