On Illegal Composition of First-Class Agent Interaction Protocols

Tim Miller **Peter McBurney**

Department of Computer Science,
University of Liverpool, Liverpool, L69 7ZF, UK
{tim, p.j.mcburney}@csc.liv.ac.uk

Abstract

In this paper, we examine the composition of first-class protocols for multi-agent systems. First-class protocols are protocols that exist as executable specifications that agents use at runtime to acquire the rules of the protocol. This is in contrast to the standard approach of hard-coding interaction protocols directly into agents — an approach that seems out of place with the goals of agents being intelligent and adaptive. In previous work, we have proposed a framework called $\mathcal{RASA}$, which regards protocols as first-class entities. $\mathcal{RASA}$ includes a formal, executable language for multi-agent protocol specification, which, in addition to specifying the order of messages using a process algebra, also allows designers to specify the rules and consequences of protocols using constraints. Rather than having hard-coded decision making mechanisms for choosing their next move, agents can inspect the protocol specification at runtime to do so. Such an approach would allow the agents to compose protocols at runtime, instead of relying on statically designed protocols. In this paper, we investigate the implications of *protocol composition* by examining the conditions under which composing existing legal protocols would lead to illegal protocols — that is, protocols that can fail during execution through no fault of the participants. We precisely define what constitutes an illegal protocol, and present proof obligations about compositions that, when discharged, demonstrate that a composition is legal.

Keywords: interaction protocols, multi-agent systems.

1 Introduction

Research into multi-agent systems aims to promote autonomy and intelligence into software agents. Intelligent agents should be able to interact socially with other agents, and adapt their behaviour to changing conditions. Despite this, research into interaction in 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 — therefore requiring agent code to 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.

In previous work [9, 10], we have proposed a framework called $\mathcal{RASA}$, which regards protocols as *first-class* entities. These first-class protocols are documents that exist within a multi-agent system, in contrast to hard-coded protocols, which exist merely as abstractions that emerge from the messages sent by the participants. To promote decoupling of agents from the protocols they use, we propose a formal, executable language for protocol specification. This language consists of a process algebra, used to specify the sequencing of messages. The messages are represented as atomic actions, and each atomic action contains the *rules* governing under which conditions the message can be sent, and the *effects* that sending the message has on a system. Rather than a protocol specification being just a sequence of arbitrary tokens, these rules and effects give the protocol *meaning*, and the rules and effects of the overall protocol can be derived compositionally from the meaning of the atomic protocols that comprise it. Instead of hard-coding the decision process of when to send messages, agent designers can implement goal-directed agents that reason about the effect of the messages they send and receive, and can choose the course of action that best achieves their goals, allowing agents to learn new protocols at runtime, and maintain libraries of protocols through which they can search to find the protocols that best achieve their goals. In addition, agents would be able to compose new protocols from existing protocols at runtime if they know of protocols that achieve their goals.

Composing protocols, whether statically at design time, or dynamically at runtime, is not an arbitrary task. Clearly, the composer must consider whether the composite protocol will go some way to achieving its goals. Determining this is a domain-specific (and most likely, agent-specific) problem, in which different agents will have different ideas within different environments or at different times as to whether a composite protocol is suitable.

However, there are certain conditions in which a protocol composition results in a protocol that can lead to a runtime error, which we categorise as illegal protocols. In this paper, we define what it means for a protocol to be illegal, and then identify the properties that must hold for protocols to be legal. We provide proof obligations for each type of composition that, when discharged, imply that a protocol is legal, and then identify how to verify a composition from existing legal protocols using these proof obligations. These proof obligations should be discharged every time a composition is performed, whether statically or dynamically. Emphasis is placed on protocols specified in the $\mathcal{RASA}$ protocol language, but such

ideas would be applicable to protocols specified in any language with features similar to $\mathcal{RASA}$'s.

The outline of this paper is as follows. Section 2 presents a brief overview of the $\mathcal{RASA}$ framework, and an example protocol specified using the $\mathcal{RASA}$ language. Section 3 formally defines our notion of an illegal protocol composition, and Section 4 presents the proofs the must be discharged to verify that a composition is illegal, and a method for discharging these. Section 5 presents related work and Section 6 concludes the paper.

2 $\mathcal{RASA}$ Overview

In this section, we present a brief overview of the $\mathcal{RASA}$ framework.

2.1 Modelling Information

Communication in multi-agent systems is performed across a *universe of discourse*. Agents send messages expressing particular properties about the universe. We assume that these messages refer to *variables*, which represent the parts of the universe that have changing values, and use other *tokens* to represent relations, functions, and constants to specify the properties of these variables and how they relate to each other.

Rather than devise a new language for expressing information, or using an existing language, we take the approach that any constraint language can be used to model the universe of discourse, provided that it has a few basic constants, operators and properties.

We use the definition of a *cylindric constraint system* proposed by De Boer *et al.* [2]. They define a cylindric constraint system as a complete algebraic lattice, $\langle C, \sqsupseteq, \sqcup, \text{true}, \text{false}, Var, \exists\rangle$. In this structure, C is the set of atomic propositions in the language, for example $X \leq Y$, $\sqsupseteq$ is an entailment operator, true and false are the least and greatest elements of C respectively, $\sqcup$ is the least upper bound operator, Var is a countable set of variables, and $\exists$ is an operator for hiding variables. The entailment operator defines a partial order over the elements in the lattice, such that $c \sqsupseteq d$ means that the information in d can be derived from c. The shorthand $c = d$ is equivalent to $c \sqsupseteq d$ and $d \sqsupseteq c$. We will use $\mathcal{L}$ to refer to the language, as well as the set of all constraints in the language; for example, $c \in \mathcal{L}$.

A constraint is one of the following: an atomic proposition, c, for example, $X = 1$, where X is a variable; a conjunction, $\phi \sqcup \psi$, where ϕ and ψ are constraints; or $\exists_x \phi$, where ϕ is a constraint and $x \in Var$. We extend this notation by allowing negation on the right of an entailment operator, for example, $c \sqsupseteq \neg d$, which is equivalent to $c \not\sqsupseteq d$. Other propositional operators are then defined from these: $\phi \vee \psi \mathrel{\widehat{=}} \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi \mathrel{\widehat{=}} \neg\phi \vee \psi$, and $\phi \leftrightarrow \psi \mathrel{\widehat{=}} \phi \rightarrow \psi \wedge \psi \rightarrow \phi$. We will continue to use the symbols ϕ and ψ to refer to constraints throughout this paper.

We introduce a renaming operator, which we will write as $[x/y]$, such that $\phi[x/y]$ means 'replace all references of y in ϕ with x'. The reader may have already noted that $\phi[x/y]$ is shorthand for $\exists_y(y = x \sqcup \phi)$.

2.2 Modelling Protocols

The $\mathcal{RASA}$ protocol specification language is based on process algebras, and resembles languages such as CSP [7]. However, we add the notion of *state* to the language. State is useful, because it allows us to build up the meaning of protocols compositionally, for example, the effect of sending two messages is the effect of sending the second in the state that results after sending the first. The final outcome of the protocol is the end state. A detailed presentation of the specification language, including operational semantics, is available in [10].

A protocol specification is a collection of protocol definitions of the format $N(x, \ldots, y) \mathrel{\widehat{=}} \pi$, in which $N, x, \ldots, y \in Var$, and π represents a protocol.

Protocols are defined using the two types of atomic protocol, and algebraic operators for building up compound protocols from these. The first atomic protocol is message sending, contains three parts, and is specified like so: $\psi \xrightarrow{c(i,j).\phi_m} \psi'$. This is read as follows: if the precondition, ψ, is provable from the current state, then the agent i is permitted to send the message ϕ_m, or any message ϕ'_m, such that $\phi'_m \sqsupseteq \phi_m$, to agent j. The effect of this message on the state is specified by the postcondition, ψ'. We allow agents to send the message ϕ'_m, such that $\phi'_m \sqsupseteq \phi_m$, so that agents can further constrain the values of the messages; thus, ϕ_m is only a template of the message. Omitting the prefix $c(i, j)$ from a message template implies that ϕ_m is an action that must be performed. In this paper, we omit (i, j) when we do not care who the sender and receiver of the message is. We use the notion of *inertia* in calculating the new state from the postcondition; that is, any variables in ψ' are constrained by ψ' in the new state, and any other variables in the state are left unchanged. The second atomic action/protocol is the empty action, the syntax of which is $\psi \rightarrow \epsilon$. This specifies that if the precondition ψ is provable from the current state, then no message sending is required.

Compound protocols can be built up from these atomic protocols. If π_1 and π_2 are two protocols, then the following are also protocols: the protocol $\pi_1; \pi_2$, which represents sequential concatenation, such that π_1 is executed, followed by π_2; the protocol $\pi_1 \cup \pi_2$, which represents a choice between π_1 and π_2; and the protocol $\mathbf{var}_x^{\psi} \cdot \pi_1$, which is a protocol the same as π_1, except that a local variable x is available over the scope of π_1, but with the constraints ψ on x remaining unchanged throughout that scope. Any variable x already in the state is out of scope until π_1 finishes executing. In addition, $\mathcal{RASA}$ supports the referencing of protocols via their names. That is, for a protocol definition $N(x) \mathrel{\widehat{=}} \pi_1$, one can reference this from within another protocol using $N(y)$, where $y \in Var$.

A key feature of this language is that it has the same syntax and semantics at all dialogue levels. Single messages are themselves protocols, and the syntax and semantics for composing two atomic protocols is the same for composing two other composite protocols. Thus, individual utterances, sequences of utterances, protocols, and combinations of protocols can all be reasoned-over, modified, composed and invoked by agents participating in an interaction using the same reasoning mechanism.

Example 2.1. We present a small example of a simple interaction in which an agent, A, proposes that another agent, B, commits to P, and B can accept or refuse this proposal.

The semantics of $\mathcal{RASA}$ is compositional, so it makes sense to present the protocol in a bottom-up manner. First, we define the $Prop$ protocol, an atomic protocol which models A sending the proposal to B:

$$Prop(A, B, P) \mathrel{\widehat{=}} \text{true} \xrightarrow{c(A,B).propose(P)} prop(P)$$

The postcondition $prop(P)$ simply indicates that the current proposal is P. The Acc and Rej protocols

model B accepting or rejecting the proposal respectively:

$$Acc(A,B) \mathrel{\widehat{=}} prop(P) \xrightarrow{c(B,A).accept(P)} cmt(B,A,P)$$
$$Rej(A,B) \mathrel{\widehat{=}} prop(P) \xrightarrow{c(B,A).reject(P)} \text{true}$$

The notation $cmt(B,A,P)$ is a constraint representing B's commitment to perform P for the creditor A.

Finally, we compose these three atomic protocols together into a composite protocol, which defines the order that the messages must occur:

$$Prot(A,B,P) \mathrel{\widehat{=}} Prop; (Acc \sqcup Ref)$$

This definition enforces the condition that the proposal must be sent by A before B can accept or reject it, and that B can only send either an accept or reject, but not both. In addition, if the path $Prop; Acc$ is taken, then B is committed to P.

One can see that, provided an agent understands the meaning of $cmt(B,A,P)$, such a protocol can be reasoned about at runtime. Firstly, agent A decides to use this protocol because it calculate that $cmt(B,A,P)$ is an outcome. If B agrees to using the protocol, then, after it receives the proposal, it can reason that accepting the proposal will lead to the state in which it is committed to performing B. If it does not accept, then there is no change, so it can decide its reply by analysing its goals and assessing their compatibility with the outcomes.

3 Illegal Protocols and Compositions

We define *composition* of a protocol as the process of either deriving a new atomic protocol, or taking one or more existing protocol definitions, and forming a larger compound protocol using the algebraic operators of the $\mathcal{RASA}$ specification language. For example, taking protocols π_1 and π_2, the composite protocol $\pi_1; \pi_2$ can be defined. The vision of first-class protocols includes agents composing their own protocols in this way, at runtime, if they do not have a protocol in their library that achieves their goals.

Clearly, there are verification issues regarding protocol composition. Some of these issues relate to the messages that are sent, the order that are sent in, and when they can be sent. Others are related to the rules and outcomes, such as whether they are correct with respect to an informal definition. Such issues are *domain specific*, and in fact, they may even be *agent specific*. In contrast, the purpose of this paper is to study *generic* properties of protocols. That is, issues regarding the relationship between protocols, and most specifically, the conditions under which they can be composed. Such properties would apply to every protocol, regardless of the domain.

In this section, we define and discuss one such generic property, which relates to protocol composition. We call this property *stuckness*. Stuckness is a property that should not be exhibited for any agent interaction protocol, and we assert that any protocol to be used should first be proved to be *stuckness free*.

Definition 3.1. Protocol Stuckness

The execution of a protocol becomes stuck if:

1. it is not terminated; and
2. it is either in a state from which no transition can be made, or the state is equivalent to false.

That is, at the current state, the rules of the protocol are such that no move can be made by any participant, or the current state contains an unsatisfiable constraint. We say that any protocol that can become stuck suffers from *stuckness*. Stuckness refers to some form of runtime error, in which, agents can execute part of a protocol, but then come to a point at which they are unable to continue executing. Such a property is an undesirable property, and one which we want to prove is not possible for any protocol[1].

Example 3.1. As an example of stuckness, consider the following protocol definition:

$$N(x) \mathrel{\widehat{=}} x = 1 \xrightarrow{c.a(x)} x = 2; x = 3 \xrightarrow{c.b(x)} x = 4$$

After $a(x)$ is sent across the channel, the state is $x = 2$. However, $x = 2$ does not entail the precondition $x = 3$, so the protocol execution becomes stuck.

To help us formally define stuckness for $\mathcal{RASA}$ protocol specifications, we first introduce some auxiliary definitions.

Definition 3.2. Local Constraints

Local constraints are the constraints on locally declared variables — that is, variables declared using the form $\mathbf{var}_x^{\psi} \cdot \pi$. Consider the following example protocol, which increments the values of x and y, provided that x is less than 10:

$$\mathbf{var}_{x_0,y_0}^{x_0=x \sqcup y_0=y} \cdot (x < 10 \xrightarrow{c.a(x)} x = x_0 + 1 \sqcup y = y_0 + 1)$$

If this is executed in the state $x = 0 \sqcup y = 5$, then the local constraints over the scope of the atomic protocol would be $x_0 = 0 \sqcup y_0 = 5$.

Throughout this section, when we discuss protocols, we will assume that they are evaluated under local constraints. If a protocol is not contained within the scope of a variable declaration, the local constraints are equivalent to the constraint 'true'. Further discussion of this is presented later in Section 4.1 at a point when their purpose should be clearer.

Definition 3.3. Weakest Explicit Precondition

The *weakest explicit precondition* of a protocol is the weakest (or most general) constraint that satisfies the precondition of the protocol. The weakest *explicit* precondition is in contrast to the weakest *calculated* precondition, which is the weakest constraint from which a protocol cannot become stuck. The terminology is used because the weakest calculated precondition must be computed taking into account the entire protocol, whereas the explicit precondition is taken as the disjunction of the preconditions of all of the atomic protocols that can be executed as the first step in the protocol. For example, consider the following protocol:

$$Q(x) \mathrel{\widehat{=}} x = 0 \xrightarrow{c.a(x)} x = 1; y = 0 \xrightarrow{c.b(x)} y = 1$$

The weakest explicit precondition of this protocol is $x = 0$, because $x = 0$ is the most general constraint under which the protocol can begin to execute. However, the calculated precondition is $x = 0 \sqcup y = 0$, because $y = 0$ must hold for the protocol to execute fully due to the fact that it is the precondition of the second sub-protocol, and that the first sub-protocol does not ensure this.

We define a function, $pre \in \pi \times \mathcal{L} \to \mathcal{L}$, which, for a protocol, π and local constraints, $\mathcal{L}$, returns the weakest *explicit* precondition of π. pre is defined formally as follows:

[1] However, we note that there may be situations in which an agent desires an interaction to become stuck, e.g. to distract a competitor, as described in [6].

$$
\begin{array}{lcl}
pre(\psi \rightarrow \epsilon, \chi) & = & \psi \sqcup \chi \\
pre(\psi \xrightarrow{c.\phi_m} \psi', \chi) & = & \psi \sqcup \chi \\
pre(\pi_1; \pi_2, \chi) & = & pre(\pi_1, \chi) \\
pre(\pi_1 \cup \pi_2, \chi) & = & pre(\pi_1, \chi) \vee pre(\pi_2, \chi) \\
pre(\mathbf{var}_x^{\psi} \cdot \pi, \chi) & = & \exists_x pre(\pi, \exists_x \chi \sqcup \psi) \\
pre(N(x), \chi) & = & pre(\pi[x/y], \chi) \\
 & & \text{where } N(y) \mathrel{\widehat{=}} \pi \in D
\end{array}
$$

We discuss these definitions briefly. The explicit preconditions of an empty or atomic protocol is the explicit precondition, ψ, conjoined with the local constraints, χ. The explicit precondition of a sequential composition, $\pi_1; \pi_2$, is the precondition of π_1. For a choice, $\pi_1 \cup \pi_2$, it is the disjunction of the preconditions of π_1 and π_2. The explicit precondition of a variable declaration, $\mathbf{var}_x^{\psi} \cdot \pi$, is the precondition of π, but including the local constraints, ψ, and with x hidden from χ so the local declaration is not confused with any already-declared x. References to x in the precondition of π are then hidden, because x is local. Finally, the explicit precondition of a reference name protocol is the precondition of the corresponding definition with the parameters appropriately renamed. We omit the local constraints parameter if it is unnecessary.

Definition 3.4. Maximal Calculated Postcondition

The *maximal calculated postcondition* (or simply maximal postcondition) of a protocol is the most general constraint such that every end state satisfies that constraint, and and that it is satisfied only by those end states. We use the term *calculated* because, unlike the precondition, the calculated postcondition requires one to calculate the end state over entire traces of protocols.

The maximal calculated postcondition is defined as a function, $post \in (\pi \times \mathcal{L} \times \mathcal{L}) \rightarrow \mathcal{L}$, which, for a protocol, initial state, and local constraints, returns the constraint that satisfies all postconditions, and only those postconditions. This can be computed on a syntactic level using the following definitions:

$$
\begin{array}{lcl}
post(\psi \rightarrow \epsilon, \phi, \chi) & = & \phi \\
post(\psi \xrightarrow{c.\phi_m} \psi', \phi, \chi) & = & \exists_{free(\phi')} \phi \sqcup \phi' \\
 & & \text{where } \phi' = \psi' \sqcup \phi_m \sqcup \chi \\
post(\pi_1; \pi_2, \phi, \chi) & = & post(\pi_2, post(\pi_1, \phi, \chi)) \\
post(\pi_1 \cup \pi_2, \phi, \chi) & = & post(\pi_1, \phi, \chi) \vee \\
 & & post(\pi_2, \phi, \chi) \\
post(\mathbf{var}_x^{\psi} \cdot \pi, \phi, \chi) & = & \exists_x post(\pi, \exists_x \phi, \exists_x \chi \sqcup \psi') \\
post(N(x), \phi, \chi) & = & post(\pi[x/y], \phi, \chi) \\
 & & \text{where } N(y) \mathrel{\widehat{=}} \pi \in D
\end{array}
$$

We comment on this definition briefly. The maximal postcondition of an empty protocol executed from a pre-state is the pre-state, because it does not change. For an atomic protocol, $\psi \xrightarrow{c.\phi_m} \psi'$, the maximal postcondition is the weakest constraint, ϕ', that satisfies the explicit postcondition, ψ', the message ϕ_m, and the local constraints χ. This is clear from the definition in [9], in which the postcondition is equivalent to this, except it is constrained by ϕ'_m instead of ϕ_m, such that $\phi'_m \sqsupseteq \phi_m$, allowing agents to constrain messages. However, if we are calculating the most general constraint, then ϕ_m is the most general message constraint, so we use this rather than any further constrained message. In addition, we conjoin this constraint with the pre-state, ϕ, but with all free variables in ϕ' hidden from ϕ.

The maximal postcondition of the sequential composition $\pi_1; \pi_2$ under the state ϕ is the maximal postcondition of π_2 under the initial state $post(\pi_1, \phi, \chi)$. That is, the maximal postcondition from executing π_2 under the maximal postcondition of π_1 under ϕ. The maximal postcondition of the choice protocol $\pi_1 \cup \pi_2$ is the disjunction of the two protocol's maximal postconditions, because one of these postconditions will hold after executing the choice. The maximal postcondition of a variable declaration $\mathbf{var}_x^{\psi} \cdot \pi$ is the maximal precondition of π under the state $\exists_x \phi$ and with local constraints $\exists_x \chi \sqcup \psi'$. References to x in ϕ and χ are hidden because any references already declared are out of scope. The maximal precondition of the referenced name $N(x)$ is the maximal precondition of the corresponding protocol π, with the parameters renamed. We omit the local constraints parameter if it is unnecessary.

Example 3.2. Refer to the definition of the protocol $Q(x)$ from above. The weakest precondition of the protocol is determined as follows:

$$
\begin{array}{ll}
 & pre(Q(x)) \\
\equiv & pre(x = 0 \xrightarrow{c.a(x)} x = 1; y = 0 \xrightarrow{c.b(x)} y = 1) \\
\equiv & pre(x = 0 \xrightarrow{c.a(x)} x = 1) \\
\equiv & x = 0
\end{array}
$$

The maximal postcondition is determined using the definition of *post*. The overall maximal postcondition of a protocol is the maximal postcondition with the weakest precondition as the initial state:

$$
\begin{array}{ll}
 & post(Q(x), pre(Q(x))) \\
\equiv & post(Q(x), x = 0) \\
\equiv & post(x = 0 \xrightarrow{c.a(x)} x = 1; \\
 & \qquad y = 0 \xrightarrow{c.b(y)} y = 1, x = 0) \\
\equiv & post(y = 0 \xrightarrow{c.b(y)} y = 1, \\
 & \qquad post(x = 0 \xrightarrow{c.a(x)} x = 1), x = 0) \\
\equiv & post(y = 0 \xrightarrow{c.b(y)} y = 1, x = 1) \\
\equiv & y = 1 \sqcup x = 1
\end{array}
$$

Definition 3.5. Formal Definition of Stuckness

Definition 3.1 presented an informal definition of stuckness for a protocol. The notion of stuckness, and its related proof obligations, is similar to the way in which preconditions are verified in model-oriented specification languages such as Z [14] and B [13]. However, in these methods, the proof is generally of the form: $pre \rightarrow \exists post$. That is, the proof is that if the precondition holds, there exists at least one poststate that satisfies the postcondition.

This definition is not enough to prove that a protocol is stuckness free. Take for example, the following protocol:

$$
\begin{array}{l}
R(x) \mathrel{\widehat{=}} (x = 0 \xrightarrow{c.a(x)} x = 1 \ \cup \ x = 0 \xrightarrow{c.b(x)} x = 2); \\
\qquad\qquad x = 1 \xrightarrow{c.d(x)} x = 10
\end{array}
$$

This defines a choice followed by an atomic protocol. The precondition for both options in the choice is $x = 0$, therefore, the overall precondition is $x = 0$. If the protocol is in a state in which the precondition holds, then the left option can be chosen, resulting in the state $x = 1$. The precondition of the next protocol is $x = 1$, so execution can complete. However, if the right option is chosen, resulting in the state $x = 2$, then the protocol becomes stuck. So, even though the precondition of this protocol implies that there is a postcondition (take the first option of the choice), the protocol can still become stuck. Therefore, an alternate definition to the $pre \rightarrow \exists post$ definition is needed.

We formally define stuckness as a relation called *stuck*, in which, for a protocol π, an initial state ϕ, and local constraints χ, $stuck(\pi, \phi, \chi)$ is true if and only if π can become stuck from the starting state ϕ under the local constraints χ. Devising a neat, formal definition of stuckness for any arbitrary protocol, like the $pre \rightarrow \exists post$ from above, does not seem possible, so instead, we define it such that each protocol operator has its own definition, each shown in Table 1.

This definition is not immediately obvious, so we spend some time discussing it. The empty protocol, $\psi \rightarrow \epsilon$, can only become stuck if its precondition is not enabled. [9] specifies the denotational semantics of an atomic protocol, $\psi \xrightarrow{c.\phi_m} \psi'$, as follows[2], assuming that ϕ is the pre-state and χ the local constraints:

$$\exists \phi'_m, \phi' \bullet (\phi \sqcup \chi \sqsupseteq \psi) \wedge (\phi'_m \sqsupseteq \phi_m) \wedge \phi' = (\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi)$$
where v is the free variables in $\psi' \sqcup \phi'_m \sqcup \chi$

Here, ϕ'_m represents the message, and ϕ' the post-state. So, this definition says that $\phi \sqcup \chi$ must satisfy the precondition, the message ϕ'_m must be a refinement of the message template ϕ_m, and the post-state is the postcondition, ψ', conjoined with the information from the message, ϕ'_m, conjoined with the local constraints, χ (which must hold after the execution), and finally, conjoined with $\exists_v \phi$, in which v is the set of variables that this protocol changed, therefore $\exists_v \phi$ is the information from the pre-state ϕ that is not overridden by the postcondition.

An atomic protocol is stuck under an initial state ϕ if it defines no behaviour. So, for the pre-state ϕ, this is equivalent to negating the definition from above:

$$\begin{aligned} & \neg \exists \phi'_m, \phi' \bullet (\phi \sqcup \chi \sqsupseteq \psi) \wedge \\ & \quad (\phi'_m \sqsupseteq \phi_m) \wedge \phi' = (\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi) \\ \equiv \quad & \forall \phi'_m, \phi' \bullet (\phi \sqcup \chi \not\sqsupseteq \psi) \vee \phi'_m \not\sqsupseteq \phi_m \vee \\ & \quad \phi' \neq (\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi) \end{aligned}$$

If for every ϕ'_m, $\phi'_m \not\sqsupseteq \phi_m$, that must mean that ϕ_m is equivalent to false, and therefore ϕ_m is unsatisfiable. Otherwise, if there is a ϕ'_m such that $\phi'_m \sqsupseteq \phi_m$, but for every ϕ', $\phi' \neq (\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi)$, this must mean that $\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi$ is unsatisfiable. Therefore, the above predicate becomes:

$$\begin{aligned} & \forall \phi'_m \bullet (\phi \sqcup \chi \not\sqsupseteq \psi) \vee \phi_m \sqsupseteq \text{false} \vee \\ & \quad (\phi'_m \sqsupseteq \phi_m \rightarrow (\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi) \sqsupseteq \text{false}) \end{aligned}$$

If for every ϕ'_m such that $\phi'_m \sqsupseteq \phi_m$ (including ϕ_m itself), $(\psi' \sqcup \phi'_m \sqcup \chi \sqcup \exists_v \phi) \sqsupseteq$ false, then we can substitute ϕ_m for ϕ'_m in this predicate (removing the quantification), because ϕ_m is the most general case of ϕ'_m:

$$\begin{aligned} & (\phi \sqcup \chi \not\sqsupseteq \psi) \vee \\ & \phi_m \sqsupseteq \text{false} \vee (\psi' \sqcup \phi_m \sqcup \chi \sqcup \exists_v \phi) \sqsupseteq \text{false} \end{aligned}$$

Clearly, if $\phi_m \sqsupseteq$ false then also $(\psi' \sqcup \phi_m \sqcup \chi \sqcup \exists_v \phi) \sqsupseteq$ false, so this first case is not necessary. Using the definition of implication, we are left with the following:

$$(\phi \sqcup \chi \sqsupseteq \psi) \rightarrow (\psi' \sqcup \phi_m \sqcup \chi \sqcup \exists_v \phi) \sqsupseteq \text{false}$$

However, $\exists_v \phi$ contains no information about the other variables in the post-state, because these variables are hidden by $\exists_v$, so conjoining it to the postcondition cannot result in an unsatisfiable constraint. We assume that ϕ is satisfiable (because it is the post-state of another protocol), we can remove it from the proof obligation. This leaves us with the following:

$$(\phi \sqcup \chi \sqsupseteq \psi) \rightarrow (\psi' \sqcup \phi_m \sqcup \chi) \sqsupseteq \text{false}$$

[2]The definition from [9] is in fact defined as a set of pre- and post-states, however, the definition presented here will suffice for discussion.

Therefore, an atomic protocol becomes stuck if the precondition is not enabled by the current state, or if the postcondition (including any constraints specified by local variable declarations) and message are inconsistent with each other, which is an intuitive definition of stuckness.

The case for the sequentially composed protocol $\pi_1; \pi_2$ is straightforward; it is stuck under a pre-state ϕ if either π_1 is stuck under ϕ, or if, once π_1 has executed, π_2 is stuck under the end-state of π_1.

This case for a choice is less straightforward than it may first appear. An initial attempt to model stuckness for a choice protocol may be to specify that the protocol $\pi_1 \cup \pi_2$ is stuck under a state ϕ if and only if π_1 and π_2 are both stuck under ϕ. This is certainly true for the 'if', but not for the 'only if' — we are aiming to verify that protocol can never become stuck, so we have to consider the case in which one of the protocols is not stuck, so can execute to its end, but the other may be, so the composite protocol can still get stuck. An initial attempt to model this may be to specify that $\pi_1 \cup \pi_2$ is stuck in a state ϕ if and only if π_1 or π_2 is stuck in ϕ. However, this does not consider the case in which the precondition of only one protocol is enabled, which does not indicate a stuck protocol, because the other protocol may still be executed in another state that satisfies its precondition.

Considering all these cases, we have the following three cases in which the protocol $\pi_1 \cup \pi_2$ is stuck under ϕ:

- the precondition of the protocol is not enabled: $\phi \not\sqsupseteq pre(\pi_1 \cup \pi_2)$;
- the precondition of the protocol π_1 is enabled, but π_1 is stuck under ϕ: $\phi \sqsupseteq pre(\pi_1) \wedge stuck(\pi_1, \phi)$; and
- the precondition of the protocol π_2 is enabled, but π_2 is stuck under *phi*: $\phi \sqsupseteq pre(\pi_2) \wedge stuck(\pi_2, \phi)$.

Therefore, $stuck(\pi_1 \cup \pi_2, \phi)$ is true if and only if any of the above three conditions holds.

A variable declaration protocol, $\mathbf{var}_x^{\psi} \cdot \pi$, is stuck under ϕ if and only if the sub-protocol π is stuck under ϕ once the locals constraints on x are taken into consideration. This is straightforward to derive from the semantics from [9] — $\mathbf{var}_x^{\psi} \cdot \pi$ is stuck in state ϕ and local constraints χ if and only if π is stuck under the state $\exists_x \phi$ and local constraints $\exists_x \chi \sqcup \psi$.

Finally, the referenced name $N(x)$ is stuck in ϕ if either N is not a name in the protocol specification, or if the protocol, π, corresponding to the definition of $N(y)$, becomes stuck in ϕ when variable y is renamed to x.

4 Verification of Protocols

In this section, we discuss how to prove that a protocol is stuckness free. Proofs are performed inductively over the structure of the protocols, and are divided into two cases: the first proving that there exists at least one state that satisfies the precondition of the protocol; and the second proving that for every state that does satisfy the precondition, executing the protocol from this state does not result in it becoming stuck.

4.1 Proof Obligations

The first proof obligation is showing that an initial state, other than false, exists. This is formally defined as follows:

$stuck(\psi \rightarrow \epsilon, \phi, \chi)$	iff	$\phi \sqcup \chi \not\sqsupseteq \psi$
$stuck(\psi \xrightarrow{c.\phi_m} \psi', \phi, \chi)$	iff	$(\phi \sqcup \chi \sqsupseteq \psi) \rightarrow (\phi_m \sqcup \psi' \sqcup \chi \sqsupseteq \text{false})$
$stuck(\pi_1; \pi_2, \phi, \chi)$	iff	$stuck(\pi_1, \phi, \chi) \vee \exists \phi' \in \mathcal{L} \bullet \phi' \sqsupseteq post(\pi, \phi, \chi) \rightarrow stuck(\pi_2, \phi', \chi)$
$stuck(\pi_1 \cup \pi_2, \phi, \chi)$	iff	$\phi \not\sqsupseteq pre(\pi_1 \cup \pi_2, \chi) \vee$ $\phi \sqsupseteq pre(\pi_1, \chi) \wedge stuck(\pi_1, \phi, \chi) \vee$ $\phi \sqsupseteq pre(\pi_2, \chi) \wedge stuck(\pi_2, \phi, \chi)$
$stuck(\mathbf{var}_x^{\psi} \cdot \pi, \phi, \chi)$	iff	$stuck(\pi, \exists_x \phi, \exists_x \chi \sqcup \psi)$
$stuck(D, N(x), \phi, \chi)$	iff	$N(y) \mathrel{\widehat{=}} \pi \notin D \vee stuck(\pi[x/y], \phi, \chi)$

Table 1: Formal Definition of Stuckness

$$\exists \phi \in \mathcal{L} \setminus \{\text{false}\} \bullet \phi \sqsupseteq pre(\pi, \chi)$$

Clearly, if there is no ϕ that satisfies the precondition of π, then the precondition is equivalent to false. Therefore, this proof obligation can be reduced to the following:

$$pre(\pi, \chi) \not\sqsupseteq \text{false}$$

This proof obligation is called the *initialisation proof*. The second case, proving that any states satisfying the precondition do not lead to the protocol to becoming stuck, is formally defined as follows:

$$\forall \phi \in \mathcal{L} \bullet \phi \in pre(\pi, \chi) \rightarrow \neg stuck(\pi, \phi, \chi)$$

Therefore, we are proving that any constraint that satisfies the precondition of a protocol cannot lead it to become stuck. However, this is equivalent to proving that π is not stuck for the most general case; that is, the weakest explicit precondition. Therefore, the following proof obligation will suffice:

$$\neg stuck(\pi, pre(\pi, \chi), \chi)$$

This proof obligation is called the *stuckness proof*.

In this section, we investigate methods for proving these properties for the different protocol operators. In addition, we make the assumption that structural induction is used to prove the legality of a protocol. Structural induction is a proof method in which one proves a property for the elements that make up a structure before proving the property for the structure itself. In the case of $\mathcal{RASA}$ protocols, the induction is performed over the protocol operators, with atomic protocols and the empty protocol as base cases. Our assumption seems reasonable given that the $\mathcal{RASA}$ framework is designed to encourage agents to compose protocols at runtime, and the protocols from which the new composite protocols are composed should already be verified as stuckness-free, so the inductive step will have already been performed.

However, proving that the sub-protocols are stuckness free does not necessarily prove it for the compound protocol. For example, consider the protocol $\pi_1; \pi_2$ — we have already established that all post-states of π_1 must satisfy the precondition of π_2, therefore, proving the legality of π_1 and π_2 is not enough to establish the legality of the composite protocol. In this section, we identify the additional proof obligations that must be discharged on top of the structural induction to prove that protocols are stuckness free.

Remark 1. Regarding Local Constraints

So far in this section, we have defined the weakest precondition and postcondition with respect to local constraints. It must be noted that we make some assumptions about the local constraints. Firstly, the reader may have noted that we define the weakest precondition of an atomic protocol, $\psi \xrightarrow{c.\phi_m} \psi'$, as $\psi \sqcup \chi$, in which χ represents the constraints on local variables. However, we do not define from where χ is derived.

We assume that a protocol is proved correct only for the local constraints under which is used. Therefore, one must collect the local constraints as they inductively move down the structure of the protocol. For example, consider the following protocol, which increments the value of the variables x and y, provided $x < 10$ holds before hand:

$$\mathbf{var}_{x_0}^{x_0=x} \cdot \mathbf{var}_{y_0}^{y_0=y} \cdot x < 10 \xrightarrow{c.\phi_m} x = x_0 + 1 \sqcup y = y_0 + 1$$

To evaluate the precondition of this protocol, one would assume that there are no local constraints. Then, the inner variable declaration, $\mathbf{var}_{y_0}^{y_0=y} \cdot \ldots$ is evaluate under the local constraints $x_0 = x$, and the inner atomic protocol is evaluated under the local constraints $x_0 = x \sqcup y_0 = y$. Therefore, the weakest explicit precondition of the inner atomic protocol is the precondition, $x < 10$, conjoined with the local constraints, to get $x < 10 \sqcup x_0 = x \sqcup y_0 = y$, which simplifies to $x < 10 \sqcup x_0 < 10 \sqcup y_0 = y$.

4.1.1 Empty Protocol

From the definitions in Section 3, we know that the empty protocol can only become stuck if its precondition does not hold. For the initialisation proof, we substitute in an empty protocol for π from the generic proof obligation and simplify:

$$\begin{array}{ll} & pre(\psi \rightarrow \epsilon, \chi) \not\sqsupseteq \text{false} \\ \equiv & \psi \sqcup \chi \not\sqsupseteq \text{false} \qquad \text{from defn. of } pre \end{array}$$

Therefore, to discharge the initialisation proof for an empty protocol, one must simply prove that its weakest precondition, $\psi \sqcup \chi$ in this instance, is satisfiable.

To prove the absence of stuckness in a protocol, we substitute in an empty protocol for π from the generic proof obligation and simplify:

$$\begin{array}{ll} & \neg stuck(\psi \rightarrow \epsilon, pre(\psi \rightarrow \epsilon, \chi), \chi) \\ \equiv & \neg(\psi \sqcup \chi \not\sqsupseteq \psi) \quad \text{from defn. of } stuck \\ \equiv & \psi \sqcup \chi \sqsupseteq \psi \qquad \text{double negation} \end{array}$$

This final line is trivially true, so we conclude that no proof obligation is necessary for the stuckness proof.

4.1.2 Atomic Protocols

Atomic protocols, along with the empty protocol, are the base cases of the inductive proof — that is, there are no sub-protocols that we must verify before proving this protocol is legal.

For the initialisation proof of atomic protocols, we substitute in an atomic protocol for π from the generic proof obligation and simplify:

$$\begin{array}{ll} & pre(\psi \xrightarrow{c.\phi_m} \psi', \chi) \not\sqsupseteq \text{false} \\ \equiv & \psi \sqcup \chi \not\sqsupseteq \text{false} \qquad \text{from defn. of } pre \end{array}$$

Therefore, to discharge the initialisation proof for an atomic protocol, one must simply prove that its weakest precondition, $\psi \sqcup \chi$ in this instance, is satisfiable.

To prove the absence of stuckness in a protocol, we substitute in an atomic protocol for π from the generic proof obligation and simplify:

$$\begin{aligned}
& \neg stuck(\psi \xrightarrow{c.\phi_m} \psi', pre(\psi \xrightarrow{c.\phi_m} \psi', \chi), \chi) \\
\equiv\ & \neg stuck(\psi \xrightarrow{c.\phi_m} \psi', \psi \sqcup \chi, \chi) \\
& \text{from defn. of } pre \\
\equiv\ & \neg(\psi \sqcup \chi \sqsupseteq \psi \rightarrow (\psi' \sqcup \phi_m \sqcup \chi \sqsupseteq \text{false})) \\
& \text{from defn. of } stuck \\
\equiv\ & \psi \sqcup \chi \not\sqsupseteq \psi \vee (\psi' \sqcup \phi_m \sqcup \chi) \not\sqsupseteq \text{false} \\
& \text{from de Morgan's laws} \\
\equiv\ & (\psi' \sqcup \phi_m \sqcup \chi) \not\sqsupseteq \text{false} \\
& \text{because } \psi \sqcup \chi \sqsupseteq \psi \text{ for any } \psi, \chi
\end{aligned}$$

Therefore, to discharge the stuckness proof obligation, one must prove that there exists a postcondition from the weakest precondition, by proving that $\psi' \sqcup \phi_m \sqcup \chi$ is satisfiable.

4.1.3 Sequential Composition

For the initialisation proof of sequential composition protocols, we substitute in a sequential composition for π from the generic proof obligation and simplify:

$$\begin{aligned}
& pre(\pi_1; \pi_2) \not\sqsupseteq \text{false} \\
\equiv\ & pre(\pi_1) \not\sqsupseteq \text{false} \qquad \text{from defn. of } pre
\end{aligned}$$

Therefore, to discharge the initialisation proof obligation for a sequential composition, $\pi_1; \pi_2$, one must discharge the proof obligation for π_1. Assuming an inductive proof over the structure of $\pi_1; \pi_2$, this will have already been discharged, therefore, such a proof is not necessary.

To prove the absence of stuckness in a protocol, we substitute in a sequential composition for π from the generic proof obligation and simplify:

$$\begin{aligned}
& \neg stuck(\pi_1; \pi_2,\ pre(\pi_1; \pi_2,\ \chi),\ \chi) \\
\equiv\ & \neg stuck(\pi_1; \pi_2, pre(\pi_1, \chi),\ \chi) \\
& \text{from defn. of } pre \\
\equiv\ & \neg(stuck(\pi_1, pre(\pi_1, \chi),\ \chi) \vee \\
& \quad stuck(\pi_2, post(\pi_1, pre(\pi_1, \chi), \chi), \chi) \\
& \text{from defn. of } stuck \\
\equiv\ & \neg stuck(\pi_1, pre(\pi_1, \chi),\ \chi) \wedge \\
& \neg stuck(\pi_2, post(\pi_1, pre(\pi_1, \chi), \chi), \chi) \\
& \text{from de Morgan's laws}
\end{aligned}$$

Assuming an inductive proof over the structure of $\pi_1; \pi_2$, the stuckness proof obligations for π_1 and π_2 would have already been discharged. Therefore, $\neg stuck(\pi_1, pre(\pi_1, \chi),\ \chi)$ need not be proved again. However, protocol π_2 would have been proved to be stuckness free only for its precondition, so we need to prove that every postcondition of π_1 satisfies the precondition of π_2. We can do this by proving that the maximal postcondition of π_1, $post(\pi_1, pre(\pi_1, \chi)$, is a stronger constraint than the precondition of π_2:

$$post(\pi_1, pre(\pi_1, \chi), \chi) \sqsupseteq pre(\pi_2, \chi)$$

Therefore, to prove that sequentially composed protocol, $\pi_1; \pi_2$, is stuckness free, one must prove that the postcondition of π_1, under its weakest precondition, implies weakest precondition of π_2.

We also include an additional proof obligation for special case for sequential composition: protocols of the form $\pi_1; (\pi_2 \cup \pi_3)$. In such protocols, it is possible that the protocol is not stuck, however, that one of π_2 or π_3 is never enabled. For example:

$$\begin{aligned}
N(x) \mathrel{\widehat{=}}\ & x = 1 \xrightarrow{c.a(x)} x = 2; \\
& (x = 2 \xrightarrow{c.b(x)} x = 3 \cup x \neq 2 \xrightarrow{c.d(x)} x = 4)
\end{aligned}$$

In this example, the postcondition $x = 2$ enables the left side of the choice every time, but never the right hand side. While this does not imply stuckness, because the left case is always enabled, it does mean that the right hand side is unnecessary. Therefore, in this case, we prove that $\pi_1; \pi_2$ and $\pi_1; \pi_3$ are both stuckness free. This seems reasonable, because

$$\pi_1; (\pi_2 \cup \pi_3) \quad = \quad \pi_1; \pi_2 \cup \pi_1; \pi_3$$

for any π_1, π_2, and π_3, therefore, it would be inconsistent for the protocol on the right side of the equality to be declared stuck, while the protocol on the left side is not. Similarly protocols of the format $(\pi_1 \cup \pi_2); \pi_3$ can be broken into the two cases $\pi_1; \pi_3$ and $\pi_2; \pi_3$. This is strictly not necessary, but it may make a proof of such a property more straightforward.

4.1.4 Choice

For the initialisation proof of a choice protocol, we substitute in a choice for π from the generic proof obligation and simplify:

$$\begin{aligned}
& pre(\pi_1 \cup \pi_2) \not\sqsupseteq \text{false} \\
\equiv\ & pre(\pi_1) \vee pre(\pi_2) \not\sqsupseteq \text{false}
\end{aligned}$$

Therefore, to discharge the initialisation proof obligation for a choice, $\pi_1 \cup \pi_2$, one must prove that one of π_1 or π_2 has an initialisation state. Assuming an inductive proof over the structure of $\pi_1 \cup \pi_2$, this will have already been discharged for both π_1 and π_2, therefore, it trivially holds for either, and this proof is not necessary.

To prove the absence of stuckness in a choice protocol, we substitute in a choice for π from the generic proof obligation and simplify:

$$\begin{aligned}
& \neg stuck(\pi_1 \cup \pi_2,\ pre(\pi_1 \cup \pi_2),\ \chi) \\
\equiv\ & \neg stuck(\pi_1 \cup \pi_2,\ pre(\pi_1) \vee pre(\pi_2),\ \chi) \\
& \text{from defn. of } pre \\
\equiv\ & \neg\big((pre(\pi_1) \vee pre(\pi_2) \not\sqsupseteq pre(\pi_1) \vee pre(\pi_2)) \vee \\
& \quad pre(\pi_1) \vee pre(\pi_2) \sqsupseteq pre(\pi_1) \wedge \\
& \quad\quad stuck(\pi_1, pre(\pi_1) \vee pre(\pi_2), \chi) \vee \\
& \quad pre(\pi_2) \vee pre(\pi_2) \sqsupseteq pre(\pi_2) \wedge \\
& \quad\quad stuck(\pi_2, pre(\pi_1) \vee pre(\pi_2), \chi)\big) \\
& \text{from defn. of } stuck
\end{aligned}$$

The first line of the above predicate is trivially false, because $pre(\pi_1) \vee pre(\pi_2)$ enables one of the preconditions of π_1 and π_2 respectively. In addition, if we are proving that π_1 is not stuck in $pre(\pi_1) \vee pre(\pi_2)$ in which $pre(\pi_1) \vee pre(\pi_2) \sqsupseteq pre(\pi_1)$, then we can prove this for the more general case of $pre(\pi_1)$, and similarly for π_2. So, we are left with the following:

$$\begin{aligned}
\Leftarrow\ & \neg\big(stuck(\pi_1, pre(\pi_1), \chi) \vee stuck(\pi_2, pre(\pi_2), \chi)\big) \\
\equiv\ & \neg stuck(\pi_1, pre(\pi_1), \chi) \wedge \neg stuck(\pi_2, pre(\pi_2), \chi)
\end{aligned}$$

So, we are left to prove that protocols π_1 and π_2 are not stuck from their weakest precondition. Assuming an inductive proof over the structure of $\pi_1 \cup \pi_2$, π_1 and π_2 would have already been proven to not become stuck from their weakest precondition. This implies that the composing two stuckness-free protocols with the choice operator will produce a new protocol that is stuckness free, and therefore, no proof obligation is necessary.

4.1.5 Variable Declaration

For the initialisation proof of variable declarations, we substitute in a variable declaration for π from the

generic proof obligation and simplify:

$$\begin{aligned} & pre(\mathbf{var}_x^{\psi} \cdot \pi) \not\sqsupseteq \text{false} \\ \equiv\ & \exists_x pre(\pi, \exists_x \chi \sqcup \psi) \not\sqsupseteq \text{false} \quad \text{from defn. of } pre \\ \equiv\ & pre(\pi, \exists_x \chi \sqcup \psi) \not\sqsupseteq \text{false} \quad \text{from defn. of } \exists \end{aligned}$$

Assuming an inductive proof over the structure of $\mathbf{var}_x^{\psi} \cdot \pi$, this will have already been proved for π, therefore, no additional proof obligation is necessary. However, if we assume an agent is composing an existing protocol π that has been verified, but not under the local constraints $\exists_x \chi \sqcup \psi$, then this additional proof obligation must be discharged. This can be done by either proving the above, or by proving that $\exists_x \chi \sqcup \psi$ entails the local constraints under which it was previously proved. The latter case is possible because, if this is proved for a specific case, then it will also hold for a more general case.

To prove the absence of stuckness in a variable declaration, we substitute in a variable declaration for π from the proof obligation and simplify:

$$\begin{aligned} & \neg stuck(\mathbf{var}_x^{\psi} \cdot \pi,\ pre(\mathbf{var}_x^{\psi} \cdot \pi,\ \chi),\ \chi) \\ \equiv\ & \neg stuck(\mathbf{var}_x^{\psi} \cdot \pi,\ \exists_x pre(\pi, \exists_x \chi \sqcup \psi),\ \chi) \\ & \text{from defn. of } pre \\ \equiv\ & \neg stuck(\pi,\ \exists_x pre(\pi, \exists_x \chi \sqcup \psi), \exists_x \chi \sqcup \psi) \\ & \text{from defn. of } stuck \end{aligned}$$

Assuming an inductive proof over the structure of $\mathbf{var}_x^{\psi} \cdot \pi$, the following would already have been proved:

$$\neg stuck(\pi,\ pre(\pi, \exists_x \chi \sqcup \psi), \exists_x \chi \sqcup \psi)$$

This looks close to the predicate we wish to prove, except that the weakest precondition of π is $pre(\pi, \exists_x \chi \sqcup \psi)$, rather than $\exists_x pre(\pi, \exists_x \chi \sqcup \psi)$, which is even weaker because x is hidden, and therefore unconstrained. However, these predicates are in fact the same, because the constraints on x specified by ψ in $\exists_x pre(\pi, \exists_x \chi \sqcup \psi)$ may be hidden, but they still influence the behaviour of π via the ψ in the local constraints. Therefore, if any value of x from ψ is causing π to become stuck, the same value will also cause this to become stuck from ψ. This means that a stuckness proof obligation is not necessary for variable declarations. This is not difficult to see considering that π is already proved to be never become stuck when its precondition is enabled, and the precondition of $\mathbf{var}_x^{\psi} \cdot \pi$ is exactly those states that, when taking into consideration the additional variable x and its constraints, satisfy the precondition of π.

However, as with the initialisation proof, if we assume that π has been proved in the context of a different set of local constraints, then the above proof obligation must be discharged for the local constraints $\exists_x \chi \sqcup \psi$.

4.1.6 Reference Name

For the initialisation proof of reference names, we substitute in a reference name for π from the generic proof obligation and simplify:

$$\begin{aligned} & pre(N(x), \chi) \not\sqsupseteq \text{false} \\ \equiv\ & pre(\pi[x/y], \chi) \not\sqsupseteq \text{false} \quad \text{where } N(y) \mathrel{\widehat{=}} \pi \in D \\ \equiv\ & pre(\pi, \chi) \not\sqsupseteq \text{false} \\ & \text{because } \phi \in \mathcal{L} \text{ implies } \phi[y/x] \in \mathcal{L} \end{aligned}$$

Therefore, to discharge the initialisation proof obligation for a reference name, $N(x)$, in which $N(y) \mathrel{\widehat{=}} \pi$ is in the protocol specification D, one must prove that there exists a pre-state that satisfies the precondition of π. Assuming an inductive proof over the structure of the protocol, π will have already been proven to have an initial state, therefore, this proof obligation is not necessary in the case that $N(y) \mathrel{\widehat{=}} \pi$.

However, we have a different case, in which $N(y)$ is not a name in the protocol specification:

$$\begin{aligned} & pre(N(x)) \not\sqsupseteq \text{false} \\ \equiv\ & \text{nothing} \quad \text{where } N(y) \mathrel{\widehat{=}} \pi \notin D \end{aligned}$$

Therefore, to discharge the initialisation proof obligation for a reference name, $N(x)$, one has to prove that $N(y) \mathrel{\widehat{=}} \pi$ is a named protocol in the protocol specification. That is, one must prove:

$$N(y) \mathrel{\widehat{=}} \pi \in D \quad \text{for some } y$$

To prove the absence of stuckness in a reference name protocol, we substitute in a reference name for π from the proof obligation and simplify:

$$\begin{aligned} & \neg stuck(N(x), pre(N(x), \chi), \chi) \\ \equiv\ & N(y) \mathrel{\widehat{=}} \pi \in D \wedge \neg stuck(\pi[x/y], pre(\pi[x/y]), \chi)) \end{aligned}$$

Clearly, the right side of the conjunction is equivalent to the proof obligation of π, so assuming an inductive proof over the structure of the protocol, this part of the proof obligation is not necessary. The left side of the conjunction has already been proved for the initialisation proof of this protocol, therefore, no additional proof obligation is necessary.

```
prove(π, χ) : {true, false}
  if π = ψ → ε then
    return ψ ⊔ χ ⋢ false
  else if π = ψ --c.φ_m--> ψ' then
    return ψ ⊔ χ ⋢ false and ψ ⊔ φ_m ⊔ χ ⋢ false
  else if π = π1; (π2 ⊔ π3) then
    return prove(π1; π2, χ) and prove(π1; π3, χ)
  else if π = π1; π2 then
    return prove(π1, χ) and prove(π2, χ) and
      post(π1, pre(π1), χ) ⊒ pre(π2, χ)
  else if π = π1 ⊔ π2 then
    return prove(π1, χ) and prove(π2, χ)
  else if π = var_x^ψ · π1 then
    return prove(π1, ∃_x χ ⊔ ψ)
  else if π = N(x) then
    return N(y) ≙ π ∈ D and prove(π[x/y], χ)
```

Figure 1: Algorithm for Proving Protocols are Legal

4.2 Proving the Absence of Stuckness

When composing a protocol, we want to prove that our compositions are legal. To do this, we can discharge the proof obligations outlined in this section. However, many of these proof obligations refer to the local constraints, so one must relate these to the composition we are verifying. In this section, we outline an algorithm for verifying an arbitrary protocol.

The algorithm, which we call *prove*, takes a protocol and local constraints, and returns true or false, indicating whether the composition is legal or not. The sketch is shown in Figure 1, and should be clear to understand from the discussions in this section.

To prove a protocol π, is legal, one simply uses $prove(\pi, \text{true})$, in which true is the local constraint; that is, there are no local variables, so there are no local constraints.

The reader may have already noted a problem with the algorithm in Figure 1 regarding termination. If there are mutually recursive references to names in a protocol, this algorithm will not terminate. For example, take the following protocol specification:

$$A \;\widehat{=}\; B \cup \epsilon$$
$$B \;\widehat{=}\; A \cup \epsilon$$

This protocol can run infinitely, as well as terminate (if an agent chooses ϵ). The *prove* algorithm, as it is above, will attempt to prove the legality of A by looking up its respective definition, $B \cup \epsilon$ and verifying that. B will in turn be verified by looking up its definition, $A \cup \epsilon$, and verifying that, and the problem starts again with A, therefore running infinitely. A minor adjustment to the above algorithm that keeps track of the named protocols and parameters that have been verified will resolve this problem.

4.3 Proving Compositions

If we are to compose two existing protocols, then the proof system is somewhat different to that outlined in the *prove* algorithm. This is because we assume that, if an agent is to construct a new protocol from existing protocols that reside in some form of library, then the protocols in the library are already legal. This means we only have to discharge the proof obligations outlined in Section 4.1. Therefore, a composition algorithm only has to prove the following:

- For every empty protocol:
  - prove the existence of an initial state.
- For every atomic protocol:
  - prove the existence of an initial state; and
  - prove the existence of a post-state.
- For every sequential composition:
  - prove that every post-state of the protocol on the left enables a precondition on the right.
- For every variable declaration:
  - prove that adding the constraints on the new local variable do not lead the sub-protocol becoming stuck.
- For every referenced name:
  - prove that the name is in the protocol specification.

Proving the legality of compositions can be done with less effort than proving the legality of entire protocols. In addition, if the protocols are taken from the library, then they will have been proved under the assumption that there are no local constraints. A composition is not under any local constraints either, so we know that the sub-protocols used in the composition are proved under the same local constraints. The exception to this is variable declarations. If we compose protocol π with a declaration of a variable, such as $\mathbf{var}_x^{\psi} \cdot \pi$, then we must prove, using the *prove* algorithm from Figure 1 that π is valid under the local constraints ψ. The reason for this is easily demonstrated with an example. The atomic protocol, $x < 10 \xrightarrow{c.\phi_m} x = y + 1$, which is a legal protocol, can be composed with the variable declaration $\mathbf{var}_{x,y}^{x=y} \cdot x < 10 \xrightarrow{c.\phi_m} x = y + 1$. This new composition is not legal, because the local constraint $x = y$ must hold over the scope of x and y, but this makes the postcondition unsatisfiable: there does not exist values for x and y such that $x = y$ and $x = y + 1$. Therefore, in the case of creating a variable declaration composed from a legal protocol, one must prove that this protocol is satisfied under the specified local constraints.

```
compose(π) : {true, false}
  if π = ψ → ϵ then
    return ψ ⋤ false
  else if π = ψ --c.φ_m--> ψ' then
    return ψ ⋤ false and ψ ⊔ φ_m ⋤ false
  else if π = π1; (π2 ∪ π3) then
    return compose(π1; π2, χ) and
           compose(π1; π3, χ)
  else if π = π1; π2 then
    return post(π1, pre(π1), true) ⊒ pre(π2, true)
  else if π = var_x^ψ · π1 then
    return prove(π1, ψ)
  else if π = N(x) then
    return N(y) ≙ π ∈ D
  else return true
```

Figure 2: Algorithm for Proving Compositions are Legal

The procedure for verifying composite protocols that have been derived from other legal protocols is specified in the algorithm called *compose*, shown in Figure 2.

Note that *compose* does not take any local constraints as a parameter. As well as not recursively proving the legality of the sub-protocols, *compose* is also different to *prove* in that atomic protocols are verified without the local constraints χ, because the local constraints are assume to be just 'true'. The variable declaration case uses the *prove* algorithm from Figure 1, for the reasons discussed above. Finally, any instances of choice protocols need not be verified, because they are legal by default, therefore, these default to the final 'else', which returns 'true'.

5 Related Work

There are a handful of languages that have been used for first-class protocol specification. Various authors have had success with approaches based on Petri Nets [3] and on declarative specification languages [4, 5, 15], as well as an algebraic language similar to $\mathcal{RASA}$ called the Lightweight Coordination Calculus [12]. [8] presents a detailed comparison of these languages, including $\mathcal{RASA}$, so we do not cover this here. The authors of the cited work have discussed and demonstrated proof methods for these languages, but these involved proving domain-specific properties, not generic properties such as stuckness.

As noted in Section 3, verification of model-oriented specifications often prove generic properties that resemble our work. For example, the Cogito [1] and B method [13] development architectures both recommend an initialisation proof, which is similar our notion of an initialisation proof, and that a precondition of an operation should satisfy its postcondition, which is similar to the second proof obligation for atomic protocols in $\mathcal{RASA}$. However, these methods differ significantly because operations in these languages as modelled as relations between precondition and postconditions using predicates, whereas $\mathcal{RASA}$ specifies no relationship between precondition and postconditions, just some information that holds after a protocol executes.

Our notion of stuckness is similar to that of Pierce's definition of stuckness for programs [11], and in fact, this is from where we take the term. However, Pierce's motivation for identifying stuckness is to define programming languages such that it is not possible to write a program that becomes stuck, whereas we are aiming to prove the absence of stuckness, while defining the $\mathcal{RASA}$ language such that stuckness is

possible. Pierce uses *runtime exceptions* to handle programs that would otherwise become stuck. Exceptions could be added to $\mathcal{RASA}$, but we feel that raising runtime exceptions when a protocol cannot continue executing is no more desirable than becoming stuck, so we prefer to eliminate stuckness for protocols before their use.

6 Conclusions and Future Work

By treating agent interaction protocols as first-class entities, $\mathcal{RASA}$ permits protocols to be dynamically inspected, referenced, invoked, composed, and shared by ever-changing collections of agents engaged in interaction. The task of protocol composition, selection, and invocation may thus be undertaken by agents rather than agent designers, acting at run-time rather than at design-time. Frameworks such as this will be necessary to achieve the full vision of collections of intelligent autonomous agents interacting in dynamic environments.

This paper takes us one step towards such visions, by identifying generic cases in which first-class protocols are deemed to be illegal. We define a notion of *stuckness* for agent interaction protocols, and formally defined stuckness for the $\mathcal{RASA}$ framework. We have presented a method for proving that a protocol composition is legal, and noted the specific proof obligations that must be discharged when composing legal protocols into larger protocols. Proof obligations are specific to the composition operator that is being used in the composition, and can be discharged by the agents at runtime. Emphasis is placed on protocols specified in the $\mathcal{RASA}$ protocol language, but such ideas would be applicable to protocols specified in any language with features similar to $\mathcal{RASA}$'s.

Before our full visions are realised, significant further work is required. In other work [9], we are investigating methods for documenting the outcomes of protocols, which will allow agents to search for the protocols that best achieve their goals. 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 $\mathcal{RASA}$ framework.

Acknowledgements

We are grateful for financial support from the EC-funded PIPS project (EC-FP6-IST-507019), website: `http://www.pips.eu.org`, and the EC-funded ASPIC project (IST-FPC-002307), website: `http://www.argumentation.org/`, and the EPSRC Market-Based Control project (GR/T10664/01), website: `http://www.marketbasedcontrol.com/`.

References

[1] A. Bloesch and O. Traynor. The Cogito tool architecture. Technical Report 95-7, Software Verification Research Centre, 1995.

[2] F. S. De Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, September 1997.

[3] L. P. de Silva, M. Winikoff, and W. Liu. Extending agents by transmitting protocols in open systems. In *Proceedings of the Challenges in Open Agent Systems Workshop*, Melbourne, Australia, 2003.

[4] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. OWL-P: A methodology for business process modeling and enactment. In *Workshop on Agent Oriented Information Systems*, pages 50–57, July 2005.

[5] N. Desai and M. P. Singh. A modular action description language for protocol composition. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI)*, pages 962–967. AAAI Press, 2007.

[6] P. E. Dunne. Prevarication in dispute protocols. In G. Sartor, editor, *Proceedings of the Ninth International Conference on AI and Law*, pages 12–21, New York, NY, USA, 2003. ACM Press.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[8] J. McGinnis and T. Miller. Amongst first-class protocols. In *Engineering Societies in the Agents World VIII*, LNAI, 2007. (To Appear).

[9] T. Miller and P. McBurney. Executable logic for reasoning and annotation of first-class interaction protocols. Technical Report ULCS-07-015, University of Liverpool, Department of Computer Science, 2007.

[10] 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.

[11] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[12] D. Robertson. Multi-agent coordination as distributed logic programming. In *Proceedings of the International Conference on Logic Programming*, volume 3132 of *LNCS*, pages 416–430. Springer, 2004.

[13] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.

[14] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[15] 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.