Auton Agent Multi-Agent Syst (2018) 32:59–133
https://doi.org/10.1007/s10458-017-9373-9

A new Hierarchical Agent Protocol Notation

Michael Winikoff[1] · **Nitin Yadav**[2,3] · **Lin Padgham**[2]

Published online: 10 July 2017

Abstract Agent interaction descriptions (or protocols) are a key aspect of the design of multi-agent systems. However, in the authors' extensive experience, the notations commonly used for specification are both difficult to use, and lack expressiveness in certain areas. Some desired modular representations are impossible to express, while others result in specifications that are unwieldy and difficult to follow. In this paper we present a new notation for expressing interaction protocols, focussing on key issues that we have found to be problematic: the ability to define flexible data-driven protocols; representation of roles including their mapping to agents; and hierarchical modularity. We provide the semantics for our notation and illustrate its use with three diverse case studies. Finally we evaluate this notation using objectively assessable criteria that we argue contribute substantially to pragmatic usability, and using a human subject evaluation of the notation's usability.

Keywords Interaction protocols · Design notations · Agent interaction · Agent oriented software engineering

1 Introduction

When designing multi-agent systems one important aspect of the software engineering design process is designing interactions between the agents. The outcome of the interaction design

✉ Michael Winikoff
michael.winikoff@otago.ac.nz

Nitin Yadav
nitin.yadav@rmit.edu.au

Lin Padgham
lin.padgham@rmit.edu.au

[1] University of Otago, Dunedin, New Zealand

[2] RMIT University, Melbourne, Australia

[3] Present Address: University of Melbourne, Melbourne, Australia

activity is captured as *interaction protocols* expressed in an *agent protocol language*. These protocols are important not just for design, but also for a range of analyses, such as checking that the system will never deadlock.

In our 20 years of experience in designing, developing and teaching about agent systems, we have found protocol design to be one of the more problematic aspects. A key issue concerns the notations used. Existing notations, of which Agent UML (AUML) is perhaps most popular, are, in our experience, difficult to use correctly, and are unable to capture certain key aspects of interactions, including particular interaction styles or patterns that are used by agent systems. In other words, commonly used notations, such as AUML, are hard to use and lack expressivity.

One area where expressivity is lacking is in providing adequate support for modularity. For instance, although AUML provides a `ref` construct, there are situations where it is impossible to specify modular protocols. For example if we have a manufacturing cell, where various robot arms and other implements need to lock a table in order to perform some task, we would like to lift out the locking protocol, and use it in many places. However, it is not possible in AUML to express a locking protocol consisting of the sequence of messages *request-Lock*, *receive-Lock*, *release-Lock* and allow a variety of steps (depending on situation and who took the lock), between receive and release. As a result of this kind of limitation protocols developed can be quite unwieldy.

Another area where we have encountered difficulties is in the clean specification of very flexible protocols for collecting some specific set of data, but where the data does not need to be collected in a particular order. This is typical of protocols for interacting with a human, and is indeed where we have encountered the issue multiple times. For example, a protocol for collecting from a person the information required to schedule a meeting may need to allow for information to be provided in (almost) any order.

These issues (and others) relate to the *expressiveness* of the notation: we want a notation that is sufficiently expressive to deal with a range of interaction types, and our experience is that existing notations are not adequate. However, in addition to expressiveness, we also want a notation that is both *pragmatic* and *precise*. Precision means that the notation's syntax and semantics are clearly and unambiguously defined. This is important for the comprehensibility of the notation, and in particular in avoiding ambiguity in the meaning of a given protocol. Furthermore, precision is essential for providing advanced tool support. One important form of advanced tool support is checking whether certain properties hold in a protocol. For example, does an auction protocol cater for the situation where two bids for an identical amount are submitted in the same round?

We also want to provide a *pragmatic* notation. This means that the notation must be usable by practicing software designers, which typically implies a graphical notation, excludes the use of logics, and, more specifically, requires that various common cases that are already handled well by existing notations—such as sequence, selection and iteration—need to remain easy to specify. It is also desirable that a notation be as simple as possible.

What makes developing a good notation hard is that there is a tension between these objectives, so there is a tradeoff involved. For instance, a more precise notation is likely to require more details to be specified in a protocol, which can reduce the notation's usability. A simpler notation is more easy to specify precisely, but is more likely to lack expressiveness. In addition to a good notation having a carefully designed balance between expressiveness and simplicity, it must also be pragmatically usable and "fit for purpose". A notation that nicely balances simplicity and expressiveness but that is not usable in practice is not a good notation. Similarly, a notation that is not fit for purpose, for instance, cannot express important types of interaction, or important aspects of the interaction design, is not a good notation either.

This paper presents a new notational framework (HAPN: Hierarchical Agent Protocol Notation) that we have developed to address these issues. HAPN is based on hierarchical Finite State Machines (FSMs) as the underlying conceptual framework, which provides the basis for the hierarchical modularity we desire, while using a familiar graphical notation. We chose to use FSMs as the starting point for a number of reasons. Firstly, the notation is likely to be familiar to software engineers. Secondly, it is a simple notation (few constructs, and easy to explain) that is graphical in nature, and hence likely to be easier to use. This relates to the need to have a pragmatic notation that is usable by software engineers. Finally, FSMs are also precisely defined, which relates to the need to be able to provide tool support.

We are very aware that many notations have been proposed in the past. However, perhaps due to premature convergence in the field, we do not believe that there exists a notation that meets the needs outlined in this section (we substantiate this claim in Sect. 6.1 where we compare our proposed notation with a number of existing protocol notations).

A key assumption that underpins HAPN, and other conventional protocol notations, such as AUML and Statecharts, is that the design is done from a global perspective: a protocol is viewed and depicted as a process from the perspective of someone who can see the whole system. Additionally, it is also convenient to assume that message delivery is *synchronous*. This simplifies the design task by assuming that a message is received immediately after it is sent, with no possibility for race conditions.

However, when a protocol is implemented, it needs to be mapped to a collection of programs that operate with a local perspective: each entity participating in the interaction can only be aware of steps that it has participated in (e.g. messages that it has sent or received). While the global perspective provides a convenient fiction that makes the designer's work easier, it does introduce the possibility that the designer may specify a protocol that cannot be implemented [16,36]. For example, the (trivial) protocol that specifies a sequential interaction where step 1 is entity A sending a message to B, and then step 2 is C sending a message to A. This cannot be implemented because C has no way of knowing when it should send its message. The standard approach for dealing with this issue is to check that a protocol is realisable, as part of the process of designing the protocol. Similarly, assuming synchronous messages can also lead to issues in protocols, and the standard approach is to check for these as part of the design process. We emphasise that these assumptions, and solutions, are not specific to HAPN, but are a standard, and long-standing, approach that has been adopted by a wide range of protocol design notations.

Therefore, when an interaction protocol is verified (whether it is expressed as a Statechart, an AUML sequence diagram, or in HAPN), there are a number of properties to be checked. These include domain-specific properties relating to the protocol at hand, such as a confirmation message always being sent after payment, as well as generic properties including: (1) properties that check for general issues that can occur in any software (e.g. safety); (2) properties that reflect issues in distributed systems, such as the possibility for deadlock or livelock; and (3) checking for the protocol's realisability. Techniques for the verification of safety, liveness, and lack of deadlock are very well-studied in the literature. Since HAPN protocols can be "flattened" into Finite State Machines with variables, and verification techniques use structures similar to FSMs, the existing techniques are applicable to verifying these properties in the context of HAPN protocols. Additionally, HAPN's support for modular protocols means that modular verification is aided, by starting with a protocol that is already specified in a modular form. As discussed above, the need to check for realisability arises because of the convenient fiction of a central process that traditional interaction design notations offer the designer. Realisability has received less attention in the literature than verification, but there is still a body of work that has tackled this issue (e.g. [7,9,16,25,27]).

Broadly speaking, this body of work considers what conditions are required in order for a distributed enactment of a global-view protocol to be problem-free, and defines algorithms to check for these conditions. Finally, the properties discussed so far are all ones that are checked at *design time*. There are also *run time* checks that can be done in order to assess whether an implemented system is following the specified protocol.

While verification is an important area for future work, the focus of this paper is specifying the HAPN notation, and providing evidence that it represents a carefully engineered solution that manages to be precise without compromising on usability, and that manages to achieve high expressiveness while remaining simple.

In the following we briefly review related work (Sect. 2), leaving its evaluation to later in the paper. We then first describe HAPN's conceptual framework (Sect. 3), and then provide its formal semantics (Sect. 4). In Sect. 5 we present a number of case studies that demonstrate how HAPN is able to effectively model protocols containing various issues that we have found to be problematic. In Sect. 6.1 we identify a number of objective characteristics that contribute significantly to pragmatic usability, and evaluate our approach along with several other notations, with respect to these features. We also (Sect. 6.2) report on an experimental evaluation with human subjects. We then briefly discuss tool support (Sect. 7) before concluding (Sect. 8).

2 Related work

We now discuss in more detail related work. In particular, we focus on three broad research strands. The first two, commitment-based interaction specification and BSPL, aim to develop notations for representing protocols that are not traditional, and that reconsider some of the assumptions that underpin conventional protocol notations. The third area considers work in the domain of business modelling, which relates in particular to information-driven interactions. Some of this work is motivated by the need to represent flexible interactions, such as those driven by information (e.g. [12]).

Commitment Machines (CMs) [45,49,50] were proposed by Yolum and Singh as a representation for interactions that focussed on the social commitments that are created and discharged in the course of an interaction. The key idea in Commitment Machines is that by focusing on commitments, rather than on messages and their sequences, interactions can be modelled in a way that allows for flexibility. A Commitment Machine defines an interaction in terms of roles, fluents (propositions whose value changes over time), possible commitments, and (communicative) actions. Each communicative action is defined in terms of its pre-conditions, and its effects on the state (i.e. the fluents and commitments). For example, an offer from a merchant might create a commitment, or a payment might make the "paid" fluent true. Commitments can be conditional (e.g. a merchant commits to sending goods, conditional on the customer paying) and these become non-conditional commitments when the condition is met (e.g. becoming a commitment to send the goods, once the customer has paid). An interaction cannot conclude while there are active (non-conditional) commitments.

The basic framework of commitments has been extended by various authors in a number of ways. Montali et al. [29] argue that commitments over propositional languages are not expressive enough. They therefore extend the language of commitments to be over description logic, and they show that the resulting extension can still be verified under certain conditions. Chopra and Singh [12] define a rich notation (called "Cupid") that allows events to be combined in various ways, and that allows time and deadlines to be specified. They argue

that in order for a commitment such as C(paid, delivered) to make sense, it really needs to specify the information links between paid and delivered: what is being paid for, and what is being committed to be delivered once payment is received? The Cupid notation has subsequently been extended with the addition of other normative statements (e.g. prohibition, authorisation) and used to model norms [13].

In addition to extending the framework, there is also work that aims to provide software designers with guidance on how to develop commitment-based protocols. The Amoeba methodology [15] guides an analyst in capturing business processes using commitments. The methodology comprises five steps: (1) identifying the roles involved in the process, (2) using commitments to capture the contractual relationships, (3) defining the meanings of messages in terms of commitments, (4) specifying ordering constraints, and (5) forming a complete business process by combining individual protocols. A key benefit of using commitments to model business processes is that the resulting specification can be easily evolved in various ways as requirements change [15]. More recently, Baldoni et al. [6] proposed the 2CL methodology for engineering commitment-based business protocols. The methodology comprises five steps, which are adopted from Amoeba and extended. A distinctive feature is the addition of various constraints (e.g. temporal, relational) that can be specified, and a graphical notation that is adopted from the business process literature for depicting these constraints. The separation of the underlying interaction from the additional constraints allows a given protocol to be customised in various ways [5]. Baldoni et al. also developed a tool to support the development process, and carried out a user evaluation of the 2CL methodology. Compared with our evaluation (see Sect. 6.2), we note that their evaluation involved Ph.D. students and post-docs, not undergraduate students, and that their participants were merely asked to fill in a structured survey of their opinions of 2CL after attending a presentation. Their participants did not actually use 2CL. In another paper Baldoni et al. [4] considered a broader context: socio-technical systems. They proposed to use commitments to model social relations in socio-technical systems, and to implement this by embedding commitments into artefacts using CArtAgO, which is integrated with the JADE agent platform using middleware that they define.

There has also been work that has considered pragmatic engineering considerations relating to Commitment Machines, such as checking various properties of a Commitment Machine to provide the designer with feedback so they can improve their design [48], and a mapping to implement Commitment Machines in a BDI programming language [44].

We now turn to the Blindingly Simple Protocol Language (BSPL) [35]. BSPL is a more recent development that focuses on information-driven interactions and modularity. A key contribution of BSPL is that it goes back to fundamentals, considering, and articulating basic principles, and then proposing a completely new notation that is not an extension of traditional notations. Two of the key assumptions that underpin BSPL are a focus on *information-flow*, rather than on control-flow, and a strong commitment to embracing consistently the assumptions that systems are distributed (hence no shared state), and asynchronous.

BSPL therefore defines a (named) protocol as defining roles and parameters, and consisting of an (unordered) collection of elements, where each element is either a message or a sub-protocol. Constraints on which elements can occur, and in what order, are expressed via information flow, specifically by the requirement that each variable in an interaction can be bound only once. So, for example, if two messages (or sub-protocols) both bind the same variable, then they are (implicitly) mutually exclusive. Each message (or protocol) has a number of parameters, which are tagged as being either[1] "in" (need to be provided before

[1] A parameter can also be tagged as "nil".

the message/protocol occurs), or "out" (are provided by the message/protocol). BSPL also requires that each message is uniquely identified by the subset of its parameters that are its key. Protocols in BSPL can be composed.

BSPL protocols can be realised following an architectural framework ("LoST"—Local State Transfer). The LoST framework [36] makes a number of assumptions (e.g. no global state, messages are immutable). While these assumptions do constrain the interactions, they provide a number of desirable properties. For instance, messages can be safely repeated, and the delay of a message cannot cause problems.

BSPL is a text based simple notation (although some recent work on design guidance [38] does introduce some graphical notation). We evaluate both Commitment Machines and BSPL in Sect. 6.1. Briefly, we note for now that we consider both notations to have considerable longer-term promise (and, indeed, we have contributed to the body of work on Commitment Machines [18,42–45]). However, we do not believe that either notation is ready yet for pragmatic usage by practicing software engineers.

Finally, we consider related work in the domain of business modelling that has argued for the importance of information-driven interactions (Baldoni et al. [3] position this work with respect to the broader literature in multi-agent system). The basic arguments of this body of work from IBM [8,14,32] are that (1) in modelling a business, one must consider both data and process, not just focus primarily on one of the two; and (2) that a *business artefact* as a mechanism for representing data (along with its life-cycle) is an effective way of modelling business operations in a way that balances formality and end-user comprehensibility. A business artefact is defined as a uniquely identified concrete document that is structured, collects relevant information in one place, and is self-describing. The term "business record" is also used.

Considered in relation to our work, we firstly observe that the context and aim of the work is quite different: the IBM team are focussed on the problem of modelling business operations, whereas we are focussed on the challenge of modelling interactions between distributed entities (specifically agents). While there are some commonalities, for instance, information-driven interactions, there are also differences.

Considering information-driven interactions, business artefacts can provide a nice abstraction for information-driven interactions, where all the relevant information is grouped in one place. So, for instance, instead of having to track the day, place, and person for a play date (see Sect. 5.1) as separate variables, we could model a single compound data variable (or "business artefact") that included these three pieces of information. However, the logic that deals with which messages are appropriate at a given point in the interaction still needs to be captured as part of the protocol.

Turning to some of the other challenging interaction cases, and the required features, these do not appear to be addressed by the use of artefacts. For instance, modularity of interaction does not appear to be considered by the work on business artefact based modelling [8,14,32]. Turning to interactions with multiple role instances, where coordination is required, it is not clear how such interactions can even be modelled in their OpS framework. The OpS framework [32] defines an artefact's life-cycle in terms of a network of atomic tasks, where artefacts (or copies of their information) flow between tasks (they also use passive repositories that can store business artefacts). For an interaction such as an auction we need to be able to specify that each bidding agent bids, and receives a response, in parallel. The OpS notation does not appear to provide a way of doing this.

3 Conceptual framework

In this section we introduce the key ideas of the framework, linking the design choices that we made back to the requirements discussed in the previous section.

Before proceeding to define the basic building blocks of HAPN transitions (messages with guards and effects), we first introduce *variables*. Variables are used to store information that changes during the enactment of a protocol, such as the leading bid in an auction. Variables can also be used to store message contents in order to refer to them later in the protocol. For example, when a booking request is received, we may want to store the name of the movie in a variable, so that later in the protocol we can send a confirmation message that includes the name of the movie. There are also situations where it is important to have variables that are common to all protocol instances, which we refer to as *system* variables (non-system variables are referred to as *local* variables). System variables can be used to model system properties or resources, in order to give correct behaviour when more than one protocol instance is being enacted. For example, the number of available seats is a system variable: if two (or more) bookings are being made concurrently, then they need to access the shared `seats` variable.

The fundamental building block of any protocol notation is the *message*. In order to have a precise notation that allows reasoning over the state of the interaction, we need to specify not just the message name, but also the sender, recipient, and message contents. Therefore each message in the protocol is of the form: $\mathsf{Sender} \rightarrow \mathsf{Receiver}\colon \mathsf{msg}(args)$, where Sender and Receiver are respectively the *roles* that send and receive the message, msg is the name of the message, and $args$ is the message contents (optional). For example, $\mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{book}(RogueOne, Monday)$ is a message from a user (role U) to a booking system role (denoted S), requesting the system to book a ticket (message name "book") with message contents indicating the movie (*RogueOne*) and the day (*Monday*). We assume that messages are tagged with a conversation identifier that allows a message to be associated with its conversation. This is a common assumption (e.g. made by FIPA) and is required to avoid ambiguity when a protocol is instantiated more than once.[2]

There are various situations where a message should only be possible if a condition holds. For example, an auctioneer should only accept a bid if the new offer is higher than the previous highest bid. We therefore allow messages to have *guards* that constrain their occurrence. Following common notational practice (e.g. UML) we denote guards by placing them after the message in square brackets ("[" "]"). For example, if we wanted to constrain a booking request to only occur if there are free seats, then we might define the constrained message:

$$\mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{book}(RogueOne, Monday)[\texttt{seats} > 0]$$

This means that if there are no seats, then the User agent is not permitted to send a booking request, and hence that a valid implementation of the User agent must check whether there are free seats before it sends a book message. This is required so that the implementation does not send[3] a message which is not legal according to the protocol, which would make the implementation invalid.

[2] Or, more precisely, when a message appears in more than one currently active protocol instance, since two different protocols can use the same message name.

[3] The protocol is a design-time specification: it does not regulate run-time messaging, so does not prevent a message from being sent.

Finally, we extend messages to permit *effects*. These can be used to specify updates to variables. For example, the effect bind(`seats`, `seats` − 1) is used to update (decrement) the number of free seats when a booking is completed. In addition to allowing effects to update the value of a variable, we also allow them to indicate the point in the protocol where a domain-specific action is performed. For example, in the booking protocol we may want to indicate that a user's reward points are updated, or in a manufacturing scenario (see Sect. 5.3) we may want to indicate when actions such as loading or joining parts are performed. These are not messages, but in various scenarios they are essential to ensure that the protocol specifies the desired behaviour. We denote that a given role S performs a domain-specific action (e.g. **updatePoints**) using an effect of the form S: **updatePoints**().

Summarising so far, a *transition* in HAPN is a message with a guard and an effect, written msg[guard]/effect, where the message msg comprises a sender, receiver, message name, and (optional) message contents. Note that any of these three components (message, guard, or effect) can be omitted, but a transition must have at least one of them. In other words, a transition can consist of just a guard, or just an effect, or (obviously) just a message.

The HAPN notation is an extension of Finite State Machines (FSMs): the transitions are between states. Informally, the semantics of a transition msg[guards]/effects is that the transition can occur only if the guard is true, and the transition corresponds to the message occurring and the effects being applied.

One weakness of FSMs is that they lack support for specifying modular protocols (such as the locking example discussed in the introduction). FSMs also do not provide support for the specification of concurrency in protocols, or for a range of other interaction types (see Sect. 6.1). We therefore need to extend FSMs. In order to provide support for modular and reusable protocols, and for concurrency, we extend FSMs into *hierarchical* FSMs (following [1]). The basic idea, which is similar to statecharts, and hence likely to be familiar to designers, is that a state can contain one or more (concurrent) sub-protocols. However, while similar to statecharts (which are compared to our notation in Sect. 6.1), there is a significant difference in the semantics: in HAPN parallel machines must synchronise their transitions on shared messages (see the next Section for a precise definition). This requirement is important in allowing protocols to be reused and combined, which we illustrate later in this section, and in the case studies in Sect. 5.

In order to make protocols reusable we allow protocols to specify *interface* variables. These are used to allow the protocol to be instantiated in different contexts. For example, in a manufacturing scenario, a locking protocol might be instantiated in one context to lock a table in a given position for one robot, and in another context to lock a table in a different position for another robot. Interface variables are bound when the protocol is instantiated, and do not subsequently change.

Extending to hierarchical protocols also requires us to consider the *scope* of local variables. Each local variable is owned by a protocol, and has a specified scope: private (can only be accessed in that protocol), protected (can be accessed by the owning protocol and any of the protocols below it, i.e. its sub-protocols, sub-sub-protocols etc.), or public (can also be accessed by parent and sibling protocols). For example, in the booking protocol (Fig. 1) the local variable `req` is owned by the top-level protocol $\mathcal{P}_{\mathcal{B}}$, and can only be accessed by that protocol. On the other hand, the local variable `paid`, which is owned by the payment protocol $\mathcal{P}_{\mathcal{P}}$, has public scope and can be accessed by the other two protocols. Note that an alternative, and equally valid, design decision, would have been to have `paid` be owned by the top-level protocol and have it be either public or protected scope (these two scopes are equivalent for variables owned by the top-level protocol).

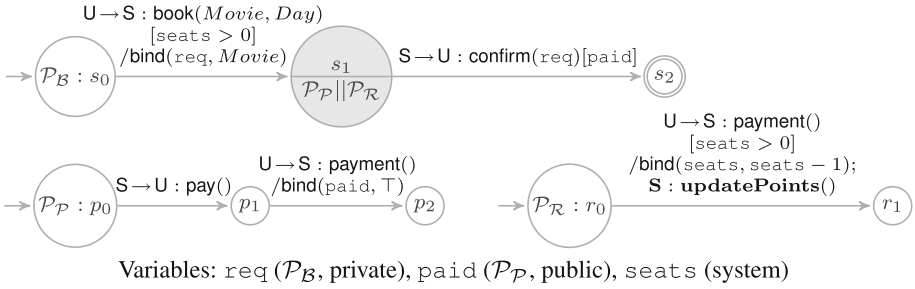


Fig. 1 A (schematic) booking example in our framework

A contrasting distinction is between *observable* and *unobservable* information. Information is *observable* if an external eavesdropper can discern it. For example, in a booking protocol (Fig. 1) the name of the movie is observable, since it is an argument of the initial book message, and hence can be observed by an external eavesdropper. On the other hand, the number of seats that is available is not communicated, and therefore it is unobservable, and cannot be known by an external observer. The distinction between observable and unobservable is important when we consider checking whether an execution conforms with a specified protocol. For example, if we want to check that an execution (i.e. an observed sequence of messages) conforms to the example booking protocol, then we cannot do so, since an external observer cannot evaluate the guard $\texttt{seats} > 0$. What we can do is verify compliance of the execution under the assumption that guards that rely on unobservable information do not need to be checked for compliance. It is worth highlighting that whether to include unobservable information in interaction protocols is a tradeoff for the designer to consider. On the one hand, it has been argued [34,36] to be undesirable, since it means that one cannot tell solely from observing messages whether the protocol is being followed correctly. On the other hand, there are cases where including such information is helpful to the designer to clarify the protocol's intentions. For example, representing the number of seats is required to be able to specify that a booking can be declined because no seats are available, and that successfully booking a seat reduces the number of available seats. This could be specified in terms of guards such as "seatsAvailable" and "updateAvailableSeats", but these leave out the important information that there is a number of available seats, and that the booking reduces this number by one (in this protocol).

Having (briefly) introduced the elements of HAPN, we now give a simple example to show the graphical notation, and to illustrate how HAPN is used. Figure 1 shows a simple protocol for booking a movie ticket ($\mathcal{P}_\mathcal{B}$, at the top of figure, where $\mathcal{B}$ is short for "Booking"). This booking protocol has two sub-protocols: $\mathcal{P}_\mathcal{P}$ and $\mathcal{P}_\mathcal{R}$ ($\mathcal{P}$ short for "Payment" and $\mathcal{R}$ short for "Reward"). The graphical notation used is an extension of the standard FSM notation. It extends the FSM notation by:

- Structuring each transition, so it must follow the form Sender → Receiver: msg($args$) [guard]/effect, where the message must specify a sender, receiver, message name msg and (optional) message content $args$.
- Showing for each (sub-)protocol its name and interface variables in the machine's initial state (in this example none of the protocols have interface variables—see the protocols later in this paper for examples of interface variables).

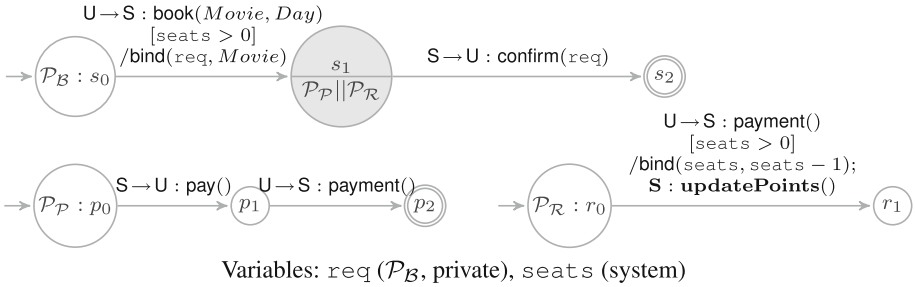


Fig. 2 Variant of booking example using a final state in a sub-protocol

– Allowing states to have sub-protocols, indicated by putting the sub-protocols in the bottom half of the state (below the line), and emphasised by shading the state (e.g. state s_1 in Figs. 1, 2).

In addition to the graphical rendition of the protocol, a complete specification needs to also provide: an indication of which is the top-level protocol (not shown in Fig. 1, but inferable to be the booking protocol, since the other two protocols are its sub-protocols); and an indication of the scope of each variable, and which protocol it belongs to. As indicated in the figure, there are three variables: `req` (which is only used in the top-level booking, and is a private variable), `paid` (owned by the payment protocol, but public, since it is also used in the top-level protocol), and `seats` which is a system variable.

We note that one benefit of using FSMs as a starting point, rather than some form of message sequence chart (e.g. AUML), is that the FSM notation avoids the additional constraint of having the layout follow agent lifelines, and, by allowing the layout to follow the process of the protocol, it therefore can make it easier to follow the overall flow of a protocol [for example compare Figs. 9 (HAPN) and 10 (AUML)]. On the other hand, AUML, with its agent lifelines, can make it easier to see what each role contributes to the protocol.

As mentioned earlier, the semantics of sub-protocols of a given state is that they occur concurrently. However, as noted earlier, a key feature of HAPN (distinguishing it from statecharts) is that such parallel protocol instances synchronise on shared messages (i.e. where a message instance matches the message part of a transition in multiple active sub-protocols). For example, state s_1 of the booking protocol has two sub-protocols, both of which have a transition with the payment message. This means that this message is only permissible when the payment protocol $\mathcal{P}_\mathcal{P}$ is in state p_1 and the protocol $\mathcal{P}_\mathcal{R}$ is in state r_0. In this situation, when the payment message occurs, both protocols transition (respectively to states p_2 and r_1), and the effects of both transitions are applied.

This ability to synchronise is crucial in allowing for protocols to be extended in a modular way. In this case we have a generic payment protocol, which is extended to also perform domain-specific updates (to the number of seats, and to the reward points). This extension is done without modifying the payment protocol itself, by adding $\mathcal{P}_\mathcal{R}$, using a shared message.

An example interaction that might occur following this booking protocol is as follows. The protocol begins in initial state s_0, and we assume that there are seats available. The user sends a booking request (book), at which point the protocol transitions to s_1, which also binds `req` to the name of the *Movie* requested (part of the contents of the book message). State s_1 has two sub-protocols. These are instantiated (in their starting states) when s_1 is entered. At this point all three protocols are "active". However, in fact, only $\mathcal{P}_\mathcal{P}$ is able to transition, since $\mathcal{P}_\mathcal{R}$ must synchronise on the payment message, and the top-level protocol cannot transition to s_2 until `paid` is true (we assume it is initially false). Therefore the next message is that

the system asks the user to pay (pay). This can be followed by the user paying (payment) which, as discussed above, results in both $\mathcal{P}_\mathcal{P}$ and $\mathcal{P}_\mathcal{R}$ transitioning, and in paid becoming true ($\top$), and the number of available seats and the user's reward points both being updated (in $\mathcal{P}_\mathcal{R}$). At this point the top-level booking protocol can confirm the booking (confirm), transitioning to s_2, and concluding the booking interaction.

Another feature of HAPN is that it allows sub-protocols to have final states. The semantics of this is that the parent state of the sub-protocol cannot be exited until the sub-protocol is in a final state.[4] This feature can sometimes be used to simplify protocols. For instance, if we modify the booking protocol by making the payment protocol $\mathcal{P}_\mathcal{P}$ have a final state p_2, then the top-level protocol cannot exit state s_1 until the payment process is concluded. This then means that the guard [paid] is redundant, and the protocol can be simplified by removing the guard, and eliminating the variable paid, yielding the protocol shown in Fig. 2.

It is worth noting that this protocol is simple, for illustration purposes. Specific aspects that would need to be elaborated for a full protocol include: (1) adding a transition for the book message to handle the case where there are no seats available, which could be done by sending an error message and allowing the user to request an alternative movie or day; and (2) dealing with the case where a seat is available at the start of the interaction, but by the time payment is made someone else has booked the seat (i.e. the guard on the transition from r_0 is false). This could be handled in a number of ways. One simple way would be to report to the user that there is no longer a seat available. Another option would be to allocate the seat to the user when they request it, but then require them to complete the transaction within a certain time period, otherwise the seat is released.

4 Formalisation

We now formalise the key notions presented in the previous section. Note that the formal definitions in this section, which provide a precise definition of the syntax and semantics of the notation, would not be used by protocol designers: they would use the graphical notation, which can be mapped directly to the formal structures defined in this section (see Sect. 4.9). The role of the formalisation, apart from ensuring that the notation has sensible and precise semantics, is to allow for tool support to assist designers by checking that their protocols have certain desirable properties.

We begin by defining variables, and then defining the three components of transitions: messages, guards, and effects. We then define simple protocols, along with their semantics, and then extend to hierarchical protocols. We then give the semantics of hierarchical protocols by defining a mapping that flattens a hierarchical protocol into a simple protocol.

We assume that each protocol shares an ontology that the sender uses to send only meaningful messages. That is, for each message definition $\mathsf{sender} \rightarrow \mathsf{recv}\colon \mathsf{msg}(V_1, \ldots, V_n)$, when msg is sent with arguments $v_1, \ldots, v_n$ it is the case that v_i is an instance of ontology concept V_i.

4.1 Variables

We distinguish between protocol interface variables V^I (which are part of the interface of a protocol, and bound when the protocol is instantiated), and variables (which can be

[4] More precisely, for any state with sub-protocols, any transitions from that state are considered to have an implicit guard that is true iff for all sub-protocols, either the sub-protocol has no final states, or it is in a final state.

changed by the effects of the protocol). The set of variables consists of *local* variables, that are specific to a protocol instance, and *system* variables that are common to all instances of a protocol. Formally, we define $V = V_L \cup V_G$ where $V_L = \{v_1, \ldots, v_n\}$ ($V_G = \{v_{n+1}, \ldots, v_m\}$) as the finite set of local (system) variables with respective finite domains $D_{v_1}, \ldots, D_{v_n}$ ($D_{v_{n+1}}, \ldots, D_{v_m}$). We assume that V_G includes a "time" variable that refers to the time elapsed since the start of the protocol.

As discussed in the previous section, each local variable has an owning protocol and a scope (except for system variables which have global scope). To avoid ambiguity we refer to a local variable v of protocol $\mathcal{P}$ by $\mathcal{P}.v$ where the distinction is significant and not otherwise clear.

The notion of variable scope that we use is standard, and we do not model it in the formalisation that follows, since this would result in additional unhelpful complexity.

4.2 Messages

A *message over* $V \cup V^I$ is a pattern of the form $\mathsf{from} \rightarrow \mathsf{to}\colon \mathsf{message}(\mathsf{args})$ which includes four components:

- from: the sender of the message (defined as a role),
- to: recipient(s) for the message (a role or set of roles),
- message: message label, and,
- args: zero or more arguments.

We also allow the use of "⇒" instead of "→" to highlight a message with multiple recipients. Any of these four components can be a constant (of the correct type), or a variable (local, system or protocol interface variable). For example, in the auction protocol (Fig. 9 in Sect. 5.2) the first announce message has three arguments: I and T are the protocol's interface variables, which are bound to the item being auctioned, and the time limit. The third argument, *price*, is a variable that is bound to the price that is announced.

Message variables appearing in the message arguments in a protocol are automatically considered to be part of the (local) variables for that protocol with private scope (e.g. *price*). Formally, a message σ in a protocol $\mathcal{P}$ can only contain variables that are drawn from the variables V or from that protocol's interface variables (V^I). We use σ_ϵ to denote an *empty* message.

4.3 Guards

A *guard over* $V \cup V^I$ is a Boolean formula built out of the sets of predicates $\mathsf{in}(N, q)$ (where N is a protocol name and q a state of that protocol), $\{\mathsf{bound}(v) \mid v \in V\}$ and $\{v\ R\ val \mid (val \in D_v \vee val \in V^I \cup V) \wedge v \in V\}$ (where R is a relational operator, $=$, $<$, $>$, $\leq$, $\geq$, etc.), and defined by the grammar:

$$guard ::= v\ R\ val \mid \mathsf{bound}(v) \mid guard_1 \wedge guard_2 \mid \neg guard_1 \mid \mathsf{in}(N, q) \mid \top$$

Note that in relational predicates the value can be a constant, a variable, or a protocol interface variable ($val \in V^I$). More generally, we allow conditions to include expressions built using Boolean connectives. The guard $\top$ denotes that no preconditions are required for sending a message. The guard $\mathsf{in}(N, q)$ returns $\top$ iff the sub-protocol $\mathcal{P}$ with name N is in state q.

4.4 Effects

An *effect over* $V \cup V^I$ is built out of the sets of operations $\{\mathsf{bind}(\mathrm{v}, val) \mid (val \in D_{\mathrm{v}} \vee val \in V^I \cup V) \wedge \mathrm{v} \in V\}$ and $\{\mathsf{unbind}(\mathrm{v}) \mid \mathrm{v} \in V\}$, as well as the state change $\mathsf{set}(N', q)$ where N' is a machine name and q is a state of machine N'. We also allow effects to include domain-specific actions. This yields the grammar:

$$\mathit{effect} ::= \mathsf{bind}(\mathrm{v}, val) \mid \mathsf{unbind}(\mathrm{v}) \mid \mathsf{set}(N, q) \mid \mathit{effect}_1; \mathit{effect}_2 \mid \epsilon \mid r\colon \mathsf{act}$$

where $r\colon \mathsf{act}$ denotes a domain action by an agent playing role r (where r can be a constant or a protocol interface variable). The symbol ϵ denotes the empty effect. We also allow the second argument of the bind operator to be an arbitrary expression over the variables and constants, for example $\mathsf{bind}(\mathrm{v}, \mathrm{v} - 1)$.

The composite effect $e_1; \ldots; e_m$ represents a *consistent* concatenation of (non-composite) effects e_1 to e_m. We require that for any transition, the effect of that transition $e_1; \ldots; e_m$ is consistent, and therefore can be executed as a single step, that is, various interleavings between e_i, where $1 \leq i \leq m$, will result in the same effect on variables. More precisely, given an effect $e_1; \ldots; e_n$ there cannot be two effects e_i and e_j where one binds a variable and another unbinds that variable, or where both bind the same variable.

4.5 Simple protocols

A *simple protocol* is a tuple $\mathcal{P} = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta \rangle$ where:

- N is a name for the protocol;
- V^I and V are a set of interface and variables ($V = V_L \cup V_G$);
- Q is a finite set of states;
- Σ is a finite set of messages over V;
- G is a finite set of guards over V;
- E is a finite set of effects over V;
- $q^0 \in Q$ is the initial state;
- $Q^F \subseteq Q$ is the set of final states; and
- $\delta\colon Q \times (\Sigma \cup \{\sigma_\epsilon\}) \times G \to Q \times E$, is the transition relation.

We define $\mathcal{P}.Q$ as the Q component of $\mathcal{P}$ and similarly for V, E, etc.

The *enacted state* of a simple protocol $\mathcal{P}$ consists of the pair $\langle q, b \rangle$ where $q \in \mathcal{P}.Q$ is one of the protocol's states, and b is a binding that gives the values of local and system variables that are bound, and interface variables. Initially the binding only contains values for interface variables, and possibly for system variables, i.e. all local variables are unbound. We use $b[v]$ to denote the value that binding b gives for the variable v, and we use $b[v] = \perp$ to indicate that variable v is not bound by b.

A transition $\delta(q, \sigma, g) = (q', e)$ indicates that when $\mathcal{P}$ is in state q and guard g holds, then the message σ may result in change of $\mathcal{P}$'s state to q' and application of effects e. For brevity, we will denote a transition $\delta(q, \sigma, g) = (q', e)$ as $q \xrightarrow{\sigma[g]/e} q'$. If the guard is empty than we elide the $[g]$ and if the effect is empty then we elide the $/e$, so the transition $q \xrightarrow{\sigma[\top]/\epsilon} q'$ would be written as just $q \xrightarrow{\sigma} q'$. Formally, the semantics is defined in terms of runs over enacted states (see Sect. 4.7).

Observe that we do not specifically list the roles involved in a protocol as these can be deciphered automatically from the messages involved in the protocol.

4.6 Hierarchical protocols

Before proceeding to define hierarchical protocols we need to extend the notion of guards and effects to hierarchical protocols. As noted earlier, a guard over a protocol $\mathcal{P} = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle$ can be a Boolean function of the form $\mathsf{in}(N, q)$, where $q \in Q$, that returns $\top$ iff the current state of $\mathcal{P}$ is q. Similarly, an effect over a protocol $\mathcal{P} = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle$ can be $\mathsf{set}(N, q)$ which makes $\mathcal{P}$'s current state q. We extend the notion of consistency of effects to include $\mathsf{set}(\cdot,\cdot)$ by requiring that in an effect $e = e_1; \ldots; e_n$:

1. for all effects $e_i = \mathsf{set}(N_i, q_i)$ there should exists a protocol $\mathcal{P}$ such that $\mathcal{P}.N = N_i$ and $q_i \in \mathcal{P}.Q$.
2. for any two constituent effects $e_i = \mathsf{set}(N_i, q_i)$, $e_j = \mathsf{set}(N_j, q_j)$ that if $N_i = N_j$ then $q_i = q_j$.

Next, we allow a state in a protocol to include a set of protocols. We call such a state a *superstate*.

A *hierarchical protocol* is a tuple $\mathcal{P}_H = \langle \mathcal{P}_0, \{\mathcal{P}_1, \ldots, P_n\}, \mu, \beta\rangle$ consisting of

1. a top-level protocol $\mathcal{P}_0 = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle$;
2. a set of (numbered) sub-protocols $\{\mathcal{P}_1, \ldots, P_n\}$, where each $\mathcal{P}_i$ is either a simple protocol, $\mathcal{P}_i = \langle N_i, V_i^I, Q_i, V_i, \Sigma_i, G_i, E_i, q_i^0, Q_i^F, \delta_i\rangle$, or a hierarchical protocol, $\mathcal{P}_i = \langle \mathcal{P}_0^i, \{\mathcal{P}_1^i, \ldots, P_m^i\}, \mu^i, \beta^i\rangle$;
3. a mapping function μ that specifies the structure of the hierarchical protocol by mapping a given state $q \in \mathcal{P}_0.Q$ to a set of its sub-protocols; and
4. an instantiation function β that specifies for each state and sub-protocol a binding for the sub-protocol's interface variables. For any state q such that $\mu(q)$ is non-empty, then for each protocol $\mathcal{P}_i \in \mu(q)$ where $\mathcal{P}_i.V^I \neq \emptyset$ we have $\beta(q, \mathcal{P}_i) = \{(v \mapsto d) \mid v \in \mathcal{P}_i.V^I\}$ where each expression d is the value that the protocol binds to the interface variable. We write $\beta(q, \mathcal{P}_i)(v)$ for $v \in \mathcal{P}_i.V^I$ to denote the value that the binding gives to v.

We need to impose a number of constraints. Firstly, we require that the mapping function μ is non-circular, i.e. that no state can be transitively mapped to itself. Secondly, we require that all of the V^I are disjoint (which is easily ensured by renaming), and that we have a single set of system variables (also easily ensured by enlarging each protocol's set of system variables to the complete set of all system variables). Thirdly, we assume that variable scope is managed (e.g. by renaming apart variables that are distinct due to scope but have the same name), and that variables are always initialised prior to use (which means that the semantics does not need to track when variables need to be destroyed, when their owning sub-protocol instance is deleted). Finally, we need to require that the effects on transitions are such that synchronised transitions always have consistent effects. So when two sub-protocols have transitions on the same message, they cannot have inconsistent effects.

4.7 Semantics of transitions

Recall that a transition $q \xrightarrow{\sigma[g]/e} q'$ indicates that when $\mathcal{P}$ is in state q and guard g holds, then the message σ may result in change of $\mathcal{P}$'s state to q' and application of effects e. Formally, the semantics is defined in terms of runs over enacted states. Given an enacted state $\langle q, b\rangle$ and received message σ', a transition $q \xrightarrow{\sigma[g]/e} q'$ is applicable if there exists a substitution of variables with values θ, where θ extends b, such that applying θ to σ makes it equal to σ' (formally $\sigma' = \sigma\theta$), and g holds given the bindings in b updated with θ. The requirement

that θ extends b captures that a received message σ' can only match a message pattern σ in a protocol if the existing bindings for variables in σ are matched. For example, if a protocol has a message $accept(V)$ where V is a variable that has been bound earlier, then a received message $accept(13)$ is only accepted by the protocol if V is bound to 13. The intuition for when g holds given bindings b is that a guard $\mathsf{bound}(\mathrm{v})$ holds iff v is bound in b, and $\mathrm{v}\ R\ val$ holds iff $b[\mathrm{v}]\ R\ val$ holds. A guard $\mathsf{in}(N,q)$ holds iff the sub-protocol named N is in state q.

Applying the transition results in the enacted state $\langle q', b'\rangle$ where b' is the result of updating b by first applying θ, and then applying the binding and unbinding effects of e. The semantics do not track actions (see definition of appeff below), as they do not affect future evolution of the protocols. This could be added for monitoring if desired.

In the case where the transition has an empty message σ_ϵ, the transition can be taken if the guard holds, and results in a state change, and the application of the effects associated with the transition.

We now define these notions formally. Given an enacted state $\langle q, b\rangle$ we write $\langle q, b\rangle \models g$ to indicate that the guard g holds with respect to the enacted state. This is defined formally as follows:

$$
\begin{aligned}
\langle q, b\rangle &\models \top && \text{iff } true\\
\langle q, b\rangle &\models g_1 \wedge g_2 && \text{iff } \langle q, b\rangle \models g_1 \text{ and } \langle q, b\rangle \models g_2\\
\langle q, b\rangle &\models \neg g && \text{iff } \langle q, b\rangle \models g \text{ does not hold}\\
\langle q, b\rangle &\models \mathsf{bound}(v) && \text{iff } b[v] \neq \bot\\
\langle q, b\rangle &\models \mathrm{v}\ R\ var && \text{iff } b[\mathrm{v}]\ R\ b[var] \text{ (for } var \in (V^I \cup V))\\
\langle q, b\rangle &\models \mathrm{v}\ R\ val && \text{iff } b[\mathrm{v}]\ R\ val \text{ (for } val \in D_{\mathrm{v}_i})\\
\langle q, b\rangle &\models \mathsf{in}(N', q') && \text{iff } \mathsf{in\text{-}f}(N', q', q)
\end{aligned}
$$

where the last clause uses the auxiliary predicate $\mathsf{in\text{-}f}(N', q', q)$. Note that the last argument, q, can be either an atomic state, or a flattened state (q, w) so there are two cases (a flattened state arises in the context of hierarchical protocols, we define these formally in the next section).

$$
\begin{aligned}
\mathsf{in\text{-}f}(N', q', q) &\quad \text{iff } machname(q) = N' \text{ and } q = q'\\
\mathsf{in\text{-}f}(N', q', (q, w)) &\quad \text{iff } \mathsf{in\text{-}f}(N', q', q) \text{ or } \exists q_i \colon \mathsf{in\text{-}f}(N', q', q_i)\\
&\qquad \text{where } w = (q_1, \ldots, q_n)
\end{aligned}
$$

where $machname(q)$ denotes the name of the machine which q is a state in. Formally: $machname(q) = name$ where $\mathcal{P}_i.N = name$ and $q \in \mathcal{P}_i.Q$. In other words, there exists a (sub-)protocol $\mathcal{P}_i$ with name $name$ that has state q as a state. This assumes that states are renamed apart so a given state q cannot appear in more than one (sub-)protocol.

Given an enacted state $\langle q, b\rangle$ we write $\mathsf{appeff}(e, \langle q, b\rangle)$ to denote the enacted state that results from applying the effects e. Note that, as mentioned earlier, a domain action $r\colon \mathsf{act}$ does not produce an effect on the state of variables or of the machine, which is why the first two cases have the same right hand side.

$$
\begin{aligned}
\mathsf{appeff}(\epsilon, \langle q, b\rangle) &= \langle q, b\rangle\\
\mathsf{appeff}(r\colon \mathsf{act}, \langle q, b\rangle) &= \langle q, b\rangle\\
\mathsf{appeff}(e_1; e_2, \langle q, b\rangle) &= \mathsf{appeff}(e_2, \mathsf{appeff}(e_1, \langle q, b\rangle))
\end{aligned}
$$

$$\begin{aligned}
\mathsf{appeff}(\mathsf{bind}(\mathrm{v}, val), \langle q, b\rangle) &= \langle q, b[\mathrm{v} := eval(val, b)]\rangle \\
&\quad \text{where } b[v := val] \text{ denotes the binding that differs} \\
&\quad \text{from } b \text{ in that } v \text{ is updated to be bound to } val \\
&\quad \text{and } eval(val, b) \text{ denotes the evaluation of the} \\
&\quad \text{expression } val \text{ with variables replaced with their} \\
&\quad \text{bindings in } b \\
\mathsf{appeff}(\mathsf{unbind}(\mathrm{v}), \langle q, b\rangle) &= \langle q, b[\mathrm{v} := \bot]\rangle \\
\mathsf{appeff}(\mathsf{set}(N', q'), \langle q, b\rangle) &= \langle \mathsf{set\text{-}f}(N', q', q), b\rangle
\end{aligned}$$

where the auxiliary function $\mathsf{set\text{-}f}(N', q', q)$ updates the state q (or (q, w) for flattened states) by changing the state of machine N' to q':

$$\begin{aligned}
\mathsf{set\text{-}f}(N', q', q) &= \begin{cases} q' & \text{if } machname(q) = N' \\ q & \text{otherwise} \end{cases} \\
\mathsf{set\text{-}f}(N', q', (q, w)) &= \begin{cases} \big(q', f^0(\mu(q'))\big) & \text{if } machname(q) = N' \\ \big(q, (q'_1, \ldots, q'_n)\big) & \text{otherwise} \end{cases} \\
&\quad \text{where } f^0 \text{ is defined in the next section and } w = (q_1, \ldots, q_n) \\
&\quad \text{and } q'_i = \mathsf{set\text{-}f}(N', q', q_i)
\end{aligned}$$

4.8 Semantics of superstates

When a protocol enters a superstate then all of that state's sub-protocols are initialized. Once a protocol exits a superstate then its sub-protocols are no longer active (or can be referred to).

Our semantics is based on the mechanism of expanding a hierarchical protocol into a "flat" Finite State Machine [1]. However, we need to extend the standard mapping used by Alur et al. [1] to deal with our richer setting. For instance, when merging a transition in one machine with a concurrent transition in another machine, the two transitions may have different guards or effects.

Given a protocol $\mathcal{P}$ we use $[\![\mathcal{P}]\!]$ to denote the corresponding flattened protocol. For convenience we define $[\![\mathcal{P}]\!] = \mathcal{P}$ when $\mathcal{P}$ is a simple protocol. The remainder of this section defines $[\![P]\!]$ when $\mathcal{P}$ is a hierarchical protocol. In the definition of $[\![\mathcal{P}]\!]$ we use μ and β as shorthands for $\mathcal{P}.\mu$ and $\mathcal{P}.\beta$ respectively.

Given a hierarchical protocol $\mathcal{P} = \langle \mathcal{P}_0, \{\mathcal{P}_1, \ldots, P_n\}, \mu, \beta\rangle$, its equivalent simple protocol is the tuple $[\![\mathcal{P}]\!] = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle$ where components V, Σ, G, E are the union of their respective elements in $\mathcal{P}_i$ (e.g. $[\![P]\!].G = \bigcup_{0 \le i \le n} \mathcal{P}_i.G$), the name of the protocol is the same as the name of the top level protocol ($[\![\mathcal{P}]\!].N = \mathcal{P}_0.N$), and the protocol's interface variable are those of the top level protocol ($[\![\mathcal{P}]\!].V^I = \mathcal{P}_0.V^I$). Note that for variables, the overall set of variables includes not just the variables V of the sub-protocols, but also the sub-protocols' interface variables V^I, i.e. $[\![P]\!].V = \mathcal{P}_0.V \cup \bigcup_{1 \le i \le n} \mathcal{P}_i.V \cup \mathcal{P}_i.V^I$.

States of $[\![\mathcal{P}]\!]$ consist of recursive structures of the form (q, w) where q is a state of the top level protocol ($q \in \mathcal{P}_0.Q$) that has sub-protocols ($\mu(q) \neq \emptyset$) and w denotes (flattened) states of protocols in $\mu(q)$, as well as simple states q where $\mu(q) = \emptyset$. The intuition is that a state in $[\![\mathcal{P}]\!]$ represents a hierarchical snapshot of $\mathcal{P}$. For example, in the flattened version of the booking protocol (Fig. 1) the state s_1, when it is entered, corresponds to the composite

flattened state $(s_1, (p_0, r_0))$. Formally:

$$[\![\mathcal{P}]\!].Q = \{q \,|\, q \in \mathcal{P}_0.Q \wedge \mu(q) = \emptyset\} \cup \{(q, w) \,|\, q \in \mathcal{P}_0.Q \wedge w \in f(\mu(q))\}$$

where $f(\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}) = \{(q_1, \ldots, q_n) \mid q_i \in [\![\mathcal{P}_i]\!].Q\}$.

If the initial state of the top-level machine does not have sub-protocols, then the initial state of the flattened machine is just the top-level machine's initial state. Otherwise, the initial state of the flattened machine is the composite state where the first component is the initial state of the top-level machine, and the second component is (recursively) the initial state of its sub-machines. Formally:

$$[\![\mathcal{P}]\!].q^0 = \begin{cases} \mathcal{P}_0.q^0 & \text{if } \mu(\mathcal{P}_0.q^0) = \emptyset \\ \left(\mathcal{P}_0.q^0,\ f^0(\mu(\mathcal{P}_0.q^0))\right) & \text{otherwise} \end{cases}$$
$$\text{where } f^0(\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}) = \left([\![\mathcal{P}_1]\!].q^0, \ldots, [\![\mathcal{P}_n]\!].q^0\right)$$

The final states of the flattened machine are all those (composite) states where the component corresponding to the state of the top-level machine is one of its final states, as well as any states in $\mathcal{P}_0.Q^F$ that do not have sub-protocols. Formally:

$$[\![\mathcal{P}]\!].Q^F = \left\{q \mid q \in \mathcal{P}_0.Q^F \wedge \mu(q) = \emptyset\right\} \cup \left\{(q, w) \mid q \in \mathcal{P}_0.Q^F \wedge (q, w) \in [\![\mathcal{P}]\!].Q\right\}$$

The transition function $\mathcal{P}.\delta$ has two categories of transitions: transitions that change a state of a top level protocol and transitions that are in the sub-protocols (the top level state remains the same).

In the first case, when a transition causes a protocol to change its state from q to q', the resulting state is such that all sub-protocols in q' get initialized to their initial states, and the sub-protocols' interface variables are bound, using β. Formally, for that case we have:

$$[\![\mathcal{P}]\!].\delta((q, w), \sigma, g) = \begin{cases} ((q',\ f^0(\mu(q'))), e^+) & \text{if } \mathcal{P}.\delta(q, \sigma, g) = (q', e) \ \wedge \ \mu(q') \neq \emptyset \\ (q', e) & \text{if } \mathcal{P}.\delta(q, \sigma, g) = (q', e) \ \wedge \ \mu(q') = \emptyset \end{cases}$$

where $(q, w) \in [\![\mathcal{P}]\!].Q$

$$\text{and } e^+ = e\ ;\ \left(\fatsemi \{\mathsf{bind}(v_i, \beta(q', \mathcal{P}_i)(v_i)) \mid \mathcal{P}_i \in \mu(q') \wedge v_i \in \mathcal{P}_i.V^I\}\right)$$
$$\text{and } \fatsemi \{e_1, \ldots, e_n\} = e_1; \ldots; e_n$$
$$\text{and } \bigwedge_{\mathcal{P}' \in \mu(q)} \left(\mathcal{P}'.Q^F = \emptyset\right) \vee \left(\bigvee_{q^f \in \mathcal{P}'.Q^F} \mathsf{in\text{-}f}(\mathcal{P}'.N, q^f, (q, w))\right)$$

The first line, first case, specifies that when we transition from a composite state (q, w) into another composite state (i.e. one where $\mu(q') \neq \emptyset$) then the resulting state is $(q', f^0(\mu(q')))$ (the auxiliary function f^0, defined earlier, maps a set of sub-protocols to a tuple of their starting states, ordered by the protocol numbers). The second case applies when the transition is to a state q' that does not have sub-protocols, in which case the resulting state is just q'. A transition from q to q' results in a flattened transition $\delta((q, w), \ldots)$ for each w where (q, w) is a valid state; this is done by the condition $(q, w) \in [\![\mathcal{P}]\!].Q$.

There are two additional conditions. The condition $e^+ = \ldots$ deals with binding the interface variables of sub-protocols. When a sub-protocol is initialised, we extend the effects of the transition e by adding in effects that bind the interface variables of sub-protocols. For each sub-protocol ($\mathcal{P}_i \in \mu(q')$) and each interface variable ($v_i \in \mathcal{P}_i.V^I$) we bind v to its associated value in $\beta(q', \mathcal{P}_i)$. Note that $\fatsemi$ converts a set of effects to a single compound effect.

The final condition deals with sub-protocols that have designated final states. As explained earlier, the semantics of final states in sub-protocols is that any state that has a sub-protocol cannot be transitioned out of, unless the sub-protocol is in a final state. This is realised by only generating transitions that satisfy this condition, that is, for any sub-protocol ($\bigwedge_{\mathcal{P}' \in \mu(q)}$), either that sub-protocol has no final states ($\mathcal{P}'.Q^F = \emptyset$), or the sub-protocol must be in one of its final states.

Note that in the definition above the message σ can also be the empty message σ_ϵ: if $\mathcal{P}.\delta(q, \sigma_\epsilon, g) = (q', e)$ then there will be a transition $[\![\mathcal{P}]\!].\delta((q, w), \sigma_\epsilon, g)$.

We also have a similarly structured definition that applies when we are transitioning from a state that does not have sub-protocols. Since in this case there are no sub-protocols, the last condition does not need to be checked.

We now turn to the second case, where sub-protocols transition concurrently on shared messages (if in the appropriate states). This is somewhat more complex in our setting than in other settings (such as Alur et al. [1]) because we associate guards and effects with a message. The intuition is that where there are two or more machines that share a message σ, then they are required to transition simultaneously.

In order to ease the exposition, we first present a simplified rule that ignores guards and effects, and then extend it. If we ignore guards and effects, then we define a transition $[\![\mathcal{P}]\!].\delta((q, w), \sigma, g) = ((q, w'), e)$ where $w = (w_1, \ldots, w_n)$ and $w' = (w'_1, \ldots, w'_n)$ so that each sub-protocol that has a transition for σ is required to have a transition on σ in state w, i.e. $[\![\mathcal{P}_i]\!].\delta(w_i, \sigma, g_i) = (w'_i, e_i)$, and for a sub-protocol that does not have a transition for σ, we have $w'_j = w_j$. This gives the following definition:

$$
\begin{aligned}
[\![\mathcal{P}]\!].\delta((q, w), \sigma, g) = {} & ((q, w'), e) \\
& \text{where } w = (w_1, \ldots, w_n) \text{ and } w' = (w'_1, \ldots, w'_n) \\
& \text{and } (q, w) \in [\![\mathcal{P}]\!].Q \wedge \sigma \neq \sigma_\epsilon \wedge \sigma \in [\![\mathcal{P}]\!].\Sigma \\
& \text{and for each sub-protocol } \mathcal{P}_i \text{ either:} \\
& \left(\neg \exists \hat{w}_i, \hat{w}'_i, \hat{g}_i, \hat{e}_i : [\![\mathcal{P}_i]\!].\delta(\hat{w}_i, \sigma, \hat{g}_i) = (\hat{w}'_i, \hat{e}_i)\right) \wedge w'_i = w_i \\
& \text{or } [\![\mathcal{P}_i]\!].\delta(w_i, \sigma, g_i) = (w'_i, e_i)
\end{aligned}
$$

We require that (q, w) is a valid state, and that σ is a valid (non-empty) message. We also require that at least one protocol has a transition $(\exists i : [\![\mathcal{P}_i]\!].\delta(w_i, \sigma, g_i) = (w'_i, e_i))$.

However, in the case of the empty message σ_ϵ, there are no synchronisation constraints, so a transition on (q, w) involves exactly one of the sub-protocols transitioning:

$$
\begin{aligned}
[\![\mathcal{P}]\!].\delta((q, w), \sigma_\epsilon, g_i) = {} & ((q, w'), e_i) \\
& \text{where } w = (w_1, \ldots, w_n) \text{ and } w' = (w'_1, \ldots, w'_n) \\
& \text{and } [\![\mathcal{P}_i]\!].\delta(w_i, \sigma_\epsilon, g_i) = (w'_i, e_i) \text{ and } w'_j = w_j \text{ for } j \neq i
\end{aligned}
$$

In order to extend this to deal with guards and effects we define the effect e and guard g of the transition in $[\![\mathcal{P}]\!]$. We define $e = e_1; \ldots; e_n$ where each e_i is either the effect of the transition on σ in the sub-protocol $\mathcal{P}_i$ (if the protocol has a transition on σ) or is the empty effect ϵ. Similarly, we define $g = g_1 \wedge \cdots \wedge g_n$ where each g_i is either the guard of the relevant transition in the sub-protocol $\mathcal{P}_i$, or is $\top$.

We then define:

$$[\![\mathcal{P}]\!].\delta((q, w), \sigma, g) = ((q, w'), e)$$

where $\sigma \neq \sigma_\epsilon$, $\sigma \in [\![\mathcal{P}]\!].\Sigma$, $(q, w) \in [\![P]\!].Q$, $w = (w_1, \ldots, w_n)$, $w' = (w'_1, \ldots, w'_n)$, and where we define $e \equiv e_1; \ldots; e_n$ and $g \equiv g_1 \wedge .s \wedge g_n$, if there exists guards g_i and effects

e_i such that for each sub-protocol $\mathcal{P}_i$:

1. if protocol $\mathcal{P}_i$ has any transition for message σ then there must be a transition for σ from the current state w, $[\![\mathcal{P}_i]\!].\delta(w_i, \sigma, \hat{g}_i) = (\hat{w}'_i, \hat{e}_i)$, and $w'_i = \hat{w}'_i$ and $e_i = \hat{e}_i$ and $g_i = \hat{g}_i$.
2. else (if $\mathcal{P}_i$ does not have any transition on σ) then $w'_i = w_i$ and $e_i = \epsilon$ and $g_i = \top$.

As before we also require that at least one protocol does have a transition.

Collecting these cases, we have the definition for the flattened state machine $[\![\mathcal{P}]\!] = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle$ given in Fig. 3. The next section illustrates these definitions by giving the flattened version of the example booking protocol discussed in Sect. 3.

4.9 Mapping informal to formal

At the start of this section we commented that these formal structures are not intended for use by designers, who would use the graphical notation (e.g. Fig. 1). We now briefly discuss the mapping from the graphical notation to the formal structures, illustrating with the simple booking example protocol that was discussed in Sect.3.

The protocol in question uses three variables. Two local variables: `req` (which is bound to the name of the Movie being requested), and `paid` (a Boolean indicating whether payment has been made); and a system variable `seats` (the number of seats available). Additionally, there are two variables that appear in the messages (*Movie* and *Day*) which are required to be in the protocol's set of variables. Note that `req` appears in the variable set of the top-level protocol whereas `paid`, which is owned by the payment sub-protocol, appears in that protocol's variable set. The system variable `seats` is in the variable set of all protocols that use it.

Each of the three protocols in Fig. 1 is mapped to a tuple. The first protocol (booking) is mapped to the tuple $\mathcal{P}_\mathcal{B} = \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle$ where the name N is $\mathcal{B}$, there are no interface variables ($V^I = \emptyset$), and Q is the set of states $\{s_0, s_1, s_2\}$. The elements V, Σ, G and E are respectively the set of variables, messages, guards, and effects that appear in the protocol. The starting state q^0 is just s_0 and there is only a single final state, so $Q^F = \{s_2\}$. Finally, δ captures the two transitions (see the equations below, noting that δ is undefined for any other inputs).

$$
\begin{aligned}
\mathcal{P}_\mathcal{B} &= \langle N, V^I, Q, V, \Sigma, G, E, q^0, Q^F, \delta\rangle \\
N &= \mathcal{B} \\
V^I &= \emptyset \\
Q &= \{s_0, s_1, s_2\} \\
V &= \{\texttt{req}, \texttt{seats}, \mathit{Movie}, \mathit{Day}\} \\
\Sigma &= \{\mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{book}(\mathit{Movie}, \mathit{Day}), \mathsf{S} \rightarrow \mathsf{U}\colon \mathsf{confirm}(\mathit{req})\} \\
G &= \{\mathit{seats} > 0, \texttt{paid}\} \\
E &= \{\mathsf{bind}(\texttt{req}, \mathit{Movie})\} \\
q^0 &= s_0 \\
Q^F &= \{s_2\} \\
\delta &= \{(s_0, \mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{book}(\mathit{Movie}, \mathit{Day}), \texttt{seats}{>}0) \mapsto (s_1, \mathsf{bind}(\texttt{req}, \mathit{Movie})), \\
&\quad (s_1, \mathsf{S} \rightarrow \mathsf{U}\colon \mathsf{confirm}(\mathit{req}), \texttt{paid}) \mapsto (s_2, \epsilon)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathcal{P}]\!].N &= \mathcal{P}_0.N \\
[\![\mathcal{P}]\!].V^I &= \mathcal{P}_0.V^I \\
[\![\mathcal{P}]\!].Q &= \{q \mid q \in \mathcal{P}_0.Q \wedge \mu(q) = \emptyset\} \\
&\quad \cup \{(q,w) \mid q \in \mathcal{P}_0.Q \wedge \mu(q) \neq \emptyset \wedge w \in f(\mu(q))\} \\
&\quad \text{where } f(\{\mathcal{P}_1,\ldots,\mathcal{P}_n\}) = \{(q_1,\ldots,q_n) \mid q_i \in [\![\mathcal{P}_i]\!].Q\} \\
[\![\mathcal{P}]\!].V &= \mathcal{P}_0.V \cup \bigcup_{1\leq i\leq n} \mathcal{P}_i.V \cup \mathcal{P}_i.V^I \qquad [\![\mathcal{P}]\!].\Sigma = \bigcup_{0\leq i\leq n} \mathcal{P}_i.\Sigma \\
[\![\mathcal{P}]\!].G &= \bigcup_{0\leq i\leq n} \mathcal{P}_i.G \qquad [\![\mathcal{P}]\!].E = \bigcup_{0\leq i\leq n} \mathcal{P}_i.E \\
[\![\mathcal{P}]\!].q^0 &= \begin{cases} \mathcal{P}_0.q^0 & \text{if } \mu(\mathcal{P}_0.q^0) = \emptyset \\ (\mathcal{P}_0.q^0, f^0(\mu(\mathcal{P}_0.q^0))) & \text{otherwise} \end{cases} \\
&\quad \text{where } f^0(\{\mathcal{P}_1,\ldots,\mathcal{P}_n\}) = ([\![\mathcal{P}_1]\!].q^0,\ldots,[\![\mathcal{P}_n]\!].q^0) \\
[\![\mathcal{P}]\!].Q^F &= \{q \mid q \in \mathcal{P}_0.Q^F \wedge \mu(q) = \emptyset\} \cup \{(q,w) \mid q \in \mathcal{P}_0.Q^F \wedge (q,w) \in [\![\mathcal{P}]\!].Q\} \\
[\![\mathcal{P}]\!].\delta((q,w),\sigma,g) &= \begin{cases} ((q', f^0(\mu(q'))), e^+) & \text{if } \mathcal{P}.\delta(q,\sigma,g) = (q',e) \ \wedge \ \mu(q') \neq \emptyset \\ (q',e) & \text{if } \mathcal{P}.\delta(q,\sigma,g) = (q',e) \ \wedge \ \mu(q') = \emptyset \end{cases} \\
&\quad \text{where } (q,w) \in [\![\mathcal{P}]\!].Q \\
&\quad \text{and } e^+ = e \,;\, \left(\fatsemi \{\mathsf{bind}(v_i, \beta(q',\mathcal{P}_i)(v_i)) \mid \mathcal{P}_i \in \mu(q') \wedge v_i \in \mathcal{P}_i.V^I\}\right) \\
&\quad \text{and } \fatsemi\{e_1,\ldots,e_n\} = e_1;\ldots;e_n \\
&\quad \text{and } \bigwedge_{\mathcal{P}'\in\mu(q)} \left(\mathcal{P}'.Q^F = \emptyset\right) \ \vee \ \left(\bigvee_{q^f \in \mathcal{P}'.Q^F} \mathsf{in}(\mathcal{P}'.N, q^f, (q,w))\right) \\
[\![\mathcal{P}]\!].\delta(q,\sigma,g) &= \begin{cases} ((q', f^0(\mu(q'))), e^+) & \text{if } \mathcal{P}.\delta(q,\sigma,g) = (q',e) \ \wedge \ \mu(q') \neq \emptyset \\ (q',e) & \text{if } \mathcal{P}.\delta(q,\sigma,g) = (q',e) \ \wedge \ \mu(q') = \emptyset \end{cases} \\
&\quad \text{where } \mu(q) = \emptyset \\
&\quad \text{and } e^+ = e \,;\, \left(\fatsemi \{\mathsf{bind}(v_i, \beta(q',\mathcal{P}_i)(v_i)) \mid \mathcal{P}_i \in \mu(q') \wedge v_i \in \mathcal{P}_i.V^I\}\right) \\
&\quad \text{and } \fatsemi\{e_1,\ldots,e_n\} = e_1;\ldots;e_n \\
[\![\mathcal{P}]\!].\delta((q,w),\sigma,g) &= ((q,w'),e) \\
&\quad \text{where } w = (w_1,\ldots,w_n) \text{ and } w' = (w'_1,\ldots,w'_n) \\
&\quad \text{and } e = e_1;\ldots;e_n \text{ and } g = g_1 \wedge \ldots \wedge g_n \\
&\quad \text{and } (q,w) \in [\![\mathcal{P}]\!].Q \wedge \sigma \neq \sigma_\epsilon \wedge \sigma \in [\![\mathcal{P}]\!].\Sigma \\
&\quad \text{and } \forall \mathcal{P}_i \in \mu(q) \ \text{ if } \exists \hat{w}_i, \hat{w}'_i, \hat{g}_i, \hat{e}_i : [\![\mathcal{P}_i]\!].\delta(\hat{w}_i,\sigma,\hat{g}_i) = (\hat{w}'_i,\hat{e}_i) \text{ then} \\
&\qquad [\![\mathcal{P}_i]\!].\delta(w_i,\sigma,\hat{g}_i) = (\hat{w}'_i,\hat{e}_i) \ \wedge g_i = \hat{g}_i \wedge w'_i = \hat{w}'_i \wedge e_i = \hat{e}_i \\
&\quad \text{else } w'_i = w_i \wedge e_i = \epsilon \wedge g_i = \top \\
&\quad \text{and } \exists i : [\![\mathcal{P}_i]\!].\delta(w_i,\sigma,g_i) = (w'_i,e_i) \\
[\![\mathcal{P}]\!].\delta((q,w),\sigma_\epsilon,g_i) &= ((q,w'),e_i) \\
&\quad \text{where } w = (w_1,\ldots,w_n) \text{ and } w' = (w'_1,\ldots,w'_n) \\
&\quad \text{and } [\![\mathcal{P}_i]\!].\delta(w_i,\sigma_\epsilon,g_i) = (w'_i,e_i) \text{ and } w'_j = w_j \text{ for } j \neq i
\end{aligned}
$$

Fig. 3 Semantics of Hierarchical FSMs

The two sub-protocols are similarly mapped:

$$
\begin{aligned}
\mathcal{P}_{\mathcal{P}} &= \langle N_1, V_1^I, Q_1, V_1, \Sigma_1, G_1, E_1, q_1^0, Q_1^F, \delta_1 \rangle \\
N_1 &= \mathcal{P} \\
V_1^I &= \emptyset \\
Q_1 &= \{p_0, p_1, p_2\} \\
V_1 &= \{\texttt{paid}\} \\
\Sigma_1 &= \{\mathsf{S} \rightarrow \mathsf{U}\colon \mathsf{pay}(), \mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{payment}()\} \\
G_1 &= \emptyset \\
E_1 &= \{\mathsf{bind}(\texttt{paid}, \top)\} \\
q_1^0 &= p_0 \\
Q_1^F &= \emptyset \\
\delta_1 &= \{(p_0, \mathsf{S} \rightarrow \mathsf{U}\colon \mathsf{pay}(), \top) \mapsto (p_1, \epsilon), \\
&\qquad (p_1, \mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{payment}(), \top) \mapsto (p_2, \mathsf{bind}(\texttt{paid}, \top))\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}_{\mathcal{R}} &= \langle N_2, V_2^I, Q_2, V_2, \Sigma_2, G_2, E_2, q_2^0, Q_2^F, \delta_2 \rangle \\
N_2 &= \mathcal{R} \\
V_2^I &= \emptyset \\
Q_2 &= \{r_0, r_1\} \\
V_2 &= \{\texttt{seats}\} \\
\Sigma_2 &= \{\mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{payment}()\} \\
G_2 &= \emptyset \\
E_2 &= \{\mathsf{bind}(\texttt{seats}, \texttt{seats} - 1), \mathbf{S}\colon \mathbf{updatePoints}()\} \\
q_2^0 &= r_0 \\
Q_2^F &= \emptyset \\
\delta_2 &= \{(r_0, \mathsf{U} \rightarrow \mathsf{S}\colon \mathsf{payment}(), \top) \mapsto \\
&\qquad (r_1, \mathsf{bind}(\texttt{seats}, \texttt{seats} - 1); \mathbf{S}\colon \mathbf{updatePoints}())\}
\end{aligned}
$$

Finally, the top-level protocol is defined as:

$$\mathcal{P} = \langle \mathcal{P}_{\mathcal{B}}, \{\mathcal{P}_{\mathcal{P}}, \mathcal{P}_{\mathcal{R}}\}, \mu, \beta \rangle$$

where

$$\mu(Q) = \begin{cases} \{\mathcal{P}_{\mathcal{P}}, \mathcal{P}_{\mathcal{R}}\} & \text{if } Q = s_1 \\ \emptyset & \text{otherwise} \end{cases}$$

(since both sub-protocols have $V^I = \emptyset$ the function β is undefined and not used).

Applying the definitions in the previous section we obtain the following flattened protocol. Note that Q includes all possible states, not all of which are actually reachable. For example, the state $(s_1, (p_0, r_1))$ is not reachable, since, due to the synchronisation constraint on shared messages, we cannot transition from r_0 to r_1 without also transitioning from p_1 to p_2.

$$
\begin{aligned}
[\![\mathcal{P}]\!] &= \langle \mathcal{B}, V^I, Q, V, \Sigma, G, E, q^0 = s_0, Q^F = \{s_2\}, \delta \rangle \\
V^I &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
Q &= \{s_0, s_2, (s_1, (p_0, r_0)), (s_1, (p_0, r_1)), (s_1, (p_1, r_0)), (s_1, (p_1, r_1)),\\
&\quad (s_1, (p_2, r_0)), (s_1, (p_2, r_1))\}\\
V &= \{\texttt{req}, \texttt{seats}, \texttt{paid}, \mathit{Movie}, \mathit{Day}\}\\
\Sigma &= \{\mathsf{U} \to \mathsf{S}\colon \mathsf{book}(\mathit{Movie}, \mathit{Day}), \mathsf{S} \to \mathsf{U}\colon \mathsf{confirm}(\mathit{req}), \mathsf{S} \to \mathsf{U}\colon \mathsf{pay}(),\\
&\quad \mathsf{U} \to \mathsf{S}\colon \mathsf{payment}()\}\\
G &= \{\mathit{seats} > 0, \texttt{paid}\}\\
E &= \{\mathsf{bind}(\texttt{req}, \mathit{Movie}), \mathsf{bind}(\texttt{paid}, \top)\}, \mathsf{bind}(\texttt{seats}, \texttt{seats} - 1),\\
&\quad \mathbf{S}\colon \mathbf{updatePoints}()\}\\
\delta &= \{(s_0, \mathsf{U} \to \mathsf{S}\colon \mathsf{book}(\mathit{Movie}, \mathit{Day}), \texttt{seats}{>}0) \mapsto ((s_1, (p_0, r_0)),\\
&\quad \mathsf{bind}(\texttt{req}, \mathit{Movie})),\\
&\quad ((s_1, (p_0, r_0)), \mathsf{S} \to \mathsf{U}\colon \mathsf{pay}(), \top) \mapsto ((s_1, (p_1, r_0)), \epsilon),\\
&\quad ((s_1, (p_1, r_0)), \mathsf{U} \to \mathsf{S}\colon \mathsf{payment}(), \top) \mapsto ((s_1, (p_2, r_1)),\\
&\quad \mathsf{bind}(\texttt{paid}, \top); \mathsf{bind}(\texttt{seats}, \texttt{seats} - 1);\\
&\quad \mathbf{S}\colon \mathbf{updatePoints}()),\\
&\quad ((s_1, (p_2, r_1)), \mathsf{S} \to \mathsf{U}\colon \mathsf{confirm}(\mathit{req}), \texttt{paid}) \mapsto (s_2, \epsilon),\\
&\quad \ldots \textit{ see text below} \ldots\\
&\}
\end{aligned}
$$

Note that the definition of $[\![\mathcal{P}]\!].\delta$ also generates some additional transitions:

1. transitions to s_2 from each of the (unreachable) states $(s_1, (p_0, r_1))$, $(s_1, (p_1, r_1))$ and $(s_1, (p_2, r_0))$;
2. transitions from $(s_1, (p_0, r_0))$ and $(s_1, (p_1, r_0))$ to s_2: these are from reachable states, but because the transitions have the guard `paid`, which only becomes true in p_2, these two transitions are not able to be taken, since the guard will be false; and
3. a transition between two (unreachable) states: $(s_1, (p_0, r_1))$ to $(s_1, (p_1, r_1))$.

Figure 4 shows the flattened protocol. A dashed circle is used to indicate unreachable states, and a dashed arrow indicates a transition that cannot be taken. This figure was generated using a Prolog program encoding the semantics of Fig. 3. Note that in the figure "action(s:updatePoints)" denotes **S**:**updatePoints**() and "m(u,s,msg)" denotes $\mathsf{u} \to \mathsf{m}\colon \mathsf{msg}()$ (and similarly for other messages).

We now consider the variant protocol (Fig. 2) where the transition from s_1 has no guard, but $\mathcal{P}_\mathcal{P}$ has p_2 as a final state, and therefore the transition from s_1 has an implicit guard condition. The translation is identical to that above except for the following:

$$
\begin{aligned}
\mathcal{P}_\mathcal{B}.\delta &= \{(s_0, \mathsf{U} \to \mathsf{S}\colon \mathsf{book}(\mathit{Movie}, \mathit{Day}), \texttt{seats}{>}0) \mapsto (s_1, \mathsf{bind}(\texttt{req}, \mathit{Movie})),\\
&\quad (s_1, \mathsf{S} \to \mathsf{U}\colon \mathsf{confirm}(\mathit{req}), \top) \mapsto (s_2, \epsilon)\}\\
\mathcal{P}_\mathcal{B}.G &= \{\mathit{seats} > 0\}\\
\mathcal{P}_\mathcal{P}.Q_1^F &= \{p_2\}\\
\mathcal{P}_\mathcal{P}.V_1 &= \emptyset\\
\mathcal{P}_\mathcal{P}.E_1 &= \emptyset\\
\mathcal{P}_\mathcal{P}.\delta_1 &= \{(p_0, \mathsf{S} \to \mathsf{U}\colon \mathsf{pay}(), \top) \mapsto (p_1, \epsilon), (p_1, \mathsf{U} \to \mathsf{S}\colon \mathsf{payment}(), \top) \mapsto (p_2, \epsilon)\}
\end{aligned}
$$

Applying the definitions of Fig. 3 we obtain a flattened protocol (Fig. 5) that differs from Fig. 4 in that the extra transitions from $(s_1, (p_i, r_j))$ do not exist: there is only a transition

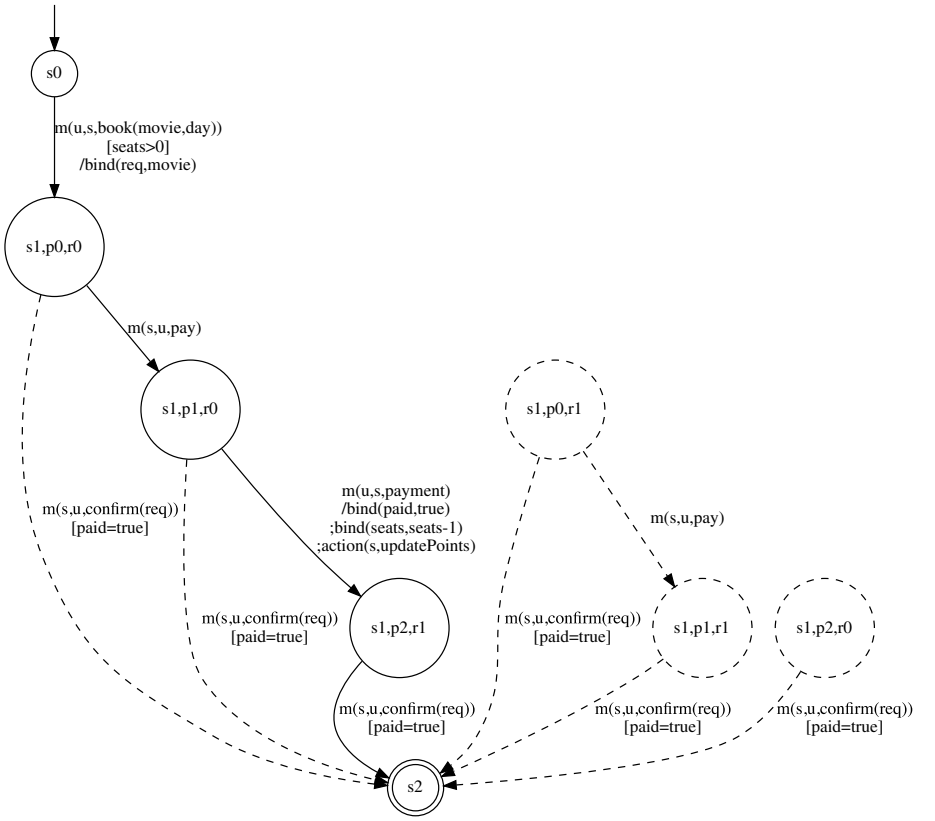


Fig. 4 Flattened booking protocol

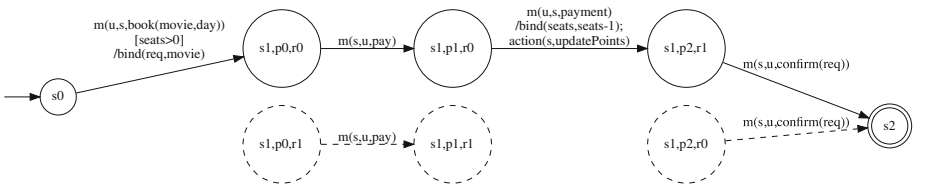


Fig. 5 Flattened variant booking protocol

from $(s_1, (p_2, r_1))$ and a transition from the unreachable state $(s_1, (p_2, r_0))$ to s_2, as well as a transition between two unreachable states $((s_1, (p_0, r_1))$ to $(s_1, (p_1, r_1)))$. It also obviously differs in the removal of the guard `paid` from both G and δ, and in the removal of the effect binding `paid`.

5 Case studies

We now present three case studies that were chosen to illustrate a wide range of scenarios. The first case study, the play date, illustrates an information-driven interaction. The second,

an auction, illustrates the use of hierarchical protocols, with a protocol instance for each bidder, to capture clearly and precisely an auction interaction. The third case study, Holonic manufacturing, shows how the hierarchical machines allow sub-protocols to be specified in a modular fashion (supporting reuse), in a way that cannot be done in AUML.

5.1 Play date

The play date example arose in the context of work we were doing with an industry partner on an intelligent toy that communicates with its child user, providing various services via independent agents that are integrated by the toy controller.

The play date organisation functionality is essentially a simple meeting organisation, where three parameters must be determined: who to play with (person: *per*), where to play (location: *loc*), and when (*day*). The difficulty came with the level of flexibility that was required in this interaction, due to it being an interaction between a (child) human user and the system, combined with the requirement of sufficient message specificity to recognise messages as belonging to this protocol. Values for the parameters (*per*, *loc*, *day*) should be able to be spontaneously provided by the child, or requested by the system. They should be able to be provided in any order, and a request regarding some variable may result in a response regarding that or any other variable. In addition, during the interaction, the child may change his/her mind about the value of any of these variables, e.g. changing to play with Alex instead of Pat, or to play on Saturday instead of Sunday. However, there are some constraints on the order: the system can only request the value for a variable that has not already been provided, and the user can only ask the system to forget a value when one has been previously provided. Although this was programmed satisfactorily, attempts to capture it properly in a protocol specification using AUML or other notations, were both awkward and complex (see below). An added complication was that the child should be able to cancel the interaction at any point prior to confirmation.

We take first the flexible approach to determining the value of the three key variables. We have defined a general purpose protocol which allows the flexible binding of any set of variables, by using an interface variable V which is bound at initialisation to a particular set of variables, in this case *per*, *loc* and *day* (since this protocol is not hierarchical, the scope of the variables doesn't matter). Looking at the start state t_0 and end state t_1 and their associated transitions (Fig. 6, ignoring for now states t_2 and t_3) we see that this flexibility is achieved by a simple cycle with three possible transitions: a message where the system can request the value of any unbound variable; a message where the user can provide the value of any variable, with the effect of it then becoming bound to that value; and a forget message from user to system which also allows for an explicit unbinding of any variable. When all variables are bound, the system can issue a confirmation message and the protocol transitions to its final state t_1. If a cancel message occurs then if nothing is bound this transitions directly to a final state, else it transitions to an intermediate state t_3, where there is a cycle of the system doing cancellation actions (e.g. removing bookings from a calendar), and unbinding for each bound variable v, until the guard requiring all variables to be unbound is true, and the system can transition to the final state. This cycle (t_3 self loop) does not involve a message. It illustrates the use of domain actions to allow the protocol to capture an important constraint: that all bookings made are undone if a cancelPD message is received. An alternative protocol that does not capture this would be simpler (no need for t_3).

We believe this representation captures clearly what messages (including sufficient content information) can be expected at any stage of the interaction, at the same time as being simple and easy to follow. This scenario is an example of a flexible information-driven

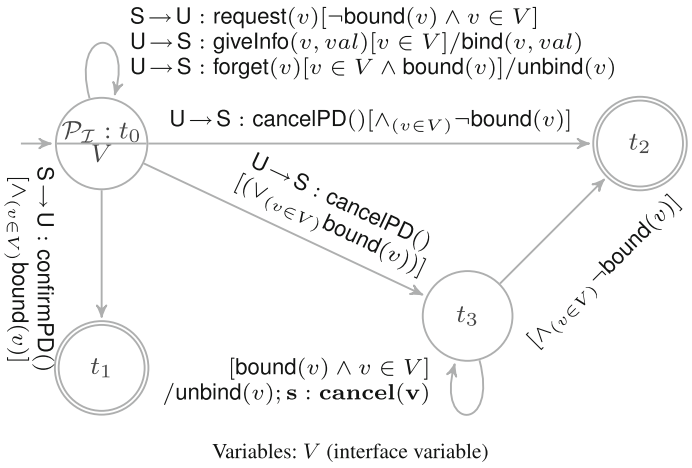


Fig. 6 Play date protocol

interaction. The HAPN protocol uses variables in guards to correctly capture the constraints on an otherwise unconstrained interaction (all messages, except for confirm and cancel, remain in state t_0).

By contrast, in AUML we could either have very generic messages (request and response), or specific messages that included some content information in the message type, e.g. request-date, and respond-date (a third option is to use variables and model the play date protocol in a similar way to HAPN, but this is not idiomatic AUML, and we consider this in Sect. 6.1).

The former, using very generic messages, (see e.g. Fig. 7) resulted in the protocol being unable to specify precisely conditions over all desired variables being bound. For example, in this AUML protocol the system can ask for the location even if it has already been provided. It is possible to have an AUML protocol with generic messages capture these conditions by moving the logic to an external entity. For instance, having an artefact, or some other external database or information system, that is updated whenever the user sends a message, and that tracks which information has been provided. This approach is problematic, since the AUML protocol is no longer complete: in order to determine which sequences of messages are legal, one must also specify and consider the logic of the artefact. It is also possible to use a simpler form of artefact (similar to the notion of business artefact used for business modelling [8,14,32]), where the artefact merely collects related information, but does not itself include any processing or logic. In this case the protocol remains similar: instead of referring to individual variables (day, place), it refers to fields of the artefact. The protocol still includes conditions over the values of the artefact's fields that are required to be able to understand which interactions are intended.

On the other hand, the latter, using specific messages such as "request-date", resulted in complex nesting of alternative options which was difficult to understand as well as to specify correctly. In addition, modelling any point cancellation was awkward as it had to be replicated at all the places where a user could potentially cancel the booking. In order to illustrate the complexity, Fig. 8 shows a Finite State Machine that captures the messages for a simplified version of the protocol that only has two variables.[5] As can be seen, capturing

[5] Notation: r1 and r2 denote "request variable 1" and "request variable 2" respectively, "g1" denotes "give information for variable 1", "f1" denotes "forget information for variable 1", "c" is the cancel request, "c1" is "cancel related to variable 1", and "conf" is "confirm".

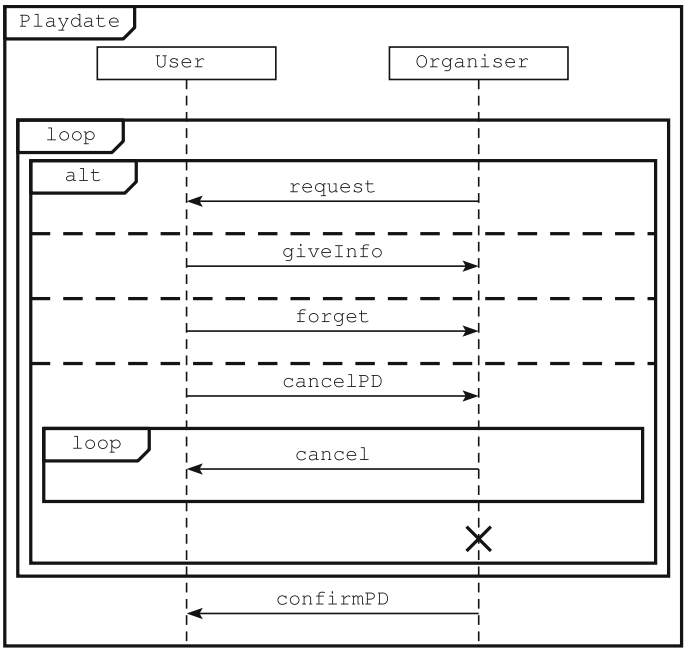


Fig. 7 Play date protocol in AUML—simple version

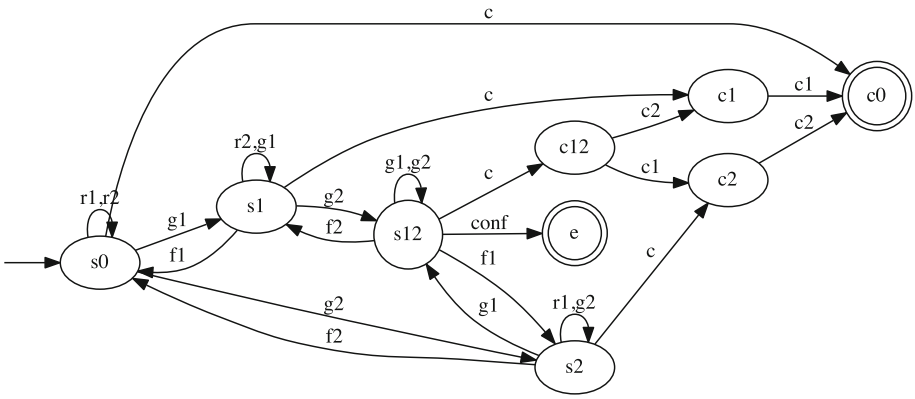
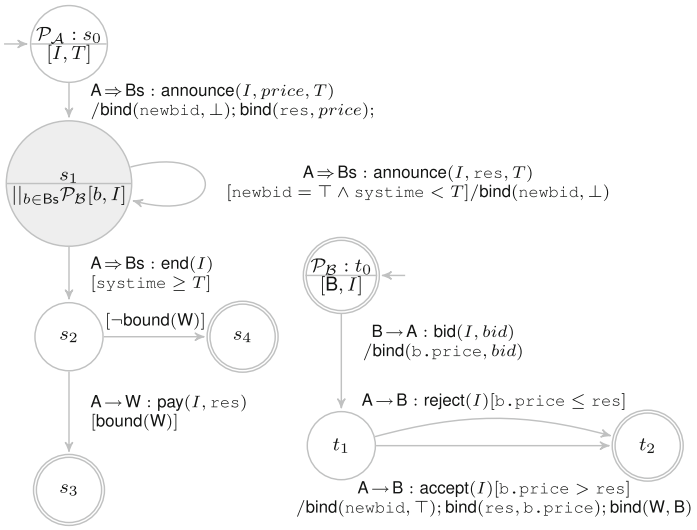


Fig. 8 Finite State Machine for play date protocol (for two variables)

the correct states and sequences is quite complex. For instance, if we have provided a value for the first variable, then we cannot request a value for this variable, and if we cancel the protocol, then we must cancel the information associated with variable 1 before we can finish. The FSM in Fig. 8 captures the inherent complexity of modelling this interaction using only messages. This complexity also would apply to an AUML version that did not (informally) use variables. There are a number of ways of mapping this FSM to AUML, the most obvious using goto/labels.

The key point is that modelling information-driven protocols can be done simply if the notation provides variables, but cannot be modelled satisfactorily only in terms of (generic)



Variables: newbid, res and W (all owned by $\mathcal{P}_\mathcal{A}$, protected/public); Bs and $price$ (message variables in $\mathcal{P}_\mathcal{A}$); bid (message variable in $\mathcal{P}_\mathcal{B}$); systime (system variable); b.price ($\mathcal{P}_\mathcal{B}$, private); I, T (interface variables of $\mathcal{P}_\mathcal{A}$); B, I (interface variables of $\mathcal{P}_\mathcal{B}$).

Fig. 9 Auction protocol in HAPN

messages. Since AUML does not provide variables (see Sect. 6.1 for a more detailed discussion), it cannot model information-driven protocols effectively, without being extended in some way, such as adding artefacts.

5.2 Auction

Auction protocols have been widely used in the literature to illustrate protocol notation and its various nuances, especially with respect to AUML (e.g. [22,41])

Figures 9 and 10 show a HAPN representation and a typical representation in AUML, respectively, of an English auction. This is based on an electronic English auction with an ending time, rather than a (usually physical) auction where it ends because there are no more bids within some (short) timeframe.

The left side of Fig. 9 shows the top level protocol $\mathcal{P}_\mathcal{A}$, with the start node having interface variables I for the item, and T for the finishing time, which are bound for the particular auction instance. The first transition occurs when the auctioneer broadcasts[6] an announcement stating the item, the starting price, and the finish time. The variables newbid and res are bound to ⊥ (False) and the starting price respectively, as the effect of this transition.

The state s_1 has a bidding sub-protocol, which conceptually is a set of protocol instances, one for each bidder, with the particular bidder bound to b, and the item to I as previously.[7] The sub-protocol $\mathcal{P}_\mathcal{B}$ (right side of Fig. 9) then specifies the interaction between the auctioneer and an individual bidder. In this interaction, the first transition is when a bidder makes a bid

[6] Bs—note that we use an "s" to highlight that this is a set of agents, and hence that a message A → Bs is a broadcast, which we highlight by using a double arrow ⇒ instead of the usual →.

[7] In an electronic auction such as ebay where the set of potential bidders is huge, and the set of actual bidders is initially unknown, the individual conversation/protocol instances would be lazily instantiated as potential bidders actually interacted.

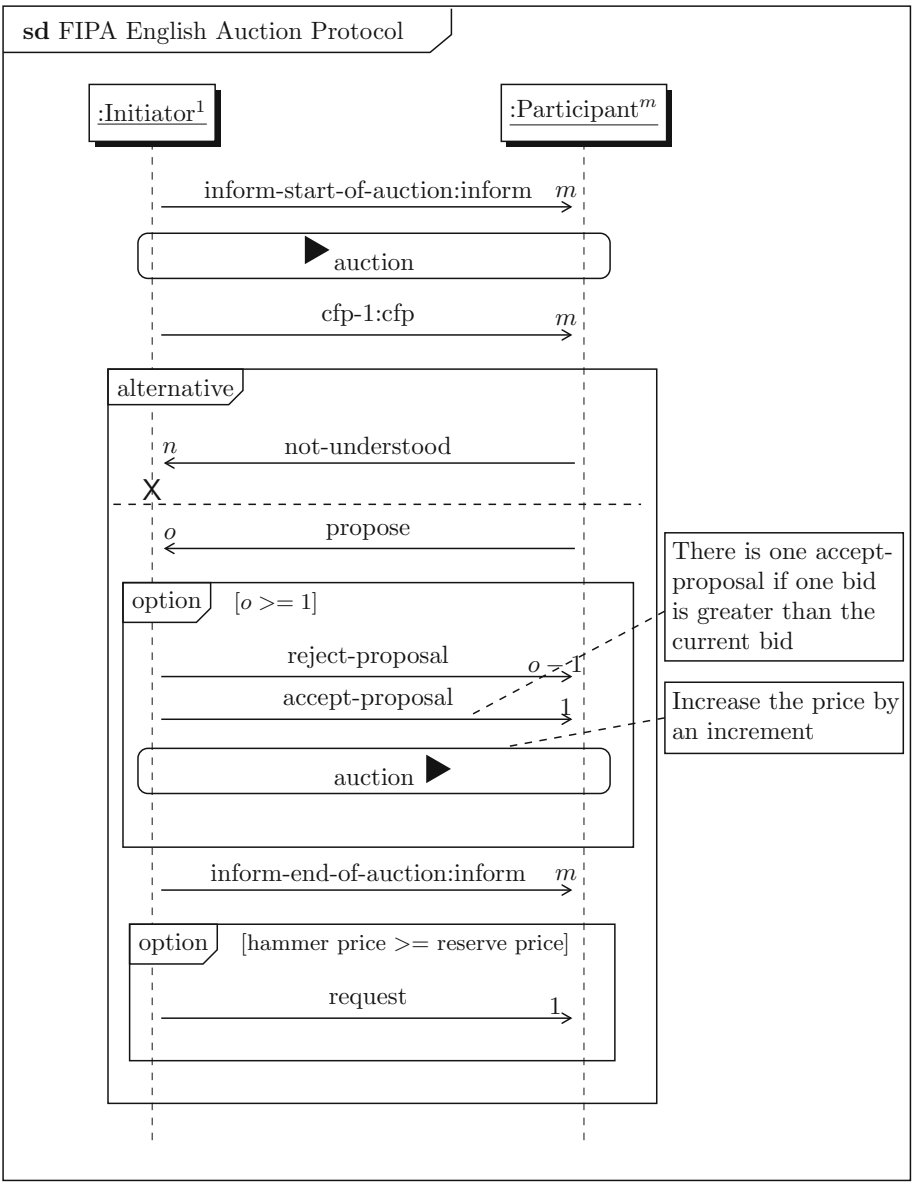
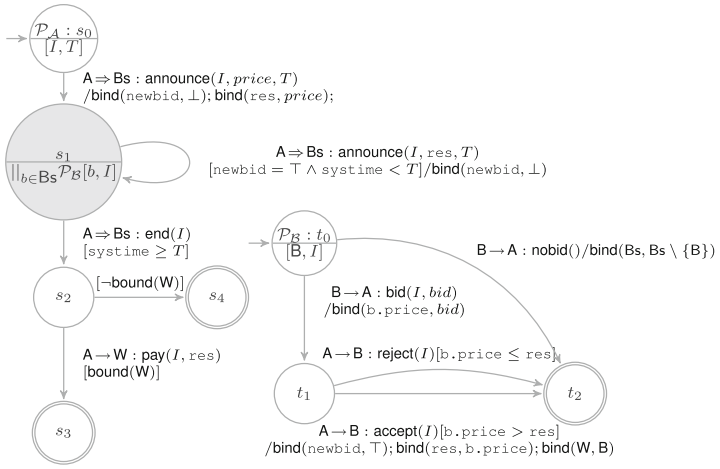


Fig. 10 English auction protocol in AUML (redrawn from Figure 31 of [20])

for the item, resulting in `b.price` (the price of this bidder) being set to what was offered, and moving to state t_1. There are then two possible transitions to state t_2. One is where the auctioneer rejects the bid, because the offer (`b.price`) is less than or equal to the current reserve (`res`). The other is where the auctioneer accepts the bid, with offer being greater than the reserve. This then has the effect of binding the reserve to the offer (`res` to `b.price`), setting the variable `newbid` to True ($\top$), and setting the current winner (W) to be this bidder. If multiple bidders send bids of equal value, then only the bid that reaches the auctioneer first is accepted. Acceptance of the first bid results in the value of the reserve price being updated



Compared with the previous version, this protocol requires each bidder to either bid or indicate that they are not bidding (nobid). There is a new transition from t_0 to t_2 for the nobid message, and t_0 is no longer a final state. Variables are identical to the previous version.

Fig. 11 Realisable Auction protocol in HAPN

to be equal to the first bid. Hence, all subsequent bids that are equal or lower than this price are rejected.

Once one or more bids have come in, their interactions have completed, and at least one bid has been accepted, control returns to the top level protocol, where (as long as time has not expired) the auctioneer again broadcasts the item and new reserve price. Note that a new round (self loop on s_1) can only be started if at least one bid was accepted (guard $\mathtt{newbid} = \top$) which implies that an acceptable bid was received, and hence that the reserve price has increased.

When time expires, s_1 transitions to s_2 with the auctioneer broadcasting an end message. The auctioneer then sends a message to the winner, if there is one, requesting payment of res for the item I. If no-one made a bid which was accepted, then W will not have been bound, and the protocol transitions to s_4, where it ends.

Note that the sub-protocol $\mathcal{P}_\mathcal{B}$ has states t_0 and t_2 indicated as final states. As discussed earlier, this constrains the transition from s_1 to s_2 so that it cannot occur if any instance of $\mathcal{P}_\mathcal{B}$ is in state t_1. However, this demonstrates an issue in the design of protocols with a global viewpoint. As discussed earlier, by providing the designer with the convenient fiction that messages are synchronous, and that the protocol can be viewed from a global perspective, we make it possible to specify unrealisable protocols. In this case, the protocol given in Fig. 9 is unrealisable, since there is a race condition between a bidder sending a bid ($t_0 \rightarrow t_1$) and the auctioneer sending an end or an announce ($s_1 \rightarrow s_2$ or $s_1 \rightarrow s_1$). As discussed earlier, the design process for protocols involves checking for realisability, and fixing unrealisable protocols. In this case one way to make the protocol realisable is to make it compulsory for each bidder to either bid, or to indicate that they are not bidding (see Fig. 11).

We believe that the HAPN auction protocol is simple and clear. Comparing it with a typical AUML representation of an auction protocol (as in Fig. 10, which is redrawn from [20, Figure 31]) we highlight that the AUML protocol[8]:

[8] Note that this protocol, which is taken from the literature, is not a great design (e.g. using goto/label instead of a loop, and terminating the Initiator when a not-understood message is sent).

1. Relies on English text to explain certain relationships, for instance the text indicating that there is one accept-proposal if a bid higher than the current bid is received. This is problematic since English is notoriously ambiguous. Indeed, in this case the text appears to not deal with the case where two equal bids are received, and, in the case where none of the bids received in a round are higher than the current bid, the textual comment overrides the semantics of the protocol by indicating that the accept-proposal message should not occur (and presumably there would be o instances of reject-proposal rather than $o - 1$).
2. Uses variables (e.g. "hammer price" and "current bid"), but these are not defined, and it is not specified how these are updated.
3. Does not specify which agent should receive the final message ("request").
4. Does not model changes to the set of participants: if an agent drops out of the protocol then the value of m is not updated.
5. Does not clearly define the recipients of messages sent to participants. The protocol includes two messages: reject-proposal (sent to $o - 1$ participants) and accept-proposal (sent to a single participant). The AUML protocol does not indicate which agent receives which message, nor does it require that the agent receiving the accept-proposal message be the agent that did not get a reject-proposal message.

In comparison, our notation does not rely on English text, or implicitly defined variables (such as "hammer price"), and specifies that the final message (pay) is sent to the agent that put in the winning bid. In addition, observe that HAPN allows specifying dynamic constraints on roles, e.g., the winner role in the Auction protocol. HAPN also allows the recipients of a message to be a variable (e.g. Bs), and, as with any variable, it can be updated. An example of this is shown in Fig. 11, where an agent that does not bid is removed from the set of bidders ($t_0 \rightarrow t_2$), and therefore no longer participates in the auction.

5.3 Holonic manufacturing

The Holonic manufacturing example captures a possible set of protocols for managing the interactions between agents in the Cambridge Holonic Packing Cell [24], often used to demonstrate the kind of flexibility and co-ordination that can be achieved by autonomous self-coordinating agents (e.g. [23]).

The cell is made up of (see Fig. 12): (1) a rotating table T which has a jig on each side of it, for placing parts to be joined into composites, (2) a robot R1 which can load and unload parts, as well as move them to the flipper, at one side of the table, (3) a robot R2 which can join the bottom and next part, at the other side of the table, and (4) a flipper F (not shown) which can turn a composite part upside down if needed, prior to adding a new part to be joined.

The cell must flexibly interleave the making of two composite parts (one on each jig), without any additional centralised controller. The table has to be rotated to allow parts assembled at one side of the table to be joined by the robot at the other side. However, the table must not be moved while any of the robots (the loader, the joiner or the flipper) are doing something. This requires that at various times the table must be locked, so that some agent can complete its work.

The challenge posed by this case study is to be able to model the process in such a way that it allows for two composite parts to be simultaneously manufactured (using the two jigs) with locking being coordinated.

Additionally, we want the specification to be modular. For example, we want to be able to define a locking protocol that is generic and can be reused, rather than one that is specific to the scenario, and is mixed up with the rest of the protocol. In using this example system in

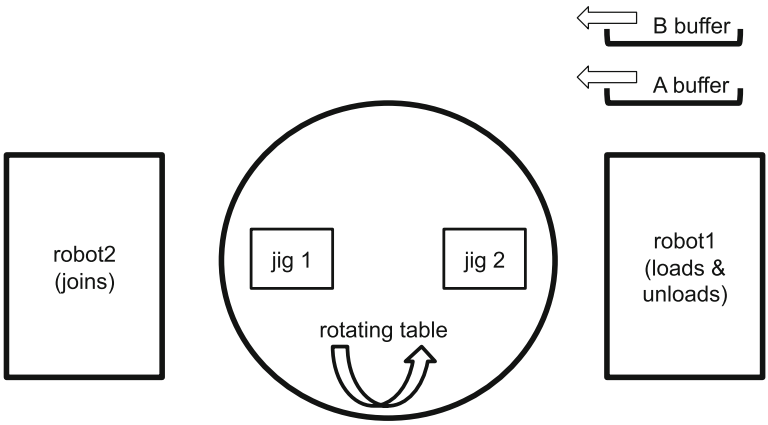


Fig. 12 Holonic manufacturing cell (flipper not shown)

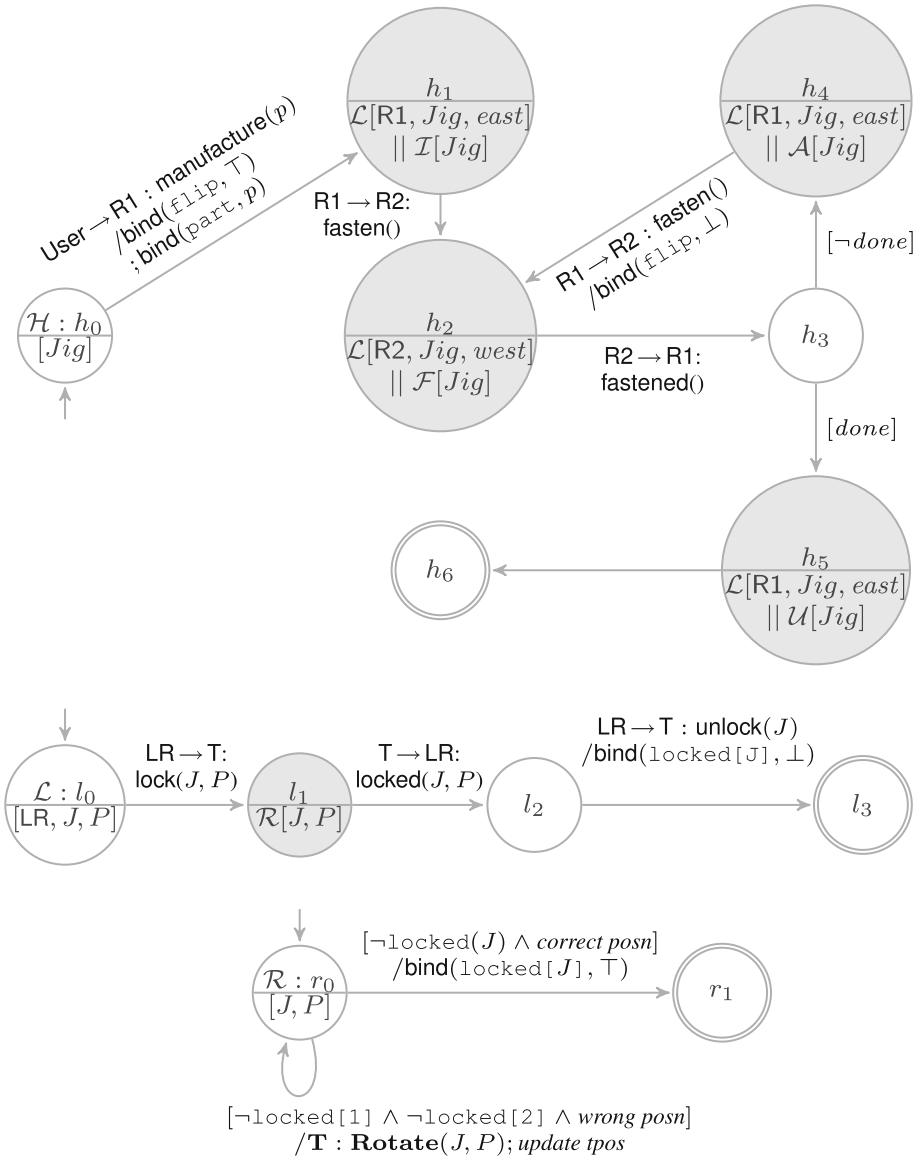
[46], Winikoff and Padgham define a (AUML) locking protocol to achieve the locking, but the unlocking is a separate single message outside of the lock sub-protocol. The reason for this (as discussed in Sect. 1) is that it is not possible in AUML to define a protocol to capture the sequence: "lock ; do something depending on context ; unlock" where the "do something depending on context" is specified outside the protocol containing lock and unlock. We show here how a general purpose locking protocol, containing the conceptually important "unlock", can be cleanly defined, then re-used in multiple places within the system. In addition to illustrating several aspects of modularity achievable in our hierarchical approach, this case study also illustrates binding of agents to roles, and the use of system and interface variables.

Figures 13 and 14 show the protocols. There is a top level protocol $\mathcal{H}$ that manages the overall process, a generic locking protocol $\mathcal{L}$ (along with a sub-protocol $\mathcal{R}$ to deal with rotating the table), and four protocols for the different stages of the manufacturing process: the initial loading of two parts $\mathcal{I}$, adding an additional part $\mathcal{A}$, fastening parts $\mathcal{F}$, and unloading the finished product $\mathcal{U}$. The addition protocol has two small sub-protocols $\mathcal{A}_1$ and $\mathcal{A}_2$.

The protocols make use of a few variables. There are two system variables that are shared across protocol instances. These capture the position of the table (`tpos`) and whether each of the two jigs has been locked (`locked[1]` and `locked[2]`). Initially `locked[j]` $= \perp$ for both values of `j` (i.e. both jigs are unlocked). The top-level protocol $\mathcal{H}$ also has two variables (both are accessible in all sub-protocols, i.e. the scope is protected/public): `flip` captures whether the composite part being worked on needs to be flipped, and `part` tracks the sequence of basic parts that still need to be added to the composite part.

All protocols have an interface variable that indicates which jig is to be used (J or Jig). The locking protocol also has two additional interface variables: the role that will be requesting the lock (Lock Requester LR, which is bound to either R1 or R2), and the desired position for the Jig (P).

We now describe each of the protocols. The top level protocol $\mathcal{H}$ has three states (h_2, h_3, h_4) that it cycles through as needed, in addition to the start state h_0, initial loading state h_1, unloading state h_5, and end state h_6. State h_4 has sub-protocols to add parts to the jig (with flipping if needed for ordering), h_2 has sub-protocols for joining parts, and h_3 is a state which either transitions to the state with unloading, or transitions back to h_2 to add a new part. States h_1, h_2, h_4 and h_5 all include the locking protocol as a sub-protocol, along with the other relevant sub-protocols (e.g. fasten, add part).



Variables: `tpos`, `locked[J]` (system); `flip`, `part` (owned by $\mathcal{H}$, protected/public)

Fig. 13 Holonic manufacturing protocols (part 1)

At the start of the process two parts must be loaded, prior to fastening, whereas at all other times a single part is loaded to add to the existing composite. We use two different states to manage this (h_1 and h_4). Alternatively, we could have merged the states, and the protocols $\mathcal{I}$ and $\mathcal{A}$, and instead used a variable to manage this. This is simply a design decision, and both are equally correct. The empty transition from h_5 to h_6 has the usual implicit guard that the sub-protocols ($\mathcal{L}$ and $\mathcal{U}$) are in final states. Some of the guards in $\mathcal{H}$ (e.g. $h_3 \rightarrow h_5$) refer to

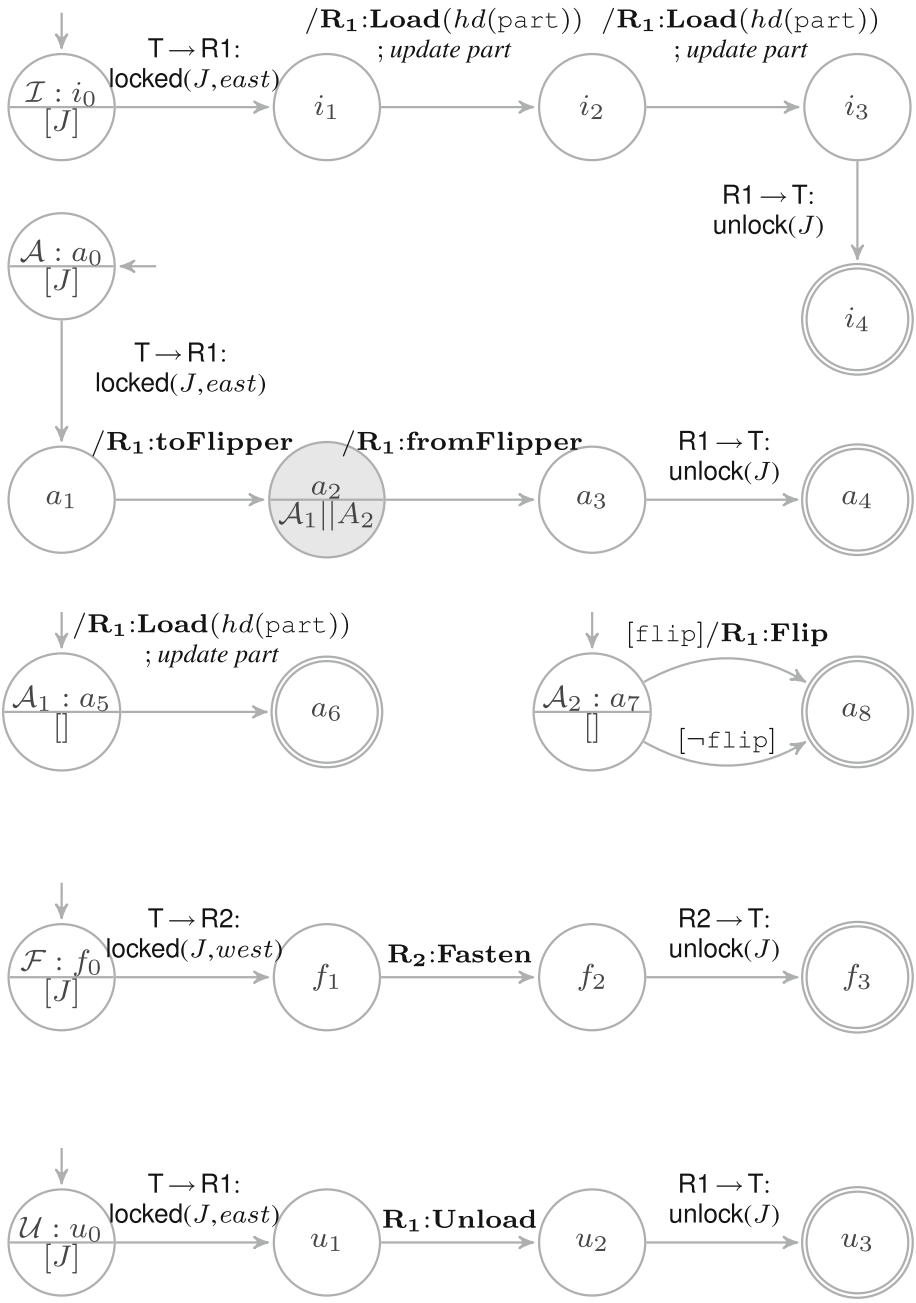


Fig. 14 Holonic manufacturing protocols (part 2)

a condition "*done*": this informal notation can be defined precisely as: $done \equiv \texttt{part} = \langle\rangle$, i.e. we are done exactly when the sequence of parts still to be added is empty.

The locking protocol $\mathcal{L}$ shows that locking involves: (1) a request from the lock requester LR to the table T to lock the jig J in a specified position P, followed by (2) a response granting

the lock, and then followed by (3) the lock requester releasing the lock. The protocol is simple and modular: it is used in a few places in the top-level protocol, sometimes by R1 to lock the jig in the East position, sometimes by R2 to lock the jig in the West position. The locking protocol is also generic in allowing the lock requester to be instantiated to any role.[9]

The interface variables for the locking protocol are bound to the jig, and the desired position. These are then used in the subsequent locked and unlock messages. This is an example of how some information about the content of the message is crucial for ensuring correctness of a protocol specification, or for tracking interactions during execution. For example, the sequence lock(J1,East) followed by locked(J2,East) should not allow the protocol to transition from l_1 to l_2 due to a mismatch in message parameters.

The locking protocol has a sub-protocol $\mathcal{R}$ which ensures (in the general case) that whatever is required before issuing the lock, is achieved. In our case the table must be rotated to the correct position based on the specific request message, which cannot be done until any other lock on the table has been released. If the table is already in the correct position, then the protocol can simply reserve jig J (by binding `locked[`J`]` to $\top$. However, if the table is not in the correct location, then it needs to be rotated, which can only be done when *both* jigs are unlocked.

The use of a different specification of $\mathcal{R}$ (e.g. moving an arm to a specified height), would allow re-use of the same basic lock protocol for a somewhat different situation. Because of their relative simplicity, these modular protocols are often quite straightforward to extend. For instance if we wanted to ensure that after some time period a lock was released, regardless of whether the requester gave it up, then we could simply add a transition based on a guard that captured the value of an elapsed time variable, with an effect of binding the relevant lock variable to $\bot$.

The protocols $\mathcal{L}$ and $\mathcal{R}$, as presented in Fig. 13, are not entirely precise: they resort to the informal notations "*correct posn*", "*wrong posn*" and "*update tpos*". For protocols intended solely for human consumption, this level of informality is typical and adequate. However, if we want the protocols to be interpretable by tools (e.g. checking which traces are valid), then we need to specify these notions precisely. This can be done by defining a representation for `tpos`. One possible representation is for `tpos` to be either $\{(Jig1, East), (Jig2, West)\}$ (i.e. jig 1 is in the East position and jig 2 is in the West position) or $\{(Jig1, West), (Jig2, East)\}$, in which case "*correct posn*" is $(J, P) \in$ `tpos`, and "*update tpos*" toggles between the two possible values of `tpos`.

The protocols $\mathcal{I}$, $\mathcal{A}$, $\mathcal{F}$, and $\mathcal{U}$ are all straightforward: they all begin by receiving a message from the table granting a lock on the relevant jig J in the required position (East for $\mathcal{I}$, $\mathcal{A}$, and $\mathcal{U}$, and West for $\mathcal{F}$), the protocols then carry out the manufacturing task (e.g. for $\mathcal{L}$ loading the two initial parts), and finish by unlocking the jig.

At any point in the manufacturing process the next part to be loaded is the first item in the list of parts (i.e. hd(`part`) where hd returns the head of the list), and when a part is loaded, the list of parts is updated. This is indicated informally in Fig. 14 by "*update part*", but could be formalised as bind(`part`, tl(`part`)) where tl returns the tail of the list.

The process of adding an extra part to an existing composite part (protocol $\mathcal{A}$), involves moving the composite part to the flipper, loading the new part, and then putting the composite part on top of the new part. This process is required because robot R2 is only able to join the bottom part onto the (possibly composite) part above it. State a_2 is where the new part is loaded, and, in parallel, the composite part is flipped (if required).

[9] Although the protocol does specify that the table T grants the lock, this can be easily generalised by replacing T with LG (Lock Granter) and adding LG to the protocol's interface variables.

We illustrate the functioning of the protocol by considering an example interaction that begins by instantiating $\mathcal{H}$ to use jig 1, and then begins with the user requesting that the system manufacture a composite part comprising $\langle a, b, c\rangle$. The interaction then proceeds as follows:

1. The manufacture message transitions from h_0 to h_1, and binds `flip` to $\top$ and `part` to the list of parts $\langle a, b, c\rangle$.
2. The sub-protocols of state h_1 are instantiated. Since the sub-protocols have final states, the transition from h_1 to h_2 cannot occur until both protocols are in their final state. Since the initial loading protocol ($\mathcal{I}$) must synchronise with the locking protocol, the only possible next transition is for the Lock Requester (in this case R1) to ask the Table to lock jig 1 in the east position. This message transitions the locking protocol to state l_1.
3. State l_1 has a sub-protocol which is instantiated. In this case we assume that jig 1 is already in the correct position, so we can transition from r_0 to r_1, resulting in `locked[1]` becoming $\top$.
4. Since protocol $\mathcal{R}$ is in its final state, the transition from l_1 is now enabled, which allows the locking and initial loading protocols to transition from l_1 to l_2 and from i_0 to i_1 (this is synchronised).
5. The initial loading protocol can then load the two parts (a first, so b is on top), transitioning from i_1 to i_2 and then to i_3, this also updates `part` to $\langle c\rangle$.
6. The initial loading and locking protocols can then do a synchronised transition to (respectively) i_4 and l_3, which corresponds to a request to unlock the table, and results in `locked[1]` becoming $\bot$.
7. Both sub-protocols of h_1 are now in a final state, so the top-level protocol can transition to h_2 on the occurrence of the message fasten.
8. State h_2 has two sub-protocols, locking (but with jig 1 in the West position), and fastening. As before, the lock requester (in this case R2) asks the table to lock jig 1 in the west position (message lock), which transitions $\mathcal{L}$ from l_0 to l_1.
9. Sub-protocol $\mathcal{R}$ is initialised. In this case the table is not in the correct position, and so it is rotated (remaining in state r_0) which updates `tpos`, and then the table is locked (transitioning to r_1 and updating `locked[1]`).
10. The message locked can now occur, which transitions to l_2 (in $\mathcal{L}$) and to f_1 (in $\mathcal{F}$).
11. The fastening protocol can then fasten the two parts a and b, to form a composite ab part (transitioning to f_2).
12. The table can now be unlocked (message unlock) which transitions $\mathcal{F}$ to f_3 and $\mathcal{L}$ to l_3, updating `locked[1]`.
13. Since $\mathcal{F}$ and $\mathcal{L}$ are both now in their final state, the top-level protocol can proceed to h_3 (message fastened).
14. Since we are not yet done (`part` $= \langle c\rangle$), we transition to h_4, to continue adding another part.
15. Similarly to h_1, the sub-protocols ($\mathcal{L}$ and, in this case, $\mathcal{A}$) are instantiated, and subsequently: (1) a lock is requested (lock); (2) the table is rotated in $\mathcal{R}$, since it is in the wrong position; (3) the table is locked (r_0 to r_1); and (4) the locked message is sent (l_1 to l_2 and a_0 to a_1).
16. At this point protocol $\mathcal{A}$ can move the ab composite part to the flipper, transitioning to a_2.
17. The two small sub-protocols ($\mathcal{A}_1$ and $\mathcal{A}_2$) are instantiated and run in parallel. This results in the next part ("c") being loaded into jig 1 (and `part` updated), and, since `flip` is $\top$, the ab part being flipped so b is at the bottom.

18. Once the loading and flipping have both been done, $\mathcal{A}$ can move the composite part back from the flipper, transitioning to a_3.
19. The table can now be unlocked (unlock), transitioning $\mathcal{L}$ to l_3 and $\mathcal{A}$ to a_4.
20. The top-level protocol can now transition back to h_2 (message: fasten, and `flip` becomes $\perp$) to fasten c to the composite ba part, which results in: (1) a lock being requested (lock); (2) the table is rotated in $\mathcal{R}$, since it is in the wrong position; (3) the table is locked (r_0 to r_1); (4) the locked message is sent (l_1 to l_2 and f_0 to f_1); (5) the parts are fastened (transition to f_2); and (vi) the table is unlocked (unlock, and transitioning to f_3 and l_3).
21. The top-level protocol can now transition to h_3 (message fastened).
22. Since we are now done (`part` $= \langle\rangle$), the protocol transitions to h_5. The process of locking and acting is similar to the other shaded states in $\mathcal{H}$: (1) a lock is requested (lock); (2) the table is rotated in $\mathcal{R}$, since it is in the wrong position; (3) the table is locked (r_0 to r_1); (4) the locked message is sent (l_1 to l_2 and u_0 to u_1); (5) the final composite part is unloaded; and (6) the table is unlocked (unlock).
23. Finally, the top-level protocol transitions to its final state (h_6) concluding the interaction.

This protocol is more complex than the previous two case studies. However, this complexity is part of the problem, and the HAPN protocol allows this complex scenario to be captured in a modular way. Each protocol is small, and can be understood. By contrast, because AUML cannot synchronise protocols, nor parameterise over parts of protocols, it cannot model a modular (and reusable) locking protocol.

6 Evaluations

In order to assess HAPN we performed two evaluations. The first (Sect. 6.1) is a feature evaluation where we identify important features, and compare HAPN to a number of prominent existing notations with respect to these features. This evaluation assesses features of the notation such as precision, simplicity, modularity and expressiveness. However, it does not directly assess the *usability* of the notation (except via proxy measures such as the notation's simplicity). We therefore also carried out a second evaluation to assess usability. This second evaluation (Sect. 6.2) is a human evaluation that directly assesses the notation's usability.

6.1 Feature evaluation

In order to evaluate our notation in comparison to other notations we identified a number of key features that we argue are important, and that can be assessed objectively. These features are: being precisely defined, having a graphical notation, being simple, supporting modularity, and allowing various specific types of interactions to be expressed effectively. We next describe each of these features. For each feature we argue why the feature is important, define the feature more precisely, and explain how we assess the feature. The motivation for the features stems from the need (as described in Sect. 1) to have a notation that is precise, usable by software engineers (i.e. pragmatic), and expressive. Expressiveness encompasses being able to capture various interaction scenarios (such as those described in the previous section), but also having support for modular decomposition of protocols, to handle larger and more complicated protocols.

The first feature is *precision*. It is important that the notation be precisely defined, since this allows tool support to be developed that provides the protocol designer with more than just

a graphical editor. Given the complexity and difficulty of developing protocols that correctly capture the desired behaviour, it is useful to have tool support for simulating possible traces through a protocol, and for checking that the protocol satisfies certain properties. This can only be done if the notation is precisely defined. We consider a notation to be precisely defined if its syntax and semantics are formally defined. We consider a notation to be partly precisely defined if there are multiple, differing, semantics for the notation. For example, as discussed later in this section, Statecharts have multiple semantics in the literature. One particular form of verification is being able to observe a running system to check whether the execution conforms to an interaction protocol. This clearly requires that the notation is formally defined. However, for all of the formally-defined notations that follow traditional principles, the ability to verify observed executions depends not on the notation as a whole, but on the semantics of messages. If a message is defined in terms of agents' mental states, then an observed message cannot be verified as being valid [34]. For example, when using typical mental state semantics for messages, an observed message where agent A informs agent B that p holds, is invalid if agent A does not believe p. However, an observer cannot check whether A believes p, and so cannot assess whether the message is valid. On the other hand, if we assume that a message is not a generic performative with semantics, but just a domain-specific message, then all the precise notations support checking an observed message sequence against the protocol. There are two specific cases to note. Firstly, Commitment Machines and BSPL commit to defining message semantics in a particular way that avoids agent mental states, and therefore they support verification of observed behaviour against protocols. Secondly, as discussed in Sect. 3, including non-observable state in a HAPN protocol means that verification of observed behaviour is with respect to a variant of the protocol. Avoiding non-observable state (as would be done in other notations), means that HAPN supports verification of observed behaviour against the protocol in the same way as other traditional notations.

Whether a notation is *graphical* is an important feature since it is generally considered [30] that graphical notations are more usable, especially by software engineers (as opposed to logicians or researchers). We consider a notation to be graphical if it uses two-dimensional layout of graphical elements to specify interaction protocols, and non-graphical if it uses one-dimensional text to specify protocols. Unsurprisingly, most of the notations that we consider are graphical. It is worth emphasising that while we consider this criterion important, we do not claim that non-graphical notations are necessarily less usable. We also note that it is possible to have a non-graphical notation for specifying interaction protocols that is augmented by a graphical visualisation of aspects of the protocol (e.g. [6]).

Simplicity here refers to the simplicity of the *notation*, not the simplicity of the protocols. The latter is related to expressiveness, and is considered under that feature. Having a simple notation is motivated by usability: all else being equal, a simple notation will be easier to learn. Additionally, a simpler notation may make it easier to provide tool support, although there are other factors that affect this. In order to assess the simplicity of the notation we follow the methodology of Moody and van Hillegersberg. They assess graphical notation complexity by counting "*the number of different graphical conventions that can appear on each diagram type*" [31, pp. 30–31]. Note that Moody and van Hillegersberg's methodology only considers graphical elements, ignoring textual elements. So, for instance, HAPN has a complexity of 5 because there are five distinct graphical elements: states, three possible state annotations (initial, final, hierarchical), and transitions. One consequence of focusing on assessing the complexity of the graphical notation is that notations that are non-graphical are not assessed ("N/A" for Commitment Machines and BSPL), so they cannot be compared (i.e. "N/A" should be interpreted as "incomparable", rather than as being good or bad). Another consequence is that only considering graphical elements might give a misleading

comparison, since one notation might have a simpler graphical notation than another by exploiting greater complexity in its textual elements. We discuss this below for each notation.

We have argued earlier for the importance of *modularity* in supporting the development, and maintenance, of larger and more complicated protocols. Without adequate support for modularity, protocols that are larger or more complex can become large, and hard to understand and maintain. We consider a notation to provide basic support for modularity if it allows a protocol to be broken into separate pieces. We also consider whether these pieces can be parameterised. Allowing parameters supports increased reuse of protocols, since they can be instantiated differently in different contexts. We assess for each notation whether it allows protocols to be parameterised by roles, and by variables ("Vars" and "Roles" in Table 1).

Finally, we consider the *expressiveness* of the notation. Expressiveness relates to the simplicity of protocols developed using the notation. A more expressive notation will, broadly speaking, allow given interaction scenarios to be expressed more simply and concisely. This can make the protocols easier to understand and maintain. Conversely, if a notation lacks support for certain interaction scenarios, this can make those sorts of interactions difficult, or even impossible, to express clearly.

In assessing the expressiveness of the different notations we have chosen four specific scenarios. These were selected as providing a diverse range of interaction types that occur in multi-agent systems, and that are therefore important to be able to cover. We do not claim that this set is complete. However, we do argue that these interaction types are typical and common in multi-agent systems, and that a notation's inability to represent these in an effective way is a significant shortcoming. Note that the selection of the case studies in the previous section was similarly motivated, and so the case studies are aligned with these four interaction types.

The four interaction types are: **P**arallelism and synchronisation, **E**xceptions, **I**nformation-driven interactions, and interactions involving multiple **R**ole **I**nstances. Note that although being able to represent sequence, choice and iteration in interactions is important, we did not include corresponding criteria in our evaluation, since we did not want to bias the evaluation in favour of traditional notations. We now briefly describe each one of these four interaction types, explaining what we mean, why we consider it important, and how we assess whether a notation can handle it adequately.

For **P**arallelism and synchronisation we assess a notation based on whether it is able to express that part of an interaction occurs in parallel with another part without having to specify all possible interleavings. The notation must also be able to capture synchronisation, i.e. that certain parallel interleavings should not be allowed. The need to model parallelism and synchronisation is very well motivated in the literature (e.g. [26]).

For **E**xceptions we assess the ease with which a notation can specify that a given part of an interaction can be aborted at any point (or, perhaps only at certain points). This needs to be able to be specified by adding a single construct to the protocol, rather than by adding many constructs. For example, in a Finite State Machine, it is possible to specify that an abort is possible during an interaction by adding a transition from every state to an abort state. However, this is not effective because the number of extra transitions grows with the size of the interaction part where the abort is allowed. On the other hand, in a Statechart one can indicate that an abort is possible from any sub-state of a given state by adding a single transition, regardless of the number of sub-states. Exceptions occur in a wide range of protocols. For example, in the play date scenario the interaction can be aborted at any point.

For **I**nformation-driven interactions we assess whether a notation can specify what information is to be collected, and allow the interaction to occur in any way that achieves this, without having to spell out all possible sequences. This type of interaction is one that we have

Table 1 Assessment of different approaches

	FSM	Petri Nets	AUML	Statecharts	CMs	BSPL	HAPN
Precise	✔	✔		(✔)	✔	✔	✔
Verify observed?	✔	✔		(✔)	✔	✔	✔
Graphical	✔	✔	✔	✔			✔
Simple	4	3	17	11	N/A	N/A	5
Modularity			✔	✔	✔	✔	✔
Variables			✔?	✔	✔	✔	✔
Roles			✔?		✔	✔	✔
P		✔	✔	✔	✔	✔	✔
E			✔	✔		✔	✔
I				✔	✔	✔	✔
RI			(✔)				(✔)

Key: "✔" = "yes", "(✔)" or "✔?" = "yes, but see discussion", blank = "no", "N/A" = "Not Applicable"; Simple = number of graphical elements, lower is simpler

observed multiple times, and is typical for protocols that involve interaction with a human (see Sect. 1). The play date scenario is an example.

Finally, there are important interaction protocols where a given role may have multiple instances ("**R**ole **I**nstances"). One common example of such a protocol is an auction. There are a number of features that are important to be able to represent in a protocol specification involving multiple role instances. Firstly, the notation needs to be able to capture that there are multiple instances of an agent that play a particular role (e.g. multiple bidders in an auction), and that certain parts of the interaction need to occur for each instance. Secondly, there are situations where there are constraints between these different instances. One example is synchronisation constraints, for instance in an auction, the auctioneer cannot start a new bidding round or end the auction until it has finished dealing with the bids that it has received. Another example of a constraint is the ability to track specific role instances. One such situation (in an auction) is that we track which bidder agent has the winning bid, and, when the auction ends, that agent is the one who wins the auction. We consider a notation to be able to model protocols with multiple role instances if it can model all of these cases. We do not claim that this is a complete set of cases for protocols with multiple role instances. However, since auctions and auction-like interactions are typical of multi-agent systems, a notation's inability to adequately model these cases is a shortcoming.

We compare our approach with four well-known and widely-used notations: Finite State Machines (FSMs), Petri nets [33], AUML [21], and Statecharts [19]. We also compare with two notations that have not been widely-used, but which represent recent work that aims to improve the flexibility of protocol notation representations: Commitment Machines (CMs) [45,49], and BSPL [35]. The results of the assessment are summarised in Table 1.

FSMs are simple, having four distinct graphical elements (normal state, final state, initial state indicator, and transition). FSMs do not make use of structured text in specifying a protocol, so there is no additional complexity from textual aspects of the notation, meaning that the assessment of graphical complexity is correct: the notation is indeed simple. FSMs are precisely defined and graphical, but lack expressiveness. For example, they cannot model parallelism, or the ability to abort an interaction, or information-driven interactions, or protocols involving multiple instances of a given role. FSMs do not provide any support for modularity.

Like FSMs, Petri nets [33] are precisely defined and graphical. They are simple, having only 3 distinct graphical elements (place, transition, arc). Like FSMs, they do not use structured text as part of the notation, so the notation is indeed as simple as the number of graphical elements suggests. They add the ability to model parallelism, but do not support the other expressivity cases. Petri nets do not provide modularity mechanisms, although extensions (hierarchical Petri nets) have been proposed [10,28].

Overall, both FSMs and Petri nets are "too simple": they are very simple (and hence amenable to being precisely defined), but lack expressivity.

AUML [21] is perhaps the most widely used notation for describing agent protocols. It is graphical (see e.g. Fig. 10), and for relatively simple interactions it is intuitive and easy to follow. The original AUML had issues with ambiguity, but most of these have been addressed with the advent of UML 2 and the new version of AUML [20,21]. However, perhaps due to the AUML work having been over-taken by UML 2.0 developments, it does not appear to have been formally defined, so is not precise. AUML supports concurrency and synchronisation, and the modelling of exceptions. Although it provides features to support the specification of protocols with multiple role instances (e.g. auctions), in fact we have found that precisely specifying such protocols in AUML is not easy, and the resulting protocols are often problematic. We saw earlier (Fig. 10 in Sect. 5.2) a typical AUML protocol for an English Auction, and noted a range of issues with the protocol. The auction protocol that we developed for our empirical evaluation (Fig. 20) was able to capture correctly the desired behaviour, but made use of non-standard AUML (a "par" box "for each participant"). We have therefore indicated AUML's support for multiple role instances (RI) in Table 1 as being "(✔)" (i.e. "yes, but …"). Turning now to information-driven interactions, AUML does not have a notion of variables. This means that while a play date protocol can be written that can be interpreted by a human as describing the correct behaviour, when considered closely, the protocol's informality and ambiguity create problems. Considering an AUML version of the play date protocol (Fig. 15, which is a copy of Fig. 17 in the "Appendix"), we note that, since variables are not precisely specified, neither is the behaviour of the protocol. For instance, does the protocol permit a request for one variable to be responded to with a different variable, e.g. request(day) followed by giveInfo(loc,home)? In order for the protocol to clearly specify its behaviour for these sorts of cases it needs to have some way of capturing the constraints between messages. The approach that we have followed relies on the use of variables in the notation, which AUML does not provide. We therefore do not consider AUML to support information-driven interactions, at least not without extensions, such as using artefacts that externalise some of the interaction constraints. Finally, considering modularity, AUML does provide facilities for modularity, but, as noted earlier, these are not precisely defined (hence the question marks in Table 1). Finally, we highlight that AUML is the most complex of the notations, with 17 distinct graphical elements, compared with 11 for Statecharts (as in [19]), and 5 or fewer for the other graphical notations. Like Statecharts and HAPN, AUML makes use of structured text to specify protocols. For example, guard conditions and loop bounds. However, not all of these are precisely defined, for instance guards are written in (unstructured) English. Overall, since AUML's graphical complexity is quite high even without considering textual elements of the notation, we conclude that the AUML notation is clearly complex. It could be argued that AUML's textual elements are simpler than Statecharts' textual elements, since there is no textual construct for effects, so we refrain from drawing conclusions regarding the relative complexity of AUML and Statecharts.

Statecharts, originally proposed back in the 80s [19] have been highly influential, and widely-used. They provide a graphical notation that is quite rich, and supports modelling

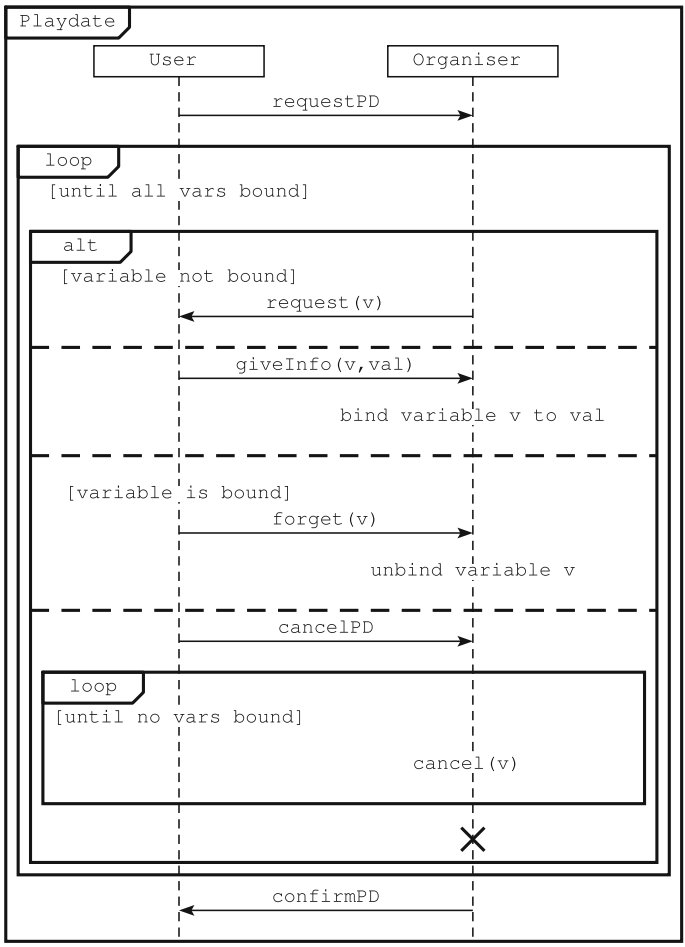


Fig. 15 The play date protocol in AUML

of parallel protocols, and aborting parts of interactions. This reflects their early history in modelling avionic systems. The existence of variables in the notation allows statecharts to model information-driven interactions. On the other hand, statecharts do not provide features for modelling interactions with multiple role instances: the Auction protocol in the appendix (Fig. 22) cannot capture precisely the management of roles (e.g. tracking which bidders are active, and who the winner is). The notation is rich, but quite complex, which has resulted in many different semantics being proposed, leading to problems. For example, Eshuis notes that "The existence of these different statechart formalisations can lead to a Babel-like confusion, because the same statechart can be interpreted completely differently under different semantics." [17, p. 66] and these differences are highlighted by Taleghani and Atlee [39]. We therefore consider statecharts to be precisely defined, but only in a weak sense, denoted "(✔)" in Table 1. Statecharts provide means of decomposing interactions into pieces, and include examples of sub-protocols that are parameterised by variables (e.g. Figure 40 of [19]). However, the notation is more focussed on describing events than interactions, and accordingly does not provide role-based abstraction. Indeed, transitions in a statechart

have *events* rather than *messages*, so the sender and receiver(s) of a message are not shown. The Statechart notation is clearly more complex than FSMs and Petri nets when we consider graphical elements. Additionally, Statecharts use structured text for guards and effects, which adds additional complexity, relative to FSMs and Petri nets.

Overall, both AUML and Statecharts are "too complex": they are fairly expressive (covering most desired cases), but they are complex and are either not precisely defined (AUML), or suffer from having multiple differing formal definitions (Statecharts).

Commitment Machines were described in Sect. 2. They are precisely defined. The CM notation is not graphical. However, there has been more recent work that has defined graphical notations to be used in designing commitment-based protocols. For example, the 2CL methodology [6] uses a range of arrows to indicate various constraints. However, these only capture one particular aspect of the design, namely the constraints. While the original descriptions of the CM framework did not deal with modularity, subsequent work, most prominently the Amoeba methodology [15], has extended the framework to allow for modularity, including the ability to define composite protocols that are formed by combining sub-protocols. Such composite protocols can include axioms that relate variables in one sub-protocol to those in another, and axioms that relate roles between sub-protocols. Since Commitment Machines define interaction sequences indirectly, this means that there is natural parallelism in the sense that, unless constrained, all possible sequences of messages are allowed. CMs also naturally model information-driven interactions. On the other hand, CMs do not provide good means for representing that an interaction can be aborted. For example, imagine a Commitment Machine version of the play date protocol. In order to prevent further requests or responses from being allowed after a cancel, we would need to add to each of these a condition that the interaction is not cancelled. Recall that we do not consider a notation to adequately support specifying exceptions if such extensive modification is required. Turning to interactions with multiple role instances, these are not supported by Commitment Machines. Finally, we note that although there has been some work on design [6,15,40,42,48] and implementation [44] aspects of interactions that are designed as CMs, this approach has not seen much adoption beyond the developers and their students. Similar comments also apply to BSPL and to HAPN.

The Blindingly Simple Protocol Language (BSPL) was described in Sect. 2. BSPL is precisely defined [37], and is textual. In defining desired properties of a BSPL protocol, Singh [37] uses a graphical notation ("causal structures") to visualise aspects of the enactment of a BSPL protocol. However, this is not a graphical version of BSPL, but a visualisation of an aspect of the enactment of a BSPL protocol. A strength of BSPL is parameterised sub-protocols, with protocol composition being one of the notation's two main constructs. BSPL supports parallelism and synchronisation. It does this not by providing a construct for parallelism, but by allowing any message to be sent unless there are constraints (e.g. missing information) that would prevent this. BSPL provides the ability to abort an interaction (by having a message or sub-protocol that binds the output variables of that part of the interaction). It naturally models information-driven interactions, but does not provide features for interactions with multiple role instances, such as auctions.[10] Finally, we note that we see BSPL, in its present form, as more of a core formal calculus than a usable notation. This view is based on our experience in trying to use BSPL, which has highlighted areas where we found the notation difficult to use, such as:

1. *Identifying* (*ordering and occurrence*) *constraints in a BSPL protocol* Since a BSPL protocol is just an unordered collection of elements, with ordering constraints expressed

[10] Since this paper was written, and possibly influenced by it, BSPL has been extended to handle auctions [11]. However, the extensions add substantial complexity to BSPL.

by the dependencies between parameters, it requires some work to determine what are the ordering constraints that are implied by the parameter adornments, and what messages are alternatives. That messages *a* and *b* cannot both occur in an interaction is represented by the two messages having a common "out" parameter. By capturing ordering and occurrence constraints indirectly, the reader has to work to identify them.

2. *Identifying the role of different parameters* The Bliss paper [38] highlights that message parameters are used for many things. By not distinguishing between these different purposes syntactically, BSPL protocols are harder to understand. It may not be clear to a reader, for instance, which parameters are there because they carry essential information for the interaction ("payload"), and which have been introduced only to ensure that certain messages cannot both occur in an interaction.
3. *Specifying loops* Expressing loops is surprisingly complex and involves subtle manipulation of multiple keys which we found difficult to understand.

Based on these considerations, which arise from our experience in attempting to use BSPL, we feel that BSPL requires some further work before it could really be considered pragmatic. As noted earlier, and as reinforced by the criteria-based comparison, BSPL is a promising and exciting development.

Overall, Commitment Machines and BSPL are simple and expressive, but, as argued above, there are concerns relating to their usability.

Finally, considering HAPN, it is obviously graphical, and is precisely defined (see Sect. 4). As shown in Sect. 5, HAPN is able to model all four cases considered: the Holonic manufacturing protocol demonstrates HAPN's support for Parallelism, synchronisation, and modularity, the play date example demonstrates HAPN's support for Information-driven interactions and Exceptions, and the Auction protocol demonstrates HAPN's ability to model a protocol with multiple role instances. However, is worth noting that the protocol does use an extension to the notation (having a sub-protocol for each instance), which is why we do not consider HAPN to fully support protocols with multiple role instances (hence "(✔)" in Table 1). HAPN's support for modularity allows protocols to be broken up and parameterised by variables and by roles. Considering the notation's simplicity, the graphical aspects of HAPN are comparable in complexity to FSMs and Petri nets, and are considerably simpler than AUML or Statecharts. However, unlike FSMs and Petri nets, HAPN does use structured text (guards, effects), which makes it more complex than FSMs and Petri nets. On the other hand, the textual aspects of HAPN are comparable to those of Statecharts, which means that the HAPN notation is indeed simpler than Statecharts, since its graphical aspects are simpler, and its textual elements are comparable. It is harder to compare HAPN to AUML, but we argue that the substantial additional graphical complexity of AUML (17 elements compared with 5 for HAPN), is more than enough to make AUML a more complex notation, despite HAPN having additional (textual) complexity, relative to AUML, in the form of structured effects. Finally, we turn to considering pragmatic usability. Since HAPN is new, it has not seen widespread use. Therefore, in order to argue that it is usable we turn to an empirical evaluation that we now describe.

6.2 Empirical evaluation

In order to assess how usable the HAPN notation is we conducted a human evaluation.[11] The aim of the evaluation was to assess the usability of the HAPN notation, compared against two well-known and widely-used notations: Agent UML (AUML) and Statecharts. We selected

[11] Ethics approval was obtained from the University of Otago (D15/224).

these two notations (from the longer list of notations in the previous section) since they are widely-used, and expressive.

6.2.1 Experimental design

The dependent variable that we wish to measure is the usability of the notation. However, usability is not simple, there are many aspects of usability that could be measured. In our evaluation we consider three aspects of usability of protocol design notations: how easy are protocols in the notation to *read and understand* (assessed by answering questions about the behaviour of the protocol, and by directly asking participants for their opinions on the notations' readability), and how easy is it to *create* protocols in the notation (assessed using a protocol creation task). The independent variable is the notation used.

We recruited 11 participants (students at the University of Otago) who had completed the second-year course INFO 211 ("Systems Analysis, Design, and Modelling"). This course covers systems analysis and design, including business process modelling and UML. Activity diagrams are covered in a single lecture, and, earlier in the course, there are two lectures on business process modelling and notations. The course does *not* cover agents, and none of the authors of this paper were involved in teaching it. Note that the course does not require that students have previously done a programming course, but in fact almost all participants had done a programming course. Participants were recruited by advertising (email to students who had completed the course), and were offered a $50 gift voucher as an incentive.

Each participant arranged a time to come in and complete the assessment tasks, which were done under exam conditions, working individually in a room with no other students. The evaluation, which was paper-based, comprised the following steps (completed in order):

1. Complete a pre-survey (see "Appendix A") which collected basic information on the student's background. Students were asked to indicate their major(s), minor(s), which courses they had passed, and their self-assessed level of familiarity with specific topics (e.g. software analysis and design, UML, Finite State Machines[12]).
2. For each of three protocols (Play date, Auction, Manufacturing): read a tutorial for the notation used, read the provided protocol, and then answer a collection of yes/no questions about the behaviour specified by the protocol. For each question participants were asked to indicate not just their answer, but also how confident they were in their answer (on a scale of 1 = not at all confident to 5 = very confident).
3. Participants were given a brief (< 1 page) description of a fourth scenario ("Voice-Driven Music System for Visually Impaired"), and were asked to design a protocol capturing the scenario using the same notation that they had used for the play date protocol.
4. Complete a post-survey that, for each notation, asked the participant to indicate whether they felt the notation was easy to read, easy to understand, and easy to write (all on a scale from 1 = very hard to 5 = very easy), and that also asked for free text comments on the notation (see "Appendix A" for specific questions). Finally, the participant was asked to rank the three notations from easiest to hardest to understand.

At various points along the way participants were asked to note the current time, so we could analyse how long different tasks took to perform. Participants were instructed to note when they took a break (indicating the start and end time of the break).

[12] Note that FSMs are not covered in INFO211, so we were expecting, and found, that few students indicated familiarity with them.

Table 2 Experimental design: assignment of notations to participants

Participant(s)	First notation (Play date)	Second notation (Auction)	Third notation (Manufacturing)
1, 7	AUML	HAPN	Statechart
2, 8	AUML	Statechart	HAPN
3, 9	HAPN	AUML	Statechart
4, 10	HAPN	Statechart	AUML
5, 11	Statchart	AUML	HAPN
6	Statechart	HAPN	AUML

Participants were provided with a printout that combined instructions, pre and post surveys, questions, and protocols. They were also provided with tutorials and quick reference sheets for each of the three notations.

In order to mitigate against differences in participant experience and skills we adopted a block design where we systematically varied the notations used and their order across the participants (see Table 2). For example, the first (and seventh) participant used AUML for their first protocol (Play date), HAPN for their second protocol (Auction), and the Statechart notation for their third protocol (Manufacturing). On the other hand, the second (and eighth) participant used AUML for their first protocol, but Statechart for their second (Auction) and HAPN for their third protocol (Manufacturing).

In developing the protocols (see "Appendix A") we took care to ensure that the protocols were equivalent in the described behaviour, were as simple as possible, and that, as much as possible, each protocol was idiomatic to its notation. For example, the three play date protocols follow a similar approach, modelling (part of) the state of the conversation through variables. This is somewhat atypical for AUML, where it would be more idiomatic to use message names instead of variables (e.g. "request-day", "give-day"). For the Auction protocol using the AUML and Statechart notations we used the parallel composition of multiple protocol instances (one for each auction participant). This is required to capture the correct behaviour, but actually is not strictly valid in either notation (e.g. AUML does not have a "par (for each [role instance of] participant)" construct). Finally, for the Manufacturing protocol[13] in AUML (Fig. 23) we could not define a modular locking protocol (for reasons discussed earlier), and so had to "unfold" the locking process.

In developing the questions to be answered we took care to explore the desired behaviour, including cases where there was some subtlety in the specified behaviour. For example, the interleaving of the bidding processes of two bidders in the Auction protocol.

We conducted two pilot trials of the evaluation. The first involved the authors of the paper completing the tasks separately. The aim was to check that the questions all had a single clear correct answer, and to check instructions for clarity. The second pilot involved a student carrying out the complete evaluation, in order to assess that the amount of time required was not excessive and that the instructions were clear. None of the pilot evaluations were included in the subsequent analysis. Finally, in order to assess the protocol creation task we developed a marking rubric (see "Appendix B").

[13] The manufacturing problem used for the evaluation is a simplified version of the one presented in Sect. 5.3: it only handles a single join, and the protocol does not model the flipper or rotating the table.

Table 3 Number of questions answered correctly by each participant, bottom row is average

Play date (30 questions)			Auction (35 questions)			Manufacturing (9 questions)		
AUML	HAPN	Statechart	AUML	HAPN	Statechart	AUML	HAPN	Statechart
20	17	14	26	24	28	2	6	5
16	22	15	22	27	27	4	3	5
27	16	18	26	27	29	6	3	7
14	19		24		26		4	4
19.25	**18.5**	**15.67**	**24.5**	**26**	**27.5**	**4**	**4**	**5.25**

Table 4 Time taken by each participant to answer questions (in minutes), bottom row is average

Play date			Auction			Manufacturing		
AUML	HAPN	Statechart	AUML	HAPN	Statechart	AUML	HAPN	Statechart
0:14	0:07	0:12	0:23	0:15	0:14	0:09	0:05	0:05
0:13	0:12	0:13	0:22	0:17	0:15	0:10	0:11	0:10
0:25	0:13	0:13	0:11	0:39	0:17	0:09	0:09	0:10
0:22	0:33		0:11		0:16		0:09	0:05
18.5	**16.25**	**12.67**	**16.75**	**23.67**	**15.5**	**9.33**	**8.5**	**7.5**

6.2.2 Analysis and results

We firstly consider the participants' performance on the tasks, specifically their ability to correctly answer questions about the behaviour specified by the protocol, and how long they took to do so.

Table 3 shows the number of correct answers for each participant, grouped by protocol, and then by the notation that the participant used for that protocol. The bottom row is the average. It appears that the participants' performance (ability to correctly answer questions about the behaviour described by the protocol) was not significantly affected by the notation used. This is confirmed by a statistical test[14] showing that none of the differences between performance for the different notations are statistically significant.

Table 4 shows the amount of time taken by each participant. Again, there does not seem to be a significant difference associated with which notation was used (the difference in the average for the Auction protocol was caused by a single outlier). Statistical testing shows no significant difference associated with different notations.

To summarise, participants' actual performance was not affected by the notation. This shows that, at least as far as ability to read and interpret protocols, the three notations lead to comparable performance. In other words, *HAPN is as easy to read and understand as AUML or Statecharts*, despite being more expressive, and having a higher level of precision and formality.

Interestingly, although participants' *performance* on a given protocol was not affected by the notation they used, the participant's *subjective assessment* of the notations did vary. Table 5 shows the participants' assessment of each notation's readability, understandability,

[14] Kruskal–Wallis, since the scores are not expected to be normally distributed, where, as usual, a difference is statistically significant when $p < 0.05$.

Table 5 Subjective evaluation of the notations by participants (1 = very hard, 5 = very easy), right columns are average (A) and median (M)

												A	M
Read													
AUML	4	4	2	1	4	2	4	1	4	3	5	3.09	4
HAPN	5	1	1	2	2	3	3	1	3	2	1	2.18	2
Statechart	4	4	2	5	1	2	5	4	3	4	2	3.27	4
Understand													
AUML	4	3	2	2	4	2	4	1	4	3	3	2.91	3
HAPN	4	1	1	2	1	3	3	1	3	2	1	2.00	2
Statechart	3	4	1	4	1	2	5	4	3	4	1	2.91	3
Write													
AUML	2	2	1	2	3	3	3	1	4	2	2	2.72	2
HAPN	1	1	5	2	3	1	1	2	1	N/A	N/A	1.89	1
Statechart	3	1	3	1	3	4	3	4	1	N/A	N/A	2.55	3

Table 6 Ranking of the notations by participants (1 = easiest, 3 = hardest), right columns are average (A) and median (M)

												A	M
AUML	2	2	1	3	1	2	2	3	2	2	1	1.91	2
HAPN	1	3	3	2	2	1	3	2	3	3	3	2.36	3
Statechart	3	1	2	1	3	3	1	1	1	1	2	1.72	1

and writability.[15] Looking at the numbers, it appears that HAPN was considered to be harder to read and understand. However, the differences are not statistically significant.

Considering the participants' overall ranking of the notations (Table 6), we observe that each notation had some participants who ranked it as easiest to understand, and each notation had participants who ranked it as least easy to understand. Again, it appears that HAPN was somewhat less preferred, but the difference is not statistically significant.

The written comments from the participants confirm that all of the notations had both clear and confusing aspects. For example, regarding AUML, comments included:

What aspects of the notation made it easy to read/write?
"The simple downwards flow of execution over time"
"clearly located sub sections and simple wording the 'boxes' are a helpful flow for simple processes"
What aspects of the notation made it hard to read/write?
"the vertical flow downwards, and the different 'types'"
"So much nesting, have to plan ahead to write complex diagrams, difficult to edit."
"- Loops were confusing - loops are just tags called 'loop'?, - AUML only appears to be easy for very simple programs"

[15] Two participants only provided an assessment of writability for the notation that they had used for the writing task.

Regarding HAPN, comments included:

What aspects of the notation made it easy to read/write?
"The familiar flow-chart style, the states of the machine being clear points, messages being separated"
"Simple flow, actions between states"
"Being similar to a programming language"
What aspects of the notation made it hard to read/write?
"the bound and bind commands confused me a bit"
"Child processes add much difficulty to the flow."
"notation on each → was confusing and made figuring out flow of execution confusing."

Finally, comments on Statecharts included:

What aspects of the notation made it easy to read/write?
"same reasons as HAPN"
"Arrows, and time on Horizontal axis make the flow obvious."
"very easy to follow, much like a flow diagram. Scope is very clear. Comments are descriptive."
What aspects of the notation made it hard to read/write?
"Transition confusing. Sub-tasks confusing. Hard to represent transitions."
"difficulty with notation and semantic information; lots of arrows, no immediate focus"
"The 'effect' notation being similar [sic] but not quite like programming (:=) and (r:act). Also hierarchie [sic] concurrency is hard to see and follow."
"To [sic] many different areas to consider in the diagram"

An interesting observation is that whereas the *notation* used did not significantly affect the subjective assessment, the *protocol* did. If we consider for each participant how they ranked the notation that they used for the Auction task, then we find that almost all participants ranked the notation they used as easiest to understand, regardless of which notation was used. Similarly, the majority of participants ranked the notation that they used for the play date protocol as being hardest to understand. In other words, regarding understandability, the protocol being considered mattered more than the notation that was used to present the protocol. The difference between the rankings, grouped by protocol, rather than by notation (Table 7) *is* statistically significant ($p = 0.001 < 0.05$, $\chi^2(2) = 13.752$, $df = 2$).

Finally, we turn to the creation task, for which participants were each asked to create a protocol for a given scenario (voice driven music system) using a specified notation. We began this paper by indicating that, in our experience, protocol design is one of the more problematic aspects of agent system design. It was therefore not surprising (especially considering that participants were part-way through their undergraduate studies) that this task was generally done very poorly. It is also worth noting that working on paper under exam conditions is

Table 7 Subjective ranking of notations grouped by protocol (1 = easiest to understand, 3 = hardest to understand), right columns are average (A) and median (M)

												A	M
Play date	2	2	3	2	3	3	2	3	3	3	2	2.54	3
Auction	1	1	1	1	1	1	3	1	2	1	1	1.27	1
Manufacturing	3	3	2	3	2	2	1	2	1	2	3	2.18	2

not an ideal setting for protocol design (although, unlike normal exams, participants did not have a time limit). One participant appears to have given up on this task (taking five minutes and submitting a design with only a single state that was crossed out). Of the remaining ten participants, only one participant (using HAPN) submitted a protocol that was a valid (HAPN) protocol. Two participants (using AUML) submitted protocols that were mostly valid AUML, but had a few oddities (e.g. using "pause" as both a message and a guard). None of the 10 protocols came close to covering all the features of the interaction. We scored each protocol on a scale of 0–24: there were 12 features of the interaction that had to be captured, and for each feature we awarded 1 if the protocol showed an attempt to model the feature, but did not do so correctly or completely, and 2 if the feature was correctly modelled. The scores given to the designs ranged from 5 to 16, with an average of 10.2. Interestingly, for each of the 12 features, there was at least one submitted protocol that captured it correctly (with two exceptions: checking for invalid navigation was attempted by a number of protocols but none got it right, and being able to specify artist and keyword in either order was only attempted by one protocol that didn't get it right[16]).

Overall we can draw the following conclusions from the creation task. As expected, protocol construction is hard. However, while hard, even though the participants were undergraduate students with no background in protocol design, a few did manage to complete (more-or-less) valid designs that captured some aspects of the desired interaction.

6.2.3 Validity

We now briefly consider potential challenges to validity.

Internal validity We mitigated against variation in participant background and ability by using a block experimental design. We mitigated against learning effects by using all possible orders (3 notations = 6 possible orders). It could be questioned whether the multiple choice questions that we used were good measurements of understanding. However, the questions were carefully constructed to assess understanding of the behaviour of protocols, especially around edge cases. Furthermore, the performance of students did vary on the questions, showing that the assessment instrument did have the ability to discriminate. Finally, the sample size (11 students) was small. However, the sample size was sufficient to show a statistically significant difference.

External validity We used multiple protocols, so we have some confidence that the findings would generalise to other protocols. A more significant challenge to generalisation is that second year students are not good proxies for practicing software designers. We might expect designers to perform better on these tasks (especially the protocol design task!). However, we do not have any reason to believe that the lack of difference between notations would change, if we were to repeat the experiment with more experienced and skilled designers.

7 Tool support

We have developed a prototype tool [47] that allows designing protocols using our notation. Development of a protocol from its conceptualisation to its design, generally, is an iterative process where one continuously refines the design by identifying issues and updating the protocol to match its requirements. Our main motivation behind this tool is to support the

[16] They had a message: "U → S: search(keyword‖Artist&&Keyword)" which is not valid HAPN.

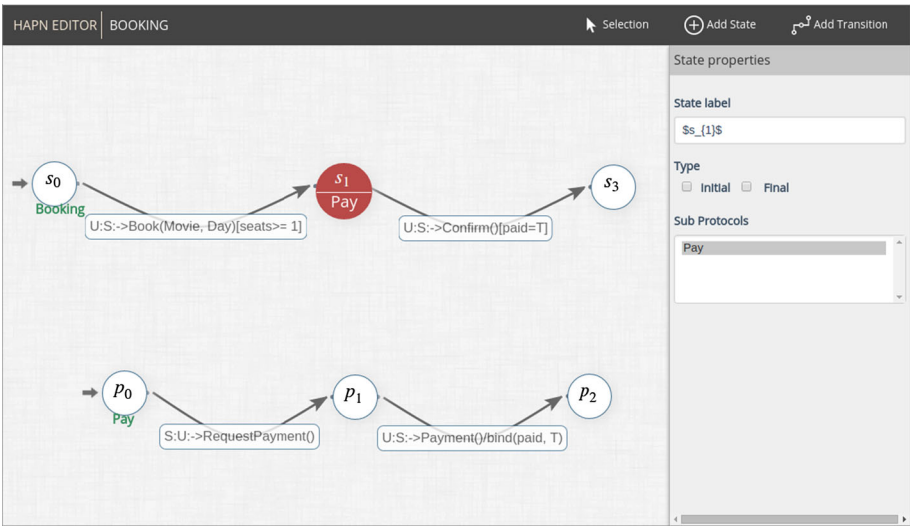


Fig. 16 A graphical editor for HAPN

overall design process. This tool allows protocols to be created and edited, and it is able to simulate protocol execution.

The simulation supports the user in viewing the behaviour of the protocol, and enables them to assess whether the specified protocol behaves as desired. This is an important feature that is made possible by the precision and formality of the HAPN notation.

Figure 16 shows a screenshot of the HAPN editor. In terms of the layout, it has two key areas: the left region that we call the *canvas*, where a designer will lay out various elements of our notation, namely, protocols, states, and transitions; and the right region to edit these elements using forms. The HAPN editor has three key operation modes: Selection, Add State, and Add Transition, which can be activated by selecting them from the top right area. The Add State and Add Transition modes allow adding states and transitions on the canvas, whereas the selection mode allows selecting existing states and transitions to modify their properties. A transition can be added between two states s_0 and s_1 by clicking state s_0, dragging the mouse while pressed, and releasing on state s_1. Releasing the mouse on the same state creates a looping transition on that state.

The tool provides a structured form to add and update properties of a state and of a transition. The form for a state allows updating its label as well as indicating whether a state is initial, and whether it is final. We allow (limited) usage of LaTeX math mode to render state and protocol names. For example, s_1 in the state name field is visible as s_1 in the canvas. In addition, one can also associate sub-protocols with a state. The names of sub-protocols associated with a state are displayed in the bottom half of that (parent) state. For example, in Figure 16 the Pay protocol has been associated as a sub-protocol of state s_1.

The form for an initial state also allows modifying the properties of its protocol (a protocol can be uniquely identified from its initial state). Properties of a protocol that are captured via the form include the protocol name and its interface variables. The name of the protocol is shown below its initial state. For example, in Fig. 16 the name Booking appears below the initial state s_0.

The property form for a transition allows updating the transition's components. The sender, receiver, and message label of a transition can be updated via free text fields. Multiple guards

and effects can be associated with a transition. For guards we allow comparing variables, checking whether a variable is bound, and whether a sub-protocol is in a particular state (but currently logical formulae other than conjunction are not supported). The effects of a transition may include binding/unbinding a variable or a domain effect. Both guards and effects also allow free text to be used.

The graphical editor automatically maintains a protocol specification (i.e. its states and transitions) based on how the user links and unlinks states with transitions.

In addition to serving as a tool for designing HAPN protocols, the editor allows a protocol designer to simulate a protocol's execution. Starting from the initial state of the top level protocol, the editor keeps a track of the current state and allows a user to *run* the protocol step by step by choosing the next transition at each step.[17] This type of simulated runtime execution provides a designer with important feedback such as the sequence of accepted messages.

The HAPN editor is built using HTML5, Javascript, and CSS and can be deployed both as a client/server and desktop based application. Integration with other desktop applications, such as model checkers and multi-agent software design tools, can be achieved by packaging the HAPN editor using the node-webkit platform.[18] In our context, using web based technologies provides two key advantages: (1) it allows building upon existing Javascript visualisation libraries, and (2) it enables us to share visualisation source code across web and desktop deployments.

8 Conclusion

We have presented a new notation for designing interactions in agent systems. The new notation, HAPN, has been carefully engineered to be able to effectively represent a range of interaction types, while remaining simple, precise, and pragmatic. We have presented the notation's conceptual framework and formal semantics. The presentation of three case studies provides evidence that HAPN can represent a range of interactions, which we have argued are problematic in existing notations. Our feature-based evaluation has highlighted that, unlike other traditional notations, HAPN manages to combine simplicity and expressiveness. Finally, our empirical evaluation has provided evidence that HAPN is usable and pragmatic. More precisely, we have shown that despite the higher precision and expressiveness, HAPN is as usable as AUML or Statecharts.

Overall, we regard HAPN as a good step forward, but there is more work to be done. Although we have argued that HAPN improves on existing traditional notations in a number of aspects (expressiveness, simplicity, precision), we have not managed to improve usability. HAPN is comparable to AUML and Statecharts in terms of usability, but this is not a great level of usability. More needs to be done to have a notation that is truly usable.

Some specific areas for future work include: (1) developing methodological guidance to support designers in developing HAPN protocols; (2) extending our understanding of how well the HAPN notation scales to larger examples; (3) defining formally what it means for two protocols to be identical, and characterising various relationships between protocols; (4) developing tool support for various forms of property checking, and to better support the

[17] Currently, the tool does not track values assigned to the variables and hence lacks the ability to evaluate guards.

[18] https://github.com/nwjs/nw.js/.

designer in following the methodology; and (5) exploring mapping protocols to implementation, and to other formalisms, such as trace expressions [2].

We note that providing tool support is an important part of the solution, since protocols can be difficult to reason about, and tools can assist designers in understanding the behaviour of their protocols, and in identifying issues in the design. Some further work concerns extending the tool support provided. This includes: extending the tool to track variable assignments during simulation, and to check guards; adding checks for the various conditions (specified in Sect. 4.6) that a well-formed protocol needs to satisfy; allowing set effects (including dealing with them in simulation); allowing guards with richer logical formulae; and, more significantly, providing analysis of protocols beyond step-by-step simulation.

We also plan to explore whether HAPN could use a slightly extended subset of the Statechart notation. This would have the advantage of using familiar graphical symbols, and would also potentially allow us to leverage one of the many existing tools for statechart editing. Finally, there is scope for additional evaluation, applying HAPN to further scenarios. Additional user evaluation, especially of protocol creation, and especially involving more experienced designers, would also be useful.

While we have not yet succeeded in the goal of improving usability, we believe that HAPN is a step in the right direction given its precise semantics and ability to capture the types of protocols we have experienced problems with in practice.

More broadly[19] we observe that the field of protocol engineering (including notations, analysis, and tools), might be advanced by developing a library of protocols. Such a library could include (informal) scenarios, protocol specifications, and implementations. Scenarios are informal descriptions of an interaction, whereas protocols are formal (or semi-formal) specifications in a given notation (e.g. AUML, BSPL, Petri nets). This library could be used to assess how well newly developed, or modified, notations are able to adequately model a range of interaction scenarios. Additionally, having such a library would allow the clarity and precision of the protocols to be evaluated by having independent parties implement them. In some cases, the different agents participating in a given interaction might be independently developed. Finally, having multiple protocol specifications and implementations for a given scenario would allow a range of comparisons of notations to be conducted (e.g. usability, complexity).

Acknowledgements This work is partially supported by the Australian Research Council and Real Thing Entertainment Pty. Ltd. under Linkage Grant No. LP110100050. We would like to thank Tim Miller for discussions relating to the empirical evaluation described in Sect. 6.2. We would also like to thank the anonymous reviewers for detailed and insightful comments that helped to improve this paper.

Appendix A: Evaluation materials: usability of notations for designing interactions in software systems

Firstly, thank you for participating!

Please note that this is not an examination of your individual ability, but instead is a study to assess how well people understand the three notations. As such, we would like participants to attempt to answer correctly and honestly, but do not feel pressured if you feel that you cannot answer some questions.

[19] We are indebted to an anonymous reviewer for this suggestion.

There are four parts to this test:

1. A brief pre-survey
2. Brief comprehension questions about **three** different interaction protocols
3. Creating a new interaction protocol
4. A brief post-survey

Important: Please complete these parts in the order that they appear in this document.

You will also be asked to write down the time at a number of places in the test. This is so we can see how long different questions take to answer. So, if possible, please complete the tasks without taking any breaks. If you do need to take a break, please note this by writing "break from *(start time)* to *(end time)*" at the place where you take the break (e.g. "took break from 10:13 to 10:47 a.m").

The time now is: _____ ***(please fill in the current time)***

Pre-survey

What major(s) are you doing (tick all that apply):

❒ Information Science ❒ Computer Science ❒ Software Engineering
❒ Other: _____

What minor(s) are you doing (tick all that apply):

❒ Information Science ❒ Computer Science ❒ Software Engineering
❒ Other: _____

Which papers[20] (i.e. courses/subjects) have you **passed**:

❒ COMP150 ❒ COMP160
❒ INFO211 ❒ INFO214 ❒ INFO221 ❒ INFO213
❒ COSC241 ❒ COSC242 ❒ COSC243 ❒ COSC244
❒ SENG301 ❒ INFO312 ❒ INFO323 ❒ INFO324

For the following topics please rate your familiarity on a scale from 1 (very familiar) to 5 (not familiar at all):

Software Analysis and Design	❒ 1 ❒ 2 ❒ 3 ❒ 4 ❒ 5
UML State Diagrams	❒ 1 ❒ 2 ❒ 3 ❒ 4 ❒ 5
UML Sequence Diagrams (aka interaction diagrams)	❒ 1 ❒ 2 ❒ 3 ❒ 4 ❒ 5
Finite State Machines	❒ 1 ❒ 2 ❒ 3 ❒ 4 ❒ 5

Comprehension of protocols

The time now is: _____ ***(please fill in the current time)***

This section presents three interaction protocols, each in a different notation. For each protocol you will be asked a sequence of questions to test how easy the protocol is to understand.

Playdate

The following interaction protocol for the Playdate scenario is in the ⟨NOTATION⟩ (e.g. **Agent UML** (**AUML**) notation). Please read the brief tutorial introduction to the notation now.

[20] For details on the papers see (e.g.) http://www.otago.ac.nz/courses/papers/index.html?papercode=COMP150.

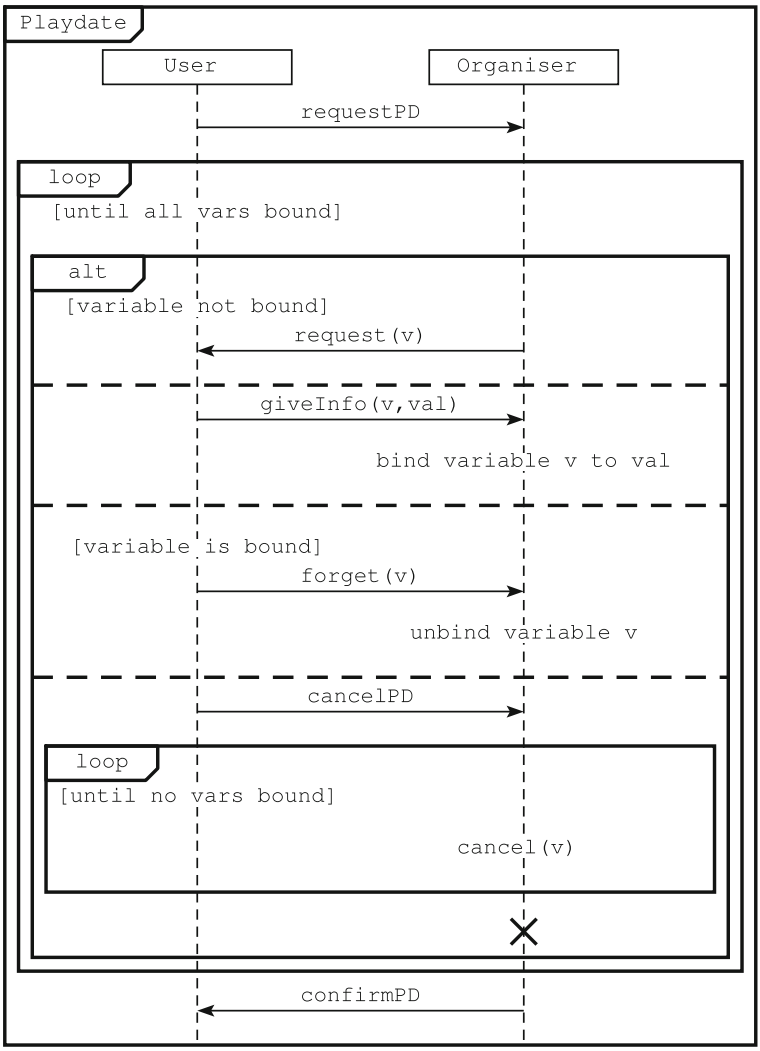


Fig. 17 The Playdate protocol in AUML

Once you have read the notation tutorial, please proceed.

The time now is: _____ ***(please fill in the current time)***

The protocol below captures an interaction where the system ("S") is organising a play date for a user ("U"). In order to confirm the playdate the system needs to know who the playdate is with ("person"), where it will be held ("location"), and when ("day").
Please take a few minutes to read the protocol in Fig. 17. Note that v can be any one of *day*, *person*, or *location*. (*Note*: *the other versions of the Playdate protocol, in the other two notations, can be found in Figs.* 18 *and* 19.)
Once you have finished reading the protocol in Fig. 17, please proceed.

The time now is: _____ ***(please fill in the current time)***

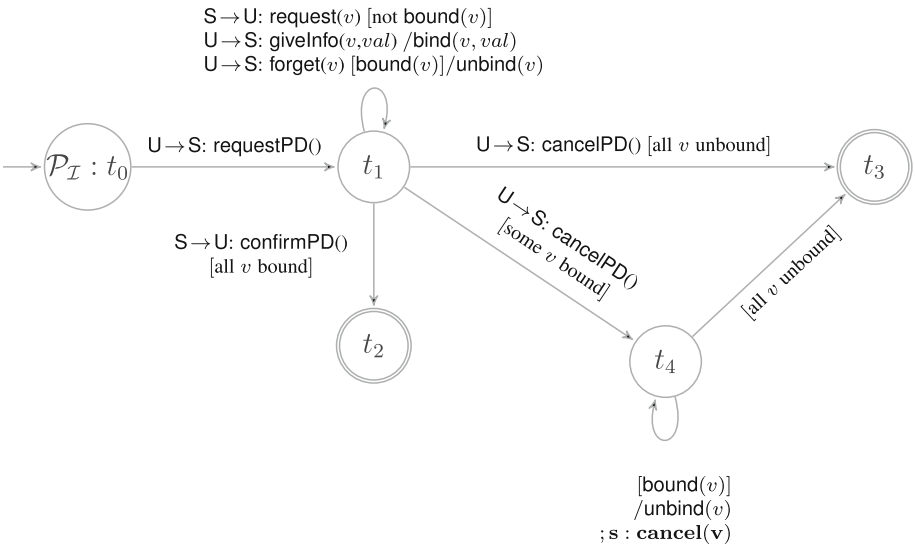


Fig. 18 The Playdate protocol in HAPN

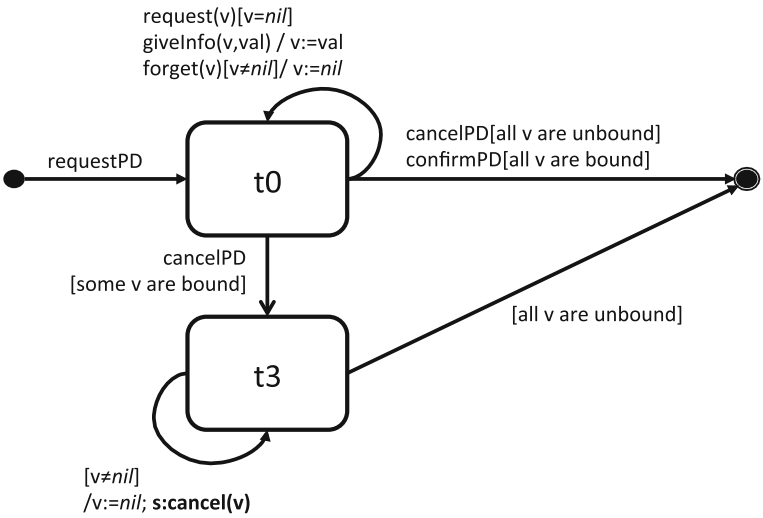


Fig. 19 The Playdate protocol in the Statechart notation

The following questions each give a sequence of messages. In some cases ending with "…" to indicate that there may be other messages. For each sequence please indicate whether that sequence is allowable according to the protocol ("ok") or whether, according to the protocol, the sequence of messages cannot occur ("no"). A sequence is *allowable* according to a protocol if that sequence can occur, and if it is a *complete* interaction (i.e. the interaction can validly stop at that point). If the sequence of messages ends with "…" then there may be further messages, and so to be acceptable it only needs to be valid, i.e. it does not need to be possible for the interaction to end at that point.

For each question you will also be asked to indicate how *confident* you are in your answer (1 = not at all confident, to 5 = very confident).

1. requestPD ; request(day) ; giveInfo(day,Monday) ; giveInfo(person,Pat) ; …
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
2. requestPD ; request(day) ; giveInfo(person,Pat) ; request(day) …
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
3. requestPD ; request(day) ; giveInfo(person,Pat) ; request(person) ; …
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
4. requestPD ; request(day) ; giveInfo(person,Pat) ; request(location) ; forget(person) ; request(person) ; …
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
5. requestPD ; request(day) ; forget(day) ; request(person) ; giveInfo(person,Pat) ; …
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
6. requestPD ; request(day) ; giveInfo(day,Monday) ; giveInfo(day,Tuesday) ; …
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
7. …giveInfo(day,Monday) ; cancelPD.
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
8. requestPD ; giveInfo(day,Monday) ; cancelPD ; cancel(day).
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
9. requestPD ; request(day) ; cancelPD
   ❒ ok ❒ no
   How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
10. requestPD ; request(day) ; giveInfo(person,Pat) ; cancelPD ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
11. requestPD ; request(day) ; giveInfo(person,Pat) ; request(day) ; giveInfo(day,Monday) ; request(location) ; giveInfo(location,Home) ; confirmPD ; cancelPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

12. requestPD ; request(day) ; giveInfo(person,Pat) ; request(day) ; giveInfo(day,Monday) ; request(location) ; giveInfo(location,Home) ; cancelPD ; confirmPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
13. requestPD ; request(day) ; giveInfo(person,Pat) ; request(day) ; giveInfo(day,Monday) ; confirmPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
14. requestPD ; request(day) ; giveInfo(person,Pat) ; request(day) ; giveInfo(day,Monday) ; giveInfo(location,Home) ; confirmPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

The time now is: _____ (please fill in the current time)

Suppose that the following sequence of messages has been sent as part of an interaction:

requestPD ; request(day) ; giveInfo(person,Pat) ; request(day) ; giveInfo(day,Monday)

Which one or more of the following messages are allowed to be sent next according to the protocol in Fig. 17? For each prospective next message, please indicate whether it can occur, and indicate how confident you are in your answer (1 = not at all confident, 5 = very confident).

15. request(day)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
16. request(person)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
17. request(location)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
18. requestPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
19. giveInfo(day, Monday)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
20. giveInfo(day, Tuesday)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

21. giveInfo(person, Pat)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
22. giveInfo(location, Home)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
23. forget(location)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
24. forget(person)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
25. forget(day)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
26. cancelPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
27. cancel(day)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
28. cancel(person)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
29. cancel(location)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
30. confirmPD
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

The time now is: _____ ***(please fill in the current time)***

Auction

The following interaction protocol for the Auction scenario is in the **Hierarchical Agent Protocol Notation** (**HAPN**). Please read the brief tutorial introduction to the notation now.

Once you have read the notation tutorial, please proceed.

The time now is: _____ ***(please fill in the current time)***

The protocol below captures an interaction where an Item is being auctioned by an Initiator, with a number of Participants taking part in the auction. An auction begins with the Initiator announcing the auction to all Participants ("start-auction(Item,Time)"). The auction then consists of one or more rounds. Each round begins with an announcement calling for bids ("call-for-bids(Item,CurrentPrice)"). During a bidding round a participant may bid ("bid(Item,Price)"), and each bid is either accepted or rejected by the Initiator ("accept-bid(Item)" or "reject-bid(Item)"). A Participant can also decline to bid ("no-bid(Item)") in which case they cannot bid in the future. Once the allowed time is up, the Initiator announces the end of the auction ("end-auction(Item)") and, if there is a winner, informs them ("inform-winner(Item,Price)").
Please take a few minutes to read the protocol in Fig. 20. *(Note: the other versions of the Auction protocol, in the other two notations, can be found in Figs.* 21 *and* 22*.)*

Once you have finished reading the protocol in Fig. 20, please proceed.

The time now is: _____ ***(please fill in the current time)***

The following questions each give a sequence of messages. In some cases ending with "…" to indicate that there may be other messages. For each sequence please indicate whether that sequence is allowable according to the protocol ("ok") or whether, according to the protocol, the sequence of messages cannot occur ("no"). A sequence is *allowable* according to a protocol if that sequence can occur, and if it is a *complete* interaction (i.e. the interaction can validly stop at that point). If the sequence of messages ends with "…" then there may be further messages, and so to be acceptable it only needs to be valid, i.e. it does not need to be possible for the interaction to end at that point.

For each question you will also be asked to indicate how *confident* you are in your answer (1 = not at all confident, to 5 = very confident).

The notation "P1:bid(ItemA,20)" indicates that Participant number 1 bids $20 for ItemA, and "P1:accept-bid(ItemA)" indicates that the Initiator accepted the bid on ItemA by Participant 1. In addition to using "P1" and "P2", the bids or responses relating to Participant 1 are in *red italic* and those relating to Participant 2 are in **blue bold**.

You should assume that the examples below are auctions with two Participants (P1 and P2).

31. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$0) ; *P1:bid(ItemA,$10)* ; **P2:bid(ItemA,$8)** ; *P1:accept-bid(ItemA)* ; **P2:reject-bid(ItemA)** ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
32. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$0) ; *P1:bid(ItemA,$10)* ; **P2:bid(ItemA,$12)** ; **P2:accept-bid(ItemA)** ; *P1:reject-bid(ItemA)* ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

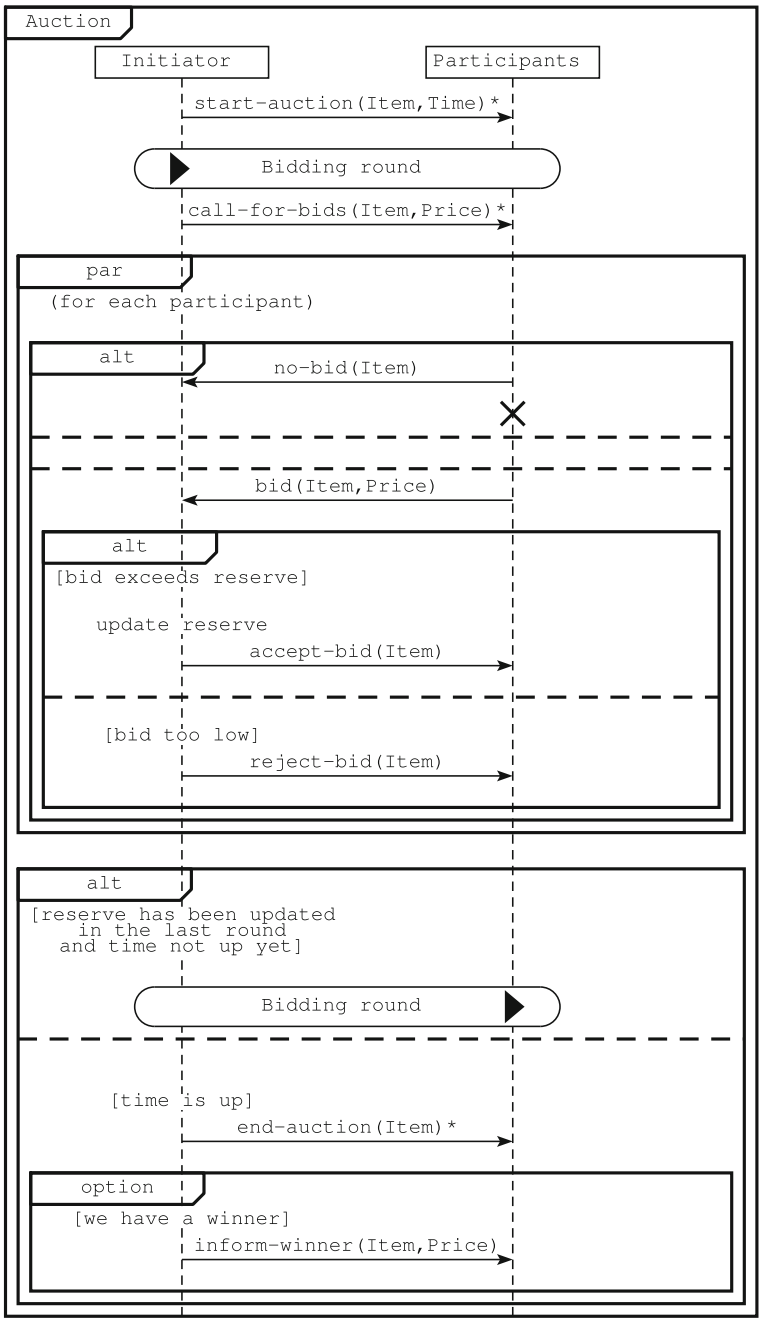


Fig. 20 The Auction protocol in AUML (*asterisk* highlights a message to multiple recipients)

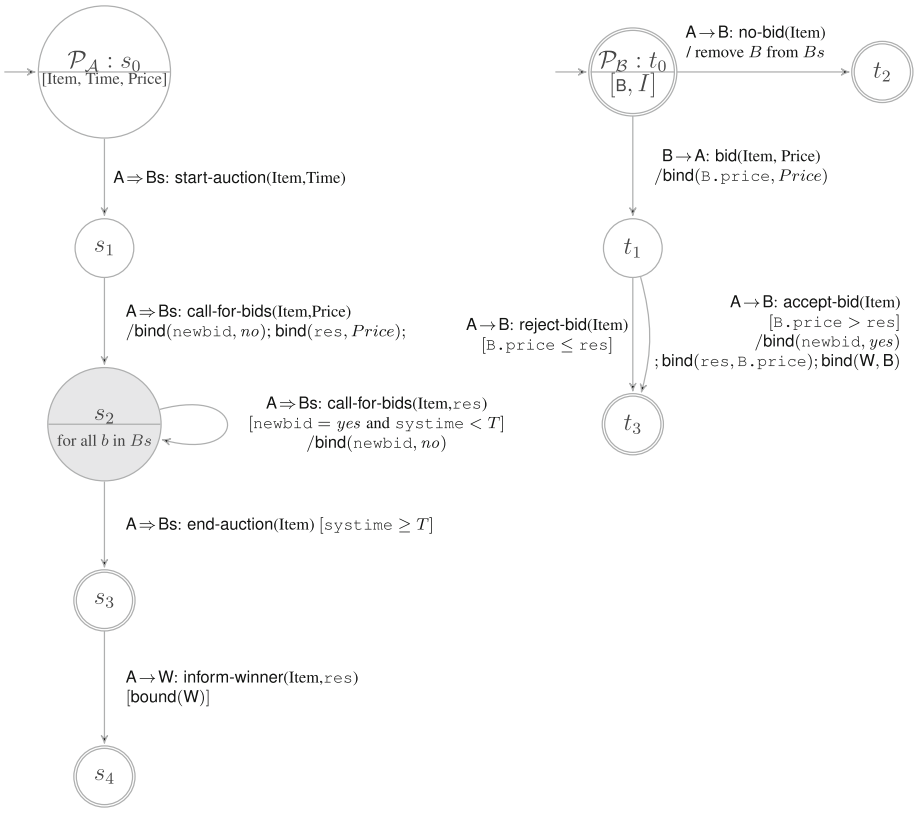


Fig. 21 The Auction protocol in HAPN. Note that the protocol on the *right* ($\mathcal{P}_B$) is a sub-protocol of state s_2 (*shaded*) in the protocol on the *left*, with one instance for each active bidder agent b in Bs

33. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$11) ; *P1:bid(ItemA,$10)* ; **P2:bid(ItemA,$12)** ; *P1:reject-bid(ItemA)* ; **P2:accept-bid(ItemA)** ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
34. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$11) ; *P1:bid(ItemA,$10)* ; *P1:reject-bid(ItemA)* ; **P2:bid(ItemA,$12)** ; **P2:accept-bid(ItemA)** ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
35. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; **P2:bid(ItemA,$4)** ; *P1:accept-bid(ItemA)* ; call-for-bids(ItemA,$11) ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

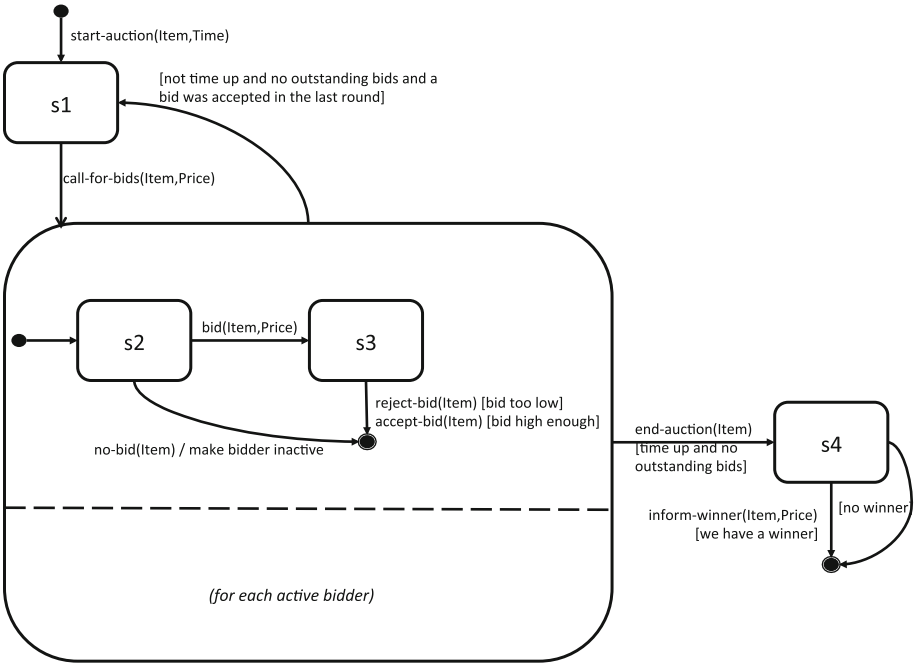


Fig. 22 The Auction protocol in the Statechart notation

36. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; **P2:bid(ItemA,$4)** ; *P1:accept-bid(ItemA)* ; **P2:reject-bid(ItemA)** ; call-for-bids(ItemA,$11) ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
37. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; **P2:no-bid(ItemA)** ; *P1:accept-bid(ItemA)* ; **P2:bid(ItemA,$12)** ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
38. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; **P2:no-bid(ItemA)** ; *P1:accept-bid(ItemA)* ; call-for-bids(ItemA,$11) ; **P2:bid(ItemA,$12)** ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
39. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; call-for-bids(ItemA,$11) ; **P2:bid(ItemA,$12)** ; …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

40. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; end-auction(ItemA) ; inform-winner(ItemA,$10).
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
41. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$10) ; *P1:bid(ItemA,$5)* ; *P1:reject-bid(ItemA)* ; call-for-bids(ItemA,$10) ; …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
42. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$10) ; call-for-bids(ItemA,$10) ; …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
43. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$10) ; end-auction(ItemA).
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
44. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; call-for-bids(ItemA,$11) ; end-auction(ItemA) ; inform-winner(ItemA,$10).
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
45. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; call-for-bids(ItemA,$11) ; **P2:bid(ItemA,$10)** ; **P2:reject-bid(Item)** ; call-for-bids(ItemA,$11) ; …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
46. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; **P2:bid(ItemA,$11)** ; **P2:accept-bid(Item)** ; call-for-bids(ItemA,$12) ; …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
47. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; *P1:bid(ItemA,$11)* ; *P1:accept-bid(Item)* ; call-for-bids(ItemA,$12) ; …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
48. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$4)* ; *P1:reject-bid(ItemA)* ; *P1:bid(ItemA,$11)* ; *P1:accept-bid(Item)* ; call-for-bids(ItemA,$12) ; …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

49. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$10)* ; *P1:accept-bid(ItemA)* ; call-for-bids(ItemA,$11) ; end-auction(ItemA).
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
50. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$4)* ; *P1:reject-bid(ItemA)* ; end-auction(ItemA).
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
51. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$8)* ; *P1:accept-bid(ItemA)*.
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
52. start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ; *P1:bid(ItemA,$8)* ; *P1:accept-bid(ItemA)* ; inform-winner(ItemA,$8).
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

The time now is: _____ ***(please fill in the current time)***

Suppose that the following sequence of messages has been sent as part of an interaction:
start-auction(ItemA,100seconds) ; call-for-bids(ItemA,$5) ;
P1:bid(ItemA,$ 10) ;P1:accept-bid(ItemA)
Which one or more of the following messages are allowed to be sent next according to the protocol in Fig. 17? For each prospective next message, please indicate whether it can occur, and indicate how confident you are in your answer (1 = not at all confident, 5 = very confident).

53. start-auction(…)
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
54. call-for-bids(itemA,$11)
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
55. P1:bid(ItemA,$11)
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
56. P2:bid(ItemA, $6)
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
57. P2:bid(ItemA, $11)
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

58. P1:no-bid(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
59. P2:no-bid(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
60. P1:accept-bid(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
61. P1:reject-bid(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
62. P2:accept-bid(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
63. P2:reject-bid(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
64. end-auction(ItemA)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
65. inform-winner(ItemA, $10)
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

The time now is: _____ ***(please fill in the current time)***

Manufacturing

The following interaction protocol for the Manufacturing scenario is in the **Statechart** notation. Please read the brief tutorial introduction to the notation now.

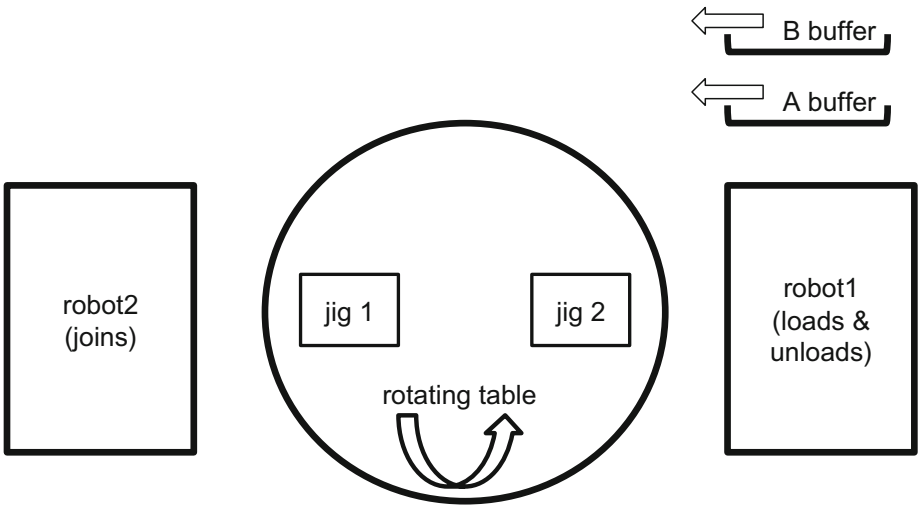
Once you have read the notation tutorial, please proceed.

The time now is: _____ ***(please fill in the current time)***

The protocol below captures a manufacturing scenario where raw parts are assembled into composite parts. The version we use is considerably simplified from a real scenario.

There is a table which has two jigs, each able to hold securely parts that are to be joined (see below). The table is able to rotate (but this is not captured in the simplified design). On one side of the table is a joiner robot, that is able to perform a JoinParts(Jig) action to join two parts

together. On the other side of the table is a Loader robot that is able to load individual parts onto the table, and unload the finished joined part from the table (actions: LoadPart(Part) and UnloadPart(jig)). The system should support simultaneous manufacturing of two composite parts, but there is an important constraint that the relevant jig must be locked while loading, or unloading, or joining is taking place. Note that the table cannot rotate while either jig is locked, in other words locking either jig also implicitly locks the table.



Please take a few minutes to read the protocol in Figure 23. *(Note: the other versions of the Manufacturing protocol, in the other two notations, can be found in Figs. 24 and 25.)*

Once you have finished reading the protocol in Fig. 23, please proceed.

The time now is: _____ ***(please fill in the current time)***

The following questions each give a sequence of messages. In some cases ending with "…" to indicate that there may be other messages. For each sequence please indicate whether that sequence is allowable according to the protocol ("ok") or whether, according to the protocol, the sequence of messages cannot occur ("no"). A sequence is *allowable* according to a protocol if that sequence can occur, and if it is a *complete* interaction (i.e. the interaction can validly stop at that point). If the sequence of messages ends with "…" then there may be further messages, and so to be acceptable it only needs to be valid, i.e. it does not need to be possible for the interaction to end at that point.

For each question you will also be asked to indicate how *confident* you are in your answer (1 = not at all confident, to 5 = very confident).

66. make(p1,p2), locked(jig1), LoadPart(p1), LoadPart(p2), …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
67. make(p1,p2), LoadPart(p1), lock(jig1), locked(jig1), LoadPart(p2), …
    ❒ ok ❒ no
    How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

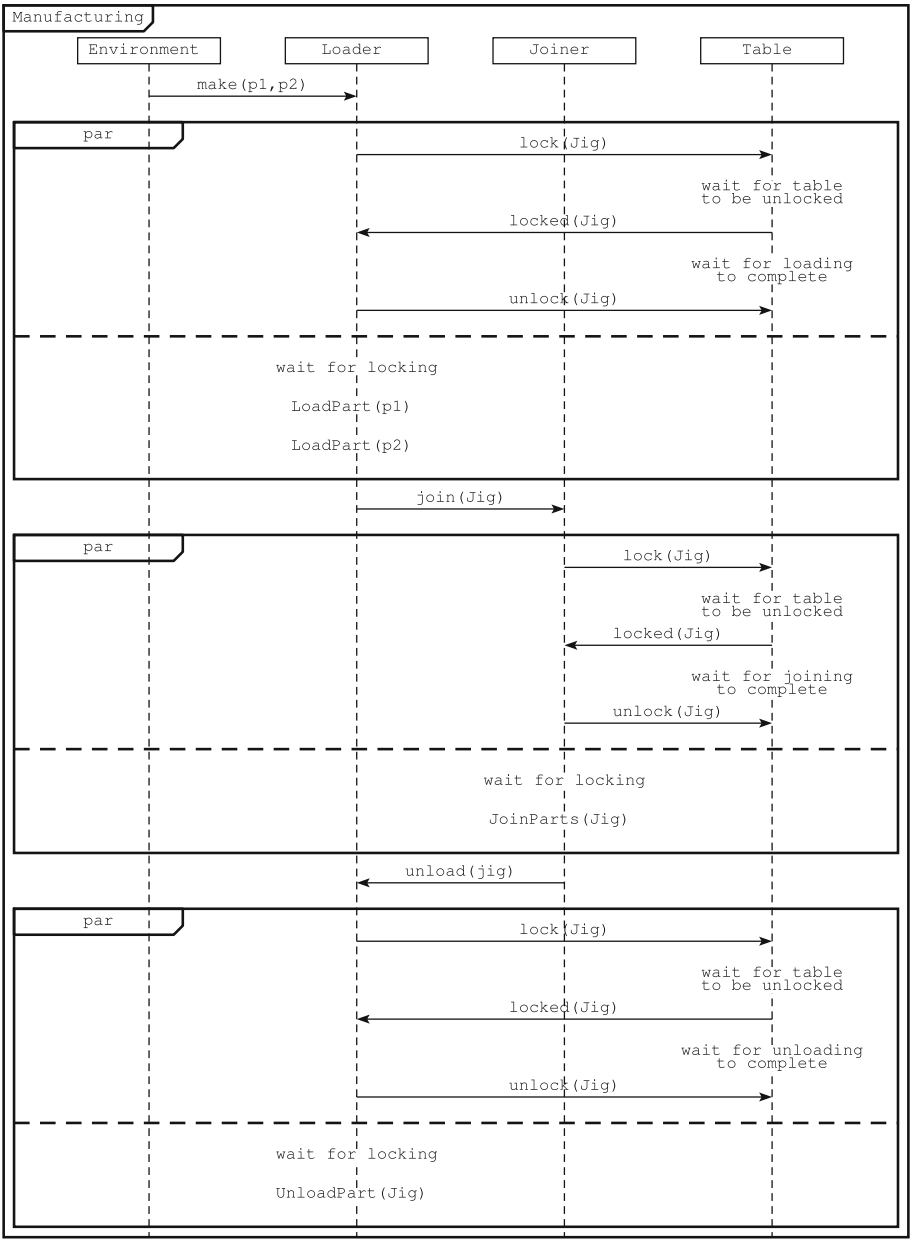


Fig. 23 The Manufacturing protocol in AUML

68. make(p1,p2), lock(jig1), locked(jig1), LoadPart(p1), unlock(jig1) …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

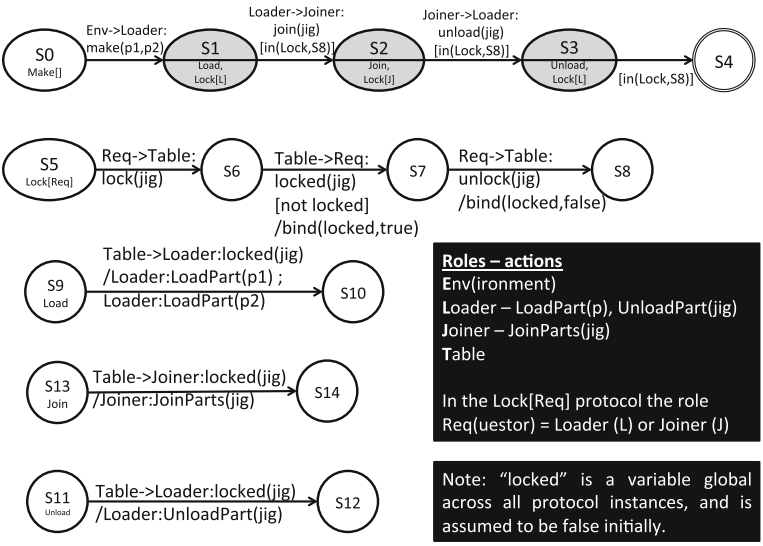


Fig. 24 The Manufacturing protocol in HAPN

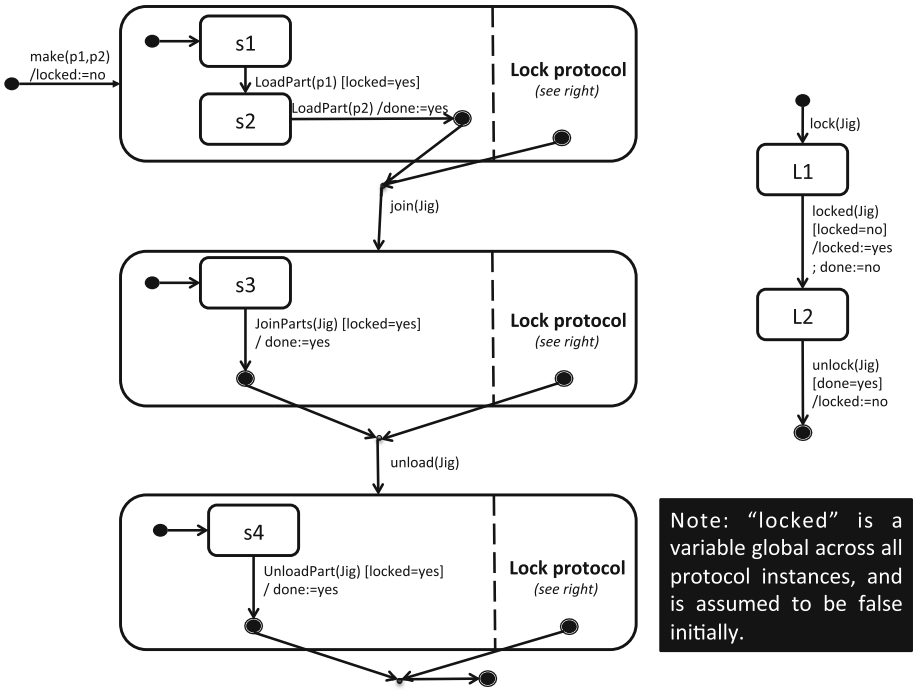


Fig. 25 The Manufacturing protocol in the Statechart notation

69. make(p1,p2), lock(jig1), locked(jig1), LoadPart(p1), LoadPart(p2), join(jig1), JoinParts(jig1), …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
70. make(p1,p2), lock(jig1), locked(jig1), LoadPart(p1), LoadPart(p2), unlock(jig), join(jig1), lock(jig1), locked(jig1), JoinParts(jig1), …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

The time now is: _____ ***(please fill in the current time)***

The remaining questions involve a situation where two parts are being manufactured at the same time, which corresponds to two instances of the interaction protocol that evolve simultaneously, the notation "1:*message*" is used to indicate that the *message* is part of the first protocol (and similarly "2:*message*" indicates it's part of the second protocol). In order to visually emphasise this, the messages associated with the first protocol instance are in *red italic* and those associated with the second protocol instance are in **blue bold**.

Recall that the two parts are being manufactured on the same table, and that the table needs to be locked while parts are being loaded, joined, or unloaded.

71. *1:make(p1,p2)* ; *1:lock(jig1)* ; *1:locked(jig1)* ; *1:LoadPart(p1)* ; *1:LoadPart(p2)* ; *1:unlock(jig1)* ; **2:make(p3,p4)** ; **2:lock(jig2)** ; **2:locked(jig2)** ; **2:LoadPart(p3)** ; **2:LoadPart(p4)** ; **2:unlock(jig2)** …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
72. *1:make(p1,p2)* ; **2:make(p3,p4)** ; **2:lock(jig2)** ; *1:lock(jig1)* ; *1:locked(jig1)* ; *1:LoadPart(p1)* ; *1:LoadPart(p2)* ; *1:unlock(jig1)* ; **2:locked(jig2)** ; **2:LoadPart(p3)** ; **2:LoadPart(p4)** ; **2:unlock(jig2)** …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
73. *1:make(p1,p2)* ; **2:make(p3,p4)** ; **2:lock(jig2)** ; **2:locked(jig2)** ; *1:lock(jig1)* ; *1:locked(jig1)* ; *1:LoadPart(p1)* ; *1:LoadPart(p2)* ; *1:unlock(jig1)* ; **2:LoadPart(p3)** ; **2:LoadPart(p4)** ; **2:unlock(jig2)** …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)
74. *1:make(p1,p2)* ; **2:make(p3,p4)** ; **2:lock(jig2)** ; **2:locked(jig2)** ; **2:LoadPart(p3)** ; **2:LoadPart(p4)** ; *1:lock(jig1)* ; *1:locked(jig1)* ; *1:LoadPart(p1)* ; *1:LoadPart(p2)* ; *1:unlock(jig1)* ; **2:unlock(jig2)** …
❒ ok ❒ no
How confident are you of your answer? ❒ 1 (not at all) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very)

The time now is: _____ ***(please fill in the current time)***

Creating a new interaction protocol

Voice-driven music system for visually impaired

Services such as Spotify and Pandora allow users to search, play, and manage music. Software that provides these services generally allow interactions via keyboard and mouse, both of which are unsuitable for visually impaired users. Hence, the need for a system that allows voice driven interactions, thus enabling visually impaired users to use such services.
We need to build a protocol specification for a system that will allow users to search for music based on certain information, select a song from the search results, and then play the selected track.

Search The user will initiate the search by providing an artist, a song keyword, or both. For the system, the song search keyword needs to be specified for it to be able to search. That is, song information is compulsory but an artist information is optional. Upon receiving the required information the system will fetch the search results. If the search results are empty, the system will convey this to the user and the search will start again from scratch.

Select The system will present search results to the user one at a time by reading a songs title and its artists name. The user can navigate through the results by sending next and previous messages. In case of an invalid navigation message (e.g., a previous message on the first result), the system will respond with an invalid command message.

Play When a user sends a play message, the song whose title and artist was most recently presented is played by the system. When a song is being played, the user can control the player be sending messages for repeating the song, pausing a song, and resuming a paused song.

Cancel The user can close the software anytime by sending a cancel message. At the level of the protocol, the cancel message simply terminates the active communication between the user and the system.

This task requires you to model a protocol specification that will meet the above requirements using the **Agent UML** (**AUML**) notation. If you make any assumptions please state them clearly. You may use any aspects of the **Agent UML** (**AUML**) notation and if needed you can add extra comments. However, please try to avoid the need for additional comments. You do not need to think about the internal development of the system, such as the programming language, etc., only the protocol specification.

Please note the time now, before you start on designing your protocol.

The time now is: _____ ***(please fill in the current time)***

Please write your protocol on the provided paper, noting any assumptions required.

Once you have completed the protocol, please note the time.

The time now is: _____ ***(please fill in the current time)***

Post-survey

(Note that the order of protocols in the post-survey was varied to match the order in which the participant used the protocols)

How would you rate the **Agent UML** (**AUML**) notation …
Easy to read? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)
Easy to understand? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)
Easy to write? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)

What aspects of the **Agent UML** (**AUML**) notation made it easy to read/write?

What aspects of the **Agent UML** (**AUML**) notation made it hard to read/write?

What, if anything, would you propose to change in the notation to improve it?

How would you rate the **Hierarchical Agent Protocol Notation** (**HAPN**) …
Easy to read? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)
Easy to understand? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)
Easy to write? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)

What aspects of the **Hierarchical Agent Protocol Notation** (**HAPN**) made it easy to read/write?

What aspects of the **Hierarchical Agent Protocol Notation** (**HAPN**) made it hard to read/write?

What, if anything, would you propose to change in the notation to improve it?

How would you rate the **Statechart** notation …
Easy to read? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)
Easy to understand? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)
Easy to write? ❒ 1 (very hard) ❒ 2 ❒ 3 ❒ 4 ❒ 5 (very easy)

What aspects of the **Statechart** notation made it easy to read/write?

What aspects of the **Statechart** notation made it hard to read/write?

What, if anything, would you propose to change in the notation to improve it?

Please rank the notations in order from easiest to understand to least easy to understand by putting numbers 1 (easiest), 2, and 3 (hardest) below:

_____ AUML _____ HAPN _____ Statecharts

Any other comments?

The time now is: _____ ***(please fill in the current time)***

Thank you for your time!

Appendix B: Marking rubric for the creation task

Search:

1. Does the protocol allow for specifying only the SongKeyword before searching?
2. Does it deal sensibly with an attempt by the user to search with only an Artist (not allowed, since SongKeyword is required)?
3. Does it allow Artist and SongKeyword to be specified in either order (if both are provided)?
4. Does it check for the search returning an empty result?

Select:

5. Does it allow the user to use Next and Prev (or similar) to navigate?
6. Does it check for invalid navigation? (e.g. Prev when at the start)

Play:

7. Does it allow for Repeat to be issued at while playing? *(technically also possible while paused, but did not require this)*
8. Does it allow for Pause & Resume, but only in alternating order? (i.e. cannot Resume unless paused, and cannot Pause unless playing)

Cancel:

9. Does it allow for a Cancel message to be received at any point in the interaction?
10. Does the Cancel message terminate the protocol?

Overall: Is the overall sequence of Search;Select;Play realized?

11. Is a successful search followed by reading the song details? (and hence Select)
12. Is a Play message followed by the protocol playing the song?

References

1. Alur, R., Kannan, S., & Yannakakis, M. (1999). Communicating hierarchical state machines. In J. Wiedermann, P. van Emde Boas, & M. Nielsen (Eds.), *Automata, languages and programming* (pp. 169–178). Berlin: Springer.
2. Ancona, D., Ferrando, A., & Mascardi, V. (2016). Comparing trace expressions and linear temporal logic for runtime verification. In E. Ábrahám, M. Bonsangue, & E. B. Johnsen (Eds.), *Theory and practice of formal methods: Essays dedicated to Frank de Boer on the occasion of his 60th birthday* (pp. 47–64). Cham: Springer.
3. Baldoni, M., Baroglio, C., Calvanese, D., Micalizio, R., & Montali, M. (2016). Data and norm-aware multiagent systems for software modularization (position paper). In M. Baldoni, J. P. Müller, I. Nunes, & R. Zalila-Wenkstern (Eds.), *Engineering multi-agent systems (EMAS) (informal workshop proceedings)* (pp. 23–38). Singapore.
4. Baldoni, M., Baroglio, C., & Capuzzimati, F. (2014). A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technology*, *14*(4), 23:1–23:23.
5. Baldoni, M., Baroglio, C., Marengo, E., & Patti, V. (2013). Constitutive and regulative specifications of commitment protocols: A decoupled approach. *ACM Transactions on Intelligent Systems and Technologies*, *4*(2), 22.
6. Baldoni, M., Baroglio, C., Marengo, E., Patti, V., & Capuzzimati, F. (2014). Engineering commitment-based business protocols with the 2CL methodology. *Autonomous Agents and Multi-Agent Systems*, *28*(4), 519–557.
7. Basu, S., Bultan, T., & Ouederni, M. (2012). Deciding choreography realizability. In J. Field & M. Hicks (Eds.), *Proceedings of the 39th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)* (pp. 191–202). Philadelphia, Pennsylvania: ACM.
8. Bhattacharya, K., Caswell, N. S., Kumaran, S., Nigam, A., & Wu, F. Y. (2007). Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, *46*(4), 703–721.
9. Bultan, T., Su, J., & Fu, X. (2006). Analyzing conversations of web services. *IEEE Internet Computing*, *10*(1), 18–25.
10. Cabac, L., Duvigneau, M., Moldt, D., & Rölke, H. (2005). Modeling dynamic architectures using nets-within-nets. In G. Ciardo & P. Darondeau (Eds.), *26th International conference on applications and theory of Petri nets (ICATPN), volume 3536 of lecture notes in computer science* (pp. 148–167). Berlin: Springer.
11. Chopra, A. K., Christie, S. H. V., & Singh, M. P. (2017). Splee: A declarative information-based language for multiagent interaction protocols. In S. Das, E. Durfee, K. Larson, & M. Winikoff (Eds.), *Autonomous agents and multi-agent systems (AAMAS)* (pp. 1054–1063). São Paulo, Brazil: IFAAMAS.
12. Chopra, A. K., & Singh, M. P. (2015). Cupid: Commitments in relational algebra. In B. Bonet & S. Koenig (Eds.), *Proceedings of the twenty-ninth AAAI conference on artificial intelligence* (pp. 2052–2059). Austin, TX: AAAI Press.
13. Chopra, A. K., & Singh, M. P. (2016). Custard: Computing norm states over information stores. In C. M. Jonker, S. Marsella, J. Thangarajah, & K. Tuyls (Eds.), *Autonomous agents & multiagent systems (AAMAS)* (pp. 1096–1105). Singapore: IFAAMAS.
14. Cohn, D., & Hull, R. (2009). Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Engineering Bulletin*, *32*(3), 3–9.
15. Desai, N., Chopra, A. K., & Singh, M. P. (2009). Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Transactions on Software Engineering and Methodology*, *19*(2). doi:10.1145/1571629.1571632.
16. Desai, N., & Singh, M. P. (2008). On the enactability of business protocols. In D. Fox & C. P. Gomes (Eds.), *Proceedings of the twenty-third AAAI conference on artificial intelligence* (pp. 1126–1131). Chicago, Il: AAAI Press.
17. Eshuis, R. (2009). Reconciling statechart semantics. *Science of Computer Programming*, *74*(3), 65–99.
18. Günay, A., Winikoff, M., & Yolum, P. (2015). Dynamically generated commitment protocols in open systems. *Autonomous Agents and Multi-Agent Systems*, *29*(2), 192–229.
19. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231–274.
20. Huget, M.-P., Bauer, B., Odell, J., Levy, R., Turci, P., Cervenka, R., & Zhu, H. (2003). *FIPA modeling: Interaction diagrams*. On www.auml.org under "Working Documents". FIPA Working Draft (version 2003-07-02).
21. Huget, M.-P., & Odell, J. (2005). Representing agent interaction protocols with agent UML. In J. Odell, P. Giorgini, & J. P. Müller (Eds.), *Agent-oriented software engineering V: 5th international workshop, AOSE 2004, Revised Selected Papers* (pp. 16–30). Berlin: Springer.

22. Huget, M.-P., Odell, J., & Bauer, B. (2004). The AUML approach. In F. Bergenti, M. P. Gleizes, & F. Zambonelli (Eds.), *Methodologies and software engineering for agent systems* (pp. 237–257). Berlin: Springer.
23. Jarvis, J., Rönnquist, R., Jarvis, D., & Jain, L. C. (2008). A conceptual model for holonic manufacturing execution. In *Holonic execution: A BDI approach, volume 106 of studies in computational intelligence* (pp. 33–42). Berlin: Springer.
24. Jarvis, J., Rönnquist, R., McFarlane, D., & Jain, L. (2006). A team-based holonic approach to robotic assembly cell control. *Journal of Network and Computer Applications*, *29*(2–3), 160–176.
25. Kazhamiakin, R., & Pistore, M. (2006). Analysis of realizability conditions for web service choreographies. In E. Najm, J. Pradat-Peyre, & V. Donzeau-Gouge (Eds.), *Formal techniques for networked and distributed systems FORTE, volume 4229 of lecture notes in computer science* (pp. 61–76). Berlin: Springer.
26. Koning, J., Huget, M., Wei, J., & Wang, X. (2001). Extended modeling languages for interaction protocol design. In M. Wooldridge, G. Weiß, & P. Ciancarini (Eds.), *Agent-oriented software engineering II, second international workshop, Revised Papers and Invited Contributions, volume 2222 of lecture notes in computer science* (pp. 68–83). Berlin: Springer.
27. Lanese, I., Guidi, C., Montesi, F., & Zavattaro, G. (2008). Bridging the gap between interaction- and process-oriented choreographies. In *Sixth IEEE international conference on software engineering and formal methods* (pp. 323–332).
28. Mazouzi, H., Fallah-Seghrouchni, A. E., & Haddad, S. (2002). Open protocol design for complex interactions in multi-agent systems. In *Autonomous agents & multiagent systems (AAMAS)* (pp. 517–526). ACM.
29. Montali, M., Calvanese, D., & De Giacomo, G. (2014). Verification of data-aware commitment-based multiagent system. In A. L. C. Bazzan, M. N. Huhns, A. Lomuscio, & P. Scerri (Eds.), *Autonomous agents and multi-agent systems (AAMAS)* (pp. 157–164). Paris, France: IFAAMAS.
30. Moody, D. L. (2009). The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, *35*(6), 756–779.
31. Moody, D. L., & van Hillegersberg, J. (2009) Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In D. Gasevic, R. Lämmel, & E. V. Wyk (Eds.), *First international conference on software language engineering, volume 5452 of lecture notes in computer science* (pp. 16–34). Berlin: Springer.
32. Nigam, A., & Caswell, N. S. (2003). Business artifacts: An approach to operational specification. *IBM Systems Journal*, *42*(3), 428–445.
33. Reisig, W. (1985). *Petri nets: An introduction*. EATCS Monographs on Theoretical Computer Science. Berlin: Springer.
34. Singh, M. P. (1998). Agent communication languages: Rethinking the principles. *Computer*, *31*, 40–47.
35. Singh, M. P. (2011). Information-driven interaction-oriented programming: BSPL, the Blindingly simple protocol language. In *Proceedings of the 10th international conference on autonomous agents and multiagent systems (AAMAS)* (pp. 491–498).
36. Singh, M. P. (2011). LoST: Local state transfer—An architectural style for the distributed enactment of business protocols. In *IEEE international conference on web Services (ICWS)* (pp. 57–64). IEEE Computer Society.
37. Singh, M. P. (2012). Semantics and verification of information-based protocols. In *Proceedings of the 11th international conference on autonomous agents and multiagent systems (AAMAS)* (pp. 1149–1156).
38. Singh, M. P. (2014). Bliss: Specifying declarative service protocols. In *Proceedings of the 11th IEEE international conference on services computing (SCC)* (pp. 1–8).
39. Taleghani, A., & Atlee, J. (2006). Semantic variations among UML StateMachines. In O. Nierstrasz, J. Whittle, D. Harel, & G. Reggio (Eds.), *Model driven engineering languages and systems, volume 4199 of lecture notes in computer science* (pp. 245–259). Berlin: Springer.
40. Telang, P. R., & Singh, M. P. (2012). Comma: A commitment-based business modeling methodology and its empirical evaluation. In W. van der Hoek, L. Padgham, V. Conitzer, & M. Winikoff (Eds.), *International conference on autonomous agents and multiagent systems AAMAS* (pp. 1073–1080). Valencia, Spain: IFAAMAS.
41. Thielscher, M., & Zhang, D. (2010). From general game descriptions to a market specification language for general trading agents. In E. David, E. Gerding, D. Sarne, & O. Shehory (Eds.), *Agent-mediated electronic commerce. Designing trading strategies and mechanisms for electronic markets* (pp. 259–274). Berlin: Springer.
42. Winikoff, M. (2006). Designing commitment-based agent interactions. In *IEEE/WIC/ACM international conference on intelligent agent technology (IAT)*.

43. Winikoff, M. (2006). Implementing flexible and robust agent interactions using distributed commitment machines. *Multiagent and Grid Systems*, *2*(4), 365–381.
44. Winikoff, M. (2007). Implementing commitment-based interactions. In *Autonomous Agents and multi-agent systems (AAMAS)* (pp. 873–880).
45. Winikoff, M., Liu, W., & Harland, J. (2004). Enhancing commitment machines. In J. Leite, A. Omicini, P. Torroni, & P. Yolum (Eds.), *Declarative agent languages and technologies II, number 3476 in lecture notes in artificial intelligence* (pp. 198–220). Berlin: Springer.
46. Winikoff, M., & Padgham, L. (2013). Agent oriented software engineering, chapter 15. In G. Weiß (Ed.), *Multiagent systems* (2nd ed., pp. 695–757). Cambridge, MA: MIT Press.
47. Yadav, N., Padgham, L., & Winikoff, M. (2015). A tool for defining agent protocols in HAPN: (demonstration). In *Autonomous agents and multiagent systems (AAMAS)* (pp. 1935–1936). IFAAMAS
48. Yolum, P. (2005). Towards design tools for protocol development. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, & M. Wooldridge (Eds.), *Autonomous agents and multi-agent systems (AAMAS)* (pp. 99–105). Utrecht, The Netherlands: ACM Press.
49. Yolum, P., & Singh, M. (2002). Commitment machines. In J.-J. C. Meyer & M. Tambe (Eds.), *Agent theories, architectures, and languages (ATAL), volume 2333 of lecture notes in computer science* (pp. 235–247). Berlin: Springer
50. Yolum, P., & Singh, M. P. (2002). Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Autonomous agents and multiagent systems (AAMAS)* (pp. 527–534).