Strabo: Declarative Specification and Implementation of Agentic Interaction Protocols

Samuel H. Christie V[1], Munindar P. Singh[1], and Amit K. Chopra[2]

[1] North Carolina State University
{schrist, singh}@ncsu.edu
[2] Lancaster University
amit.chopra@lancaster.ac.uk

Abstract. The last few years have witnessed major advances in the modeling and implementation of multiagent systems based on declarative interaction protocols. Our contribution, Strabo, establishes the relevance of these advances to ongoing industry efforts in Agentic AI. Specifically, we consider UCP, the *Universal Commerce Protocol*, a recent Google-led effort to standardize e-commerce interactions for AI agents. Our exercise is in two parts. One, we model the part of UCP dealing with checkouts as a declarative Langshaw protocol and implement agents using Peach, a programming model for Langshaw. This part of the exercise brings out the advantages of formal, declarative specifications. Two, we show that Peach agents can interoperate with UCP agents implemented by Google, thereby establishing the fidelity of our approach with respect to UCP. Such interoperation enables the incremental introduction of declarative protocols and agents into a conventional setting, indicating a pathway by which EMAS ideas could influence practice without demanding a wholesale update.

1 Introduction

Interaction protocols model multiagent systems and serve as blueprints for engineering agents. Recent work, as exemplified by BSPL [13, 14, 15] and Langshaw [16], has emphasized formal, declarative approaches for specifying protocols. Formal specification enables verifying protocols for correctness before any agents are implemented. Declarative specifications better reflect stakeholder intuitions about the meaning of interactions [12] and, consequently, support flexible interactions between agents. Programming models that facilitate implementing agents based on declarative protocols [3, 5, 8, 9] support improved code structure and dealing with changing requirements. Protocols are conceptual abstractions. Not specifying them doesn't mean they don't have to be implemented. It's just that, absent explicit protocols, developers end up encoding the relevant interaction constraints directly in low-level code, which is cumbersome and hides assumptions and errors.

The Agentic AI paradigm has emerged concurrently with the advances in protocols. Its basic value proposition is that LLM-based agents will do all kinds

of tasks on behalf of their users. Many of these tasks will involve interacting with the agents of the other users, e.g., for shopping, booking travel, and so on. This recognition has led to a flurry of industry efforts aimed at standardizing interactions between LLM agents. Among the prominent ones are the Google-led Universal Commerce Protocol (UCP) [17], which standardizes e-commerce interactions such as checkout and order processing, and the Agent2Agent (A2A) protocol [1], which provides a general-purpose framework for interagent task delegation. These protocols are specified informally in terms of JSON schemas, prose descriptions, and example HTTP traces.

Contributions. Our objective is to establish the relevance of the "academic" declarative approaches for "practical" uses. To this end, we contribute *Strabo*, a general method for exploiting declarative protocols in Langshaw in the context of ongoing industry efforts that model protocols via JSON-RPC or REST API calls. The method has two components: (1) Mapping the industry effort into a Langshaw protocol and implementing agents using Peach [5], a programming model for Langshaw, and (2) Creating a bridge layer that enables Peach agents to interoperate with industry agents, thus demonstrating interoperation with legacy agents built with conventional methods.

We explain Strabo by modeling UCP's *Checkout* capability as a Langshaw protocol and implementing Peach agents to enact that protocol. We demonstrate the interoperability of these Peach agents with sample agents available in a UCP code repository[3] to establish the fidelity of the Langshaw protocol with respect to UCP.

This exercise establishes that formal, declarative protocols are directly applicable to industry efforts. Whereas UCP is informally specified, leaving important aspects ambiguous, the Langshaw model of UCP specifies interaction constraints formally, bringing precision without sacrificing practical interoperability. Our exercise also surfaced implicit assumptions in UCP and revealed enhancements required by Langshaw and Peach.

2 Background on UCP, Langshaw, and Peach

We now describe the existing approaches that we build upon in this paper.

2.1 UCP, the Universal Commerce Protocol

UCP addresses e-commerce interactions involving the roles of Platform (customer), Business (merchant), Credential Provider (digital wallets), and Payment Provider (e.g., PayPal and Stripe). The idea is that Platform, as representative of the shopper, shops with Business and uses the Credential Provider to generate payment tokens that the Business redeems with the Payment Provider.

[3] UCP sample code is available at https://github.com/Universal-Commerce-Protocol/samples

A typical UCP interaction proceeds as follows. Both PLATFORM and BUSINESS publish UCP profiles that list their capabilities. Capabilities are the broad activities involved in an e-commerce interaction, such as *Checkout*, *Order*, and *Identity Linking* (for computing loyalty rewards, and so on). Capabilities may be extended. For example, *Fulfillment* extends *Checkout* to support the delivery of physical goods. Based on its own profile and that of PLATFORM, BUSINESS determines if they can interoperate.

Capabilities. A capability is defined by a JSON schema. Table 1 gives some fields in the *Checkout* capability (the full schema is in Appendix A).

Field	Type	Required	Description
id	string	Yes	Unique identifier of the checkout session.
line_items	LineItemResponse[]	Yes	List of line items being checked out.
buyer	Buyer	No	Representation of the buyer.
status	string	Yes	Checkout state indicating the current phase and required action.
totals	TotalResponse[]	Yes	Different cart totals.
messages	Message[]	No	List of messages with error and info about the checkout session state.
payment	PaymentResponse	Yes	Payment configuration and handler info.
order	OrderConfirmation	No	Details about an order created for this session.

Table 1. The *Checkout* capability (extracted from [17]).

Operations. All operations on an instance of the *Checkout* capability—i.e., *Create*, *Get*, *Update*, *Complete*, and *Cancel*—are performed by PLATFORM. Each operation specifies an input and an output schema comprising required and optional fields. The output schema specifies *Checkout* instances that are returned by BUSINESS in response. BUSINESS recomputes the instance based on the input in an *Update*. Listings 1 and 2, with minor divergences from UCP documentation, show a *Create* and its (truncated) response.

Listing 1. A *Create* operation.

```
POST /checkout-sessions HTTP/1.1
{"line_items": [
  {"item": {
     "id": "item_123",
     "title": "Red T-Shirt",
     "price": 2500},
   "id": "li_1", "quantity": 2}]}
```

Listing 2. Response to Listing 1 (truncated).

```
HTTP/1.1 201 Created
{
  "id": "chk_1234567890",
  "status": "incomplete",
  "messages": [{"type": "error", "code": "missing",
                "path": "$.buyer.email",
                "content": "Buyer email is required"}],
  "currency": "USD",
  "totals": [{"type": "subtotal", "amount": 5000},
             {"type": "tax", "amount": 400},
             {"type": "total", "amount": 5400}],
  "payment": {"instruments": [/*...*/]},
  "line_items": [/*...*/], "links": [/*...*/]
}
```

The response indicates that the buyer's email is missing. PLATFORM addresses this exception with an *Update* (`PUT /checkout-sessions/{id}`) that supplies the buyer and, optionally, the fulfillment fields; BUSINESS recomputes and returns the full instance. Once the instance reaches status `ready_for_complete`, PLATFORM sends *Complete* (`POST /checkout-sessions/{id}/complete`) with payment instrument data.

2.2 Langshaw

Langshaw [16] is a declarative protocol language for specifying multiagent interactions. We illustrate its concepts using the checkout protocol in Listing 3, which we discuss in detail in Section 3.

A Langshaw protocol defines *who* interacts (roles), *what* constitutes a complete enactment, and *what they do* (actions with attributes). The **who** clause names the roles—in our example, PLATFORM and BUSINESS. The **do** clause lists actions, each performed by a specific role and with some attributes. For instance, *Create* is performed by PLATFORM and carries attributes such as line_items and buyer.

Some attributes are designated as *key*. Keys correlate actions into *enactments*, that is, protocol instances. A protocol declares one or more attributes as keys in its **what** clause; all actions bearing the same key values belong to the same enactment. In *SimpleUCP*, cid is the sole key, so all actions sharing a cid value constitute one checkout session.

An attribute can only be bound once within a given enactment; once bound, it is immutable. Performing an action either reuses the binding of each parameter or generates a new one. Action names are themselves bound as attributes when the action occurs, so an action that references another action's name depends on it having occurred. For example, *Created* references *Create*, establishing a causal dependency: *Created* can only occur after *Create*.

The **sayso** clause assigns data ownership—which role may bind which attributes. In the simplest case, a sayso entry gives a role absolute ownership over

an attribute: only that role may produce its binding. However, sayso entries may also specify priority orderings among roles, allowing controlled sharing of attributes where one role's binding takes precedence over another's. The **nono** clause declares mutual exclusion (e.g., *Completed* and *Canceled* cannot both occur in the same enactment), and the **nogo** clause declares directional exclusion (e.g., once *Cancel* occurs, *Complete* is blocked, but not vice versa). The **what** clause also specifies a completion goal: which actions must occur for an enactment to be considered complete.

2.3 Peach

Peach [5] is a programming model for building agents that participate in Langshaw protocols. A developer instantiates a PeachAdapter (our tooling) by specifying a protocol, the role the agent plays, and an executor that handles communication and the underlying state model. The adapter implements the protocol's semantics: it tracks which actions are feasible given the current state of the enactment, enforces sayso and causal dependencies, and manages key correlation. Developers build agent logic on top of this adapter rather than reimplementing protocol constraints in application code.

The adapter exposes a small API. Crucially, adapter.enabled() returns the actions the agent may currently perform, given the enactment state. The developer selects an action, binds its attribute values via FeasibleAction.bind(), and submits the result via adapter.attempt(), which returns a promise. The promise resolves when the action is finalized or rejects if the action fails (e.g., due to a conflicting concurrent action), allowing the developer to handle failures explicitly. To react to other agents' actions, the developer registers observation handlers using the @adapter.observation decorator. A @adapter.on_completion handler fires when the protocol's completion goal is satisfied. This API is the same regardless of the executor: Peach currently supports a synchronous executor that connects to a SyncServer providing shared state with either instant or batched finalization, and an asynchronous executor based on direct message passing.

A key aspect of Peach is its enablement-based programming model (from BSPL [13] and Kiko [9]). Rather than requiring the developer to determine which actions are valid after each event—consulting the protocol specification and encoding the operational logic in event handlers—the adapter computes the set of currently feasible actions and presents them directly. The developer (or, in principle, an LLM operating the agent) simply selects from the enabled actions and provides bindings. This shifts the burden of protocol compliance entirely to the adapter: the developer makes domain decisions, not protocol decisions.

Operationally, each adapter computes enabled actions from the protocol and the current state of the enactment, and concurrent attempts within the Peach MAS are reconciled at the SyncServer using its configured finalization policy (instant or batched). Because correlation is by key, agents engaged in overlapping enactments do not interfere; each enactment's state is independent. A formal operational semantics is given in [5].

3 Modeling UCP in Langshaw

There are potentially several ways of capturing UCP's *Checkout* capability in Langshaw. We focus here on an *atomic* variant that captures the simplest complete checkout: all information is submitted in a single action, exposing the essential data dependencies and completion conditions. An *incremental* variant that decomposes updates into separate actions, making data flow and authority explicit, is given in Appendix C. Our exercise surfaces several assumptions that UCP leaves implicit.

3.1 A Closer Look at UCP *Checkout*

We make the following remarks about UCP.

- R_1 Each *Checkout* instance has a unique identifier. Instances are thought of as *sessions*. The *Create* does not bear the instance identifier. It is generated in the response.
- R_2 Some fields, e.g., id, are scalar whereas others, e.g., line_items, are composite.
- R_3 Some fields, e.g., line_items, are *required* in an instance; others, e.g., buyer, are optional. An operation's response may additionally specify some of the optional fields, e.g., order in the response to a *Complete*, as required.
- R_4 Updates can be partial, possibly overriding existing information.
- R_5 Operations are conceptually request-response in nature, as is evident from their input-output style specification.
- R_6 UCP implicitly assumes a synchronous model for operations. Specifically, Platform may issue an operation on an instance only after the previous operation on the instance has returned a response. Otherwise, Platform and Business could have incompatible views of the instance. To see this, suppose Platform performs an *Update* on an instance and, before it receives a response, also performs a *Complete* on it. Now, if *Complete* is processed by Business before *Update* (maybe because *Update* is delayed in the network), the order placed may not reflect the intentions of Platform.

3.2 UCP in Langshaw

Listing 3 gives *SimpleUCP*, a checkout protocol in Langshaw that captures the UCP checkout interactions. The protocol is *atomic* in the sense that Platform submits all checkout information in a single *Create* action; Business responds with either *Created* (bearing the server-generated id) or *Failed* (bearing reason, indicating that creation was rejected—e.g., due to missing required fields or a server error). If the checkout was created successfully, Platform may *Complete* or *Cancel* it: a *Complete* yields a *Completed* action carrying the order details, and a *Cancel* yields a *Canceled* acknowledgment.

Listing 3. *SimpleUCP* checkout protocol in Langshaw.

```
SimpleUCP
who Platform, Business
what cid key, Completed or Failed or Canceled
do
  Platform: Create(cid,line_items, currency, buyer,
      payment_pref, discount_codes, fulfillment_pref)
  Business: Created(cid, Create, id, totals, payment,
      discounts, fulfillment)
  Business: Failed(cid, Create, reason)
  Platform: Complete(cid, Created, id, payment_data,
      risk_signals)
  Business: Completed(cid, Complete, order)
  Platform: Cancel(cid, Created, id)
  Business: Canceled(cid, Cancel)
sayso
  Platform: line_items, currency, buyer, discount_codes,
      fulfillment_pref, payment_pref, payment_data,
      risk_signals
  Business: id, totals, payment, discounts, fulfillment,
      order, reason
nono
  Completed Failed
  Created Failed
  Completed Canceled
nogo
  Cancel -/> Complete
```

The *nono* clauses enforce mutual exclusion among outcomes: *Created* and *Failed* cannot both occur in a session (creation either succeeds or fails), *Completed* and *Failed* cannot both occur (a completed checkout cannot also be failed), and *Completed* and *Canceled* cannot both occur (a checkout is completed or canceled, not both). The *nogo* from *Cancel* to *Complete* adds a *directional* exclusion: once *Cancel* is finalized, *Complete* is disabled. Cancellation is thus a terminal outcome, acknowledged by Business with *Canceled*, rather than a state from which the checkout could still complete. These constraints creates an action dependency structure visualized in Figure 1.

Table 2 shows the mapping from UCP operations to *SimpleUCP* actions. Notice that UCP's *Update* is realized in *SimpleUCP* via two actions: the *Cancel* of the existing checkout and the *Create* of a new one.

4 Programming Peach UCP Agents

We now show how developers build agents for a Langshaw protocol using Peach. The central point is the separation of concerns: agent code contains only domain decisions (which actions to take and what values to bind), whereas the adapter handles protocol compliance, key correlation, and communication. We show a Platform agent and a Business proxy agent for *SimpleUCP*.

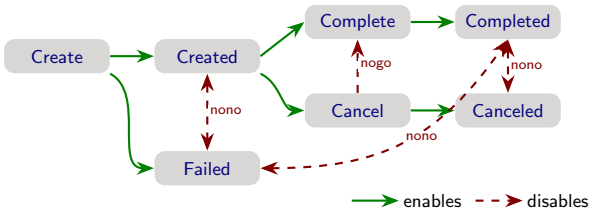

Fig. 1. Enable/disable structure of the *SimpleUCP* protocol (Listing 3). Solid green arrows are causal dependencies—performing an action enables its successors. Dashed red arrows are exclusions: the bidirectional *nono* edges forbid conflicting outcomes, while the directional *nogo* edge from *Cancel* to *Complete* disables completion once a checkout is canceled.

UCP	**HTTP**	***SimpleUCP***	**Role**
Create	`POST /checkout-sessions`	*Create*	PLATFORM
	201 Created	*Created*	BUSINESS
	4xx/5xx	*Failed*	BUSINESS
Get	`GET /checkout-sessions/{id}`	—	—
Update	`PUT /checkout-sessions/{id}`	(*Cancel*,*Create*)	PLATFORM
Complete	`POST .../complete`	*Complete*	PLATFORM
	200 OK	*Completed*	BUSINESS
Cancel	`POST .../cancel`	*Cancel*	PLATFORM
	200 OK	*Canceled*	BUSINESS

Table 2. Mapping from UCP operations to SimpleUCP actions. Get is unused because the adapter tracks protocol state. Update is realized at the protocol level by the Platform agent issuing (Cancel, Create); on the wire, the outbound Create itself is a POST followed by a PUT to supply fields UCP does not accept in the initial request.

4.1 Platform Agent

The PLATFORM agent drives the checkout process. Listing 4 shows the adapter setup: the developer loads a protocol, creates an adapter for the PLATFORM role, and registers handlers for observations and protocol completion. Listing 5 shows the enablement-driven handlers, each invoked by the adapter when its action becomes feasible. The adapter also ensures consistency and correctness by presenting only currently valid actions, propagating bindings from prior observations, and accepting via FeasibleAction.bind() only attributes the agent has sayso over.

Listing 4. PLATFORM agent core: adapter setup and observation handlers.

```
protocol = LangshawProtocol.load("protocols/simple-ucp.lsh")
adapter = PeachAdapter("platform", protocol, "Platform",
    executor)
done = asyncio.Event()
@adapter.on_completion
async def on_done(keys, mas_id):
```

```
    done.set()
@adapter.observation('Created', 'Failed', 'Completed')
async def on_observation(action, performer):
    print(f"Observed {action.action.name} by {performer}")
```

Listing 5. PLATFORM agent enablement-driven handlers: the adapter invokes each handler when its action becomes feasible.

```
@adapter.on_enabled('Create')
async def on_create_enabled(fa):
    await adapter.attempt([fa.bind(
        line_items=SAMPLE_LINE_ITEMS, currency="USD",
        buyer=SAMPLE_BUYER, payment_pref={},
        discount_codes={"codes": []},
        fulfillment_pref=SAMPLE_FULFILLMENT_PREF)])
@adapter.on_enabled('Complete')
async def on_complete_enabled(fa):
    await adapter.attempt([fa.bind(
        payment_data=SAMPLE_PAYMENT_DATA,
        risk_signals={"ip": "127.0.0.1"})])
```

5 Interoperation with Google's UCP Agents

5.1 Architecture

Figure 2 shows the runtime architecture. The Peach MAS comprises a PLATFORM agent that drives the checkout flow and a BUSINESS proxy adapter that bridges to the external merchant. Both sides share access to a SyncServer that maintains the protocol's social state. When the PLATFORM agent performs a *SimpleUCP* action, the SyncServer notifies the Business Proxy, which uses its built-in field mapping to construct one or more corresponding UCP requests, sends them to the actual UCP BUSINESS over HTTP, parses the responses, and performs the corresponding *SimpleUCP* action. Field-name translation is configured inline via the route() DSL; neither the PLATFORM agent nor the proxy contains hand-written HTTP code, with the only imperative code being orchestration for actions that map to multiple HTTP calls (such as *Create*, which requires a `POST` followed by a `PUT`).

5.2 Proxy

The proxy bridges the Langshaw protocol to the external merchant HTTP API. ProxyAdapter is a PeachAdapter subclass with built-in HTTP client and field mapping. Routes are defined via the route() DSL, and for each route, the adapter auto-registers an observation handler that fires the HTTP request and submits the response action. For *SimpleUCP*, *Create* requires orchestration: a single Langshaw *Create* maps to a POST to `/checkout-sessions` followed by a PUT (to supply fields like discounts and fulfillment that UCP does not accept in the

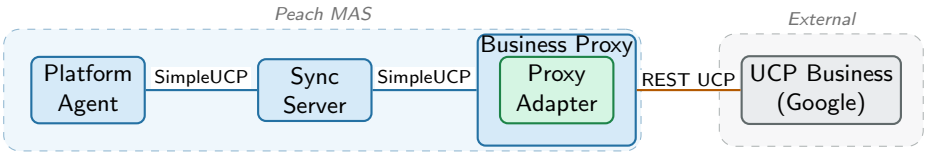

Fig. 2. Strabo Architecture. The Peach MAS interoperates with Google's UCP merchant via a Strabo proxy, which translates Langshaw actions in *SimpleUCP* to REST calls using built-in field mapping configured via the route() DSL.

initial request). Listing 6 shows the core of the proxy: routes are declared inline, and the *Create* action is excluded from auto-proxy so that a custom handler can orchestrate the two HTTP calls. All other actions (e.g., *Complete*, *Cancel*) are handled automatically. The PUT is declared as an auxiliary Update -¿ Updated route (Listing 6); *Update* and *Updated* are internal to the proxy and are not *SimpleUCP* actions.

Listing 6. ProxyAdapter setup with route() DSL and custom *Create* orchestration.

```
proxy = ProxyAdapter(
    "simple_proxy", protocol, "Business", executor,
    base_url=merchant_url, internal=["cid"],
    header_hook=ucp_header_hook, exclude=["Create"])
proxy.route("Create -> Created", "POST /checkout-sessions")
proxy.route("Update -> Updated", "PUT
    /checkout-sessions/{id}",
    request={"discount_codes": "$.discounts",
             "fulfillment_pref": "$.fulfillment",
             "payment_pref": "$.payment"})
proxy.route("Complete -> Completed",
    "POST /checkout-sessions/{id}/complete")
proxy.route("Cancel -> Canceled",
    "POST /checkout-sessions/{id}/cancel")
@proxy.observation('Create')
async def on_create(action, performer):
    if performer == proxy.name: return
    create_resp = await proxy.request("Create",
        {k: action.bindings[k]
         for k in ("line_items", "currency", "buyer")})
    update_resp = await proxy.request("Update",
        {**action.bindings, "id": create_resp["id"]})
    combined = {**action.bindings, **create_resp,
        **update_resp}
    await proxy.attempt([proxy.build_action("Created",
        combined)])
```

For each route() declaration, ProxyAdapter auto-generates an observation handler that fires when the trigger action is observed, sends the corresponding HTTP request, and submits a fixed response action back into the protocol.

This assumes a one-to-one mapping between request and response actions. Protocols with alternative responses (e.g., *Create* yielding either *Created* or *Failed* depending on the HTTP status) are supported but require a custom observation handler that inspects the response and selects the appropriate action, as illustrated above for *Create*.

5.3 Field Mapping

Field mapping is configured inline via the route() DSL (visible in Listing 6). Each route() call declares a round-trip: a pair of Langshaw actions mapped to an HTTP endpoint. The adapter supports three kinds of mappings. *Identity* mappings require no configuration: when a Langshaw attribute name matches the corresponding JSON field name (e.g., buyer ↔ `$.buyer`), the adapter translates automatically. *Explicit* mappings handle cases where names differ: the request and response keyword arguments specify non-identity translations (e.g., discount_codes ↔ `$.discounts`, or payment_pref ↔ `$.payment`). *Excluded* attributes—those declared via internal—never appear in HTTP traffic (e.g., cid, which is the Langshaw enactment key). The complete mapping between UCP fields and *SimpleUCP* attributes is given in Appendix B.

5.4 Interactions

Figure 3 shows the interactions for a complete *SimpleUCP* checkout. Each Langshaw action submitted by Platform is intercepted by the proxy, translated into HTTP calls using the route's field mapping, and the response is returned as a Langshaw observation—keeping the Platform agent entirely free of HTTP concerns.

6 Evaluation

We evaluate our approach along four dimensions. *Clarity*: Does the Langshaw specification make implicit assumptions explicit? *Generality*: Does the formalism naturally support protocol variants that UCP does not distinguish? *Fidelity*: Can Peach agents interoperate with Google's UCP reference implementation? *Economy*: How much agent code is needed and what fraction is domain logic versus boilerplate?

6.1 Clarity and Generality

We focus on how the *SimpleUCP* Langshaw protocol addresses the UCP features noted in the remarks above.

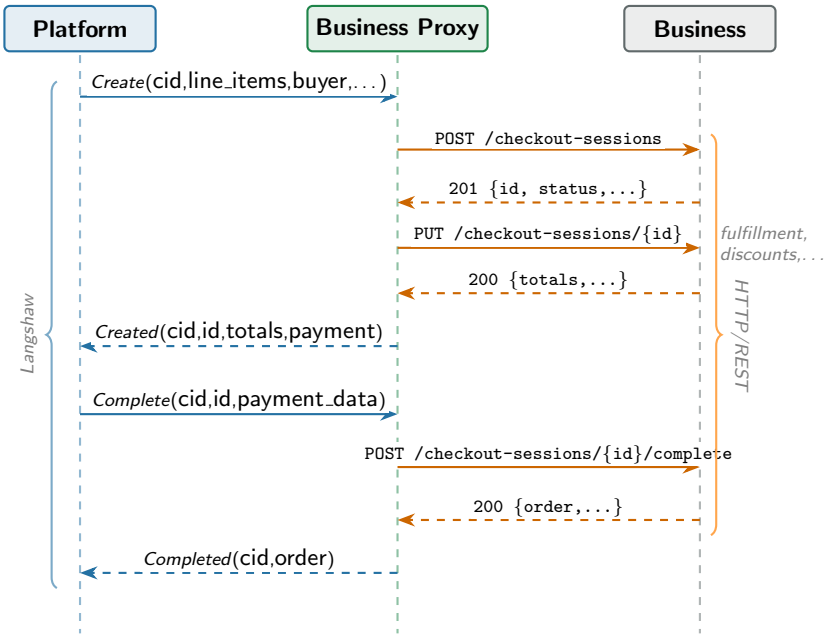

Fig. 3. Interactions for a *SimpleUCP* checkout via the proxy. PLATFORM speaks Langshaw; the proxy bridges to the merchant's HTTP API using its built-in field mapping. The proxy-internal `PUT` supplies fields (e.g., discounts and fulfillment) that UCP does not accept in the initial `POST`. Solid arrows: requests; dashed: responses.

Checkout identity (Remark R_1). In UCP, checkouts are identified by id, which is generated by BUSINESS in response to PLATFORM's *Create*. Such a model not only reflects a centralized mindset, but also is incompatible with meaning: Being unidentified, a UCP *Create* by itself cannot be meaningful. Since Langshaw's purpose is to support meaning, it identifies every action instance. In *SimpleUCP*, this is accomplished by declaring cid as the key for every action, including *Create*. For compatibility with UCP, *SimpleUCP* introduces the (non-key) attribute id, over which BUSINESS has sayso; it is generated by BUSINESS in *Created*, and is used by PLATFORM in subsequent actions such as *Complete* and *Cancel*. For interoperation with Google's sample agents, the proxies strip out cid.

Scalar versus composite fields (Remark R_2). UCP distinguishes scalar fields (e.g., id, a string) from composite ones (e.g., line_items, an array of objects). Langshaw treats all attributes uniformly; their internal structure is orthogonal to the protocol specification. Peach handles scalar and composite values equally well by treating composite fields as opaque values compared using deep equality (a built-in Python feature over dictionaries) and serializing all values for storage as JSON, which is compatible with the format received from UCP.

Modeling optional attributes (Remark R_3). UCP marks certain fields as optional: buyer, fulfillment, and discount_codes may or may not be supplied. One way to capture such optionality is via null values. In the *SimpleUCP* protocol, all attributes appear on the *Create* action. An attribute that the agent does not wish to supply is bound to a null value; the proxy strips null-valued fields before constructing the HTTP request. This approach is concise but relies on a convention external to the protocol specification: Langshaw does not distinguish a deliberately null binding from a missing one. A better approach would be to add support for `optional` action attributes in the Langshaw language. We leave this extension as future work.

The incremental variant in Appendix C avoids null values altogether by giving each optional concern its own action.

Overwrite semantics (Remark R_4). UCP allows Platform to selectively update fields that have already been bound in a session. In Langshaw, attributes may be bound at most once in an enactment; they are immutable thereafter. Moreover, *SimpleUCP* models a checkout as a complete, immutable object. To revise a checkout, Platform cancels the current session and creates a new one with the updated data—a cancel-and-recreate pattern that is semantically equivalent to an in-place update. This comes at some cost: a *Cancel* is ambiguous, as observers cannot distinguish a genuine user cancellation from a revision. In exchange, each session has a definitive state, and it is always clear which version of the checkout is being completed. Mutable amendment would conflict with Langshaw's commitment to immutability, which lets agents reason about enactment state without a shared mutable store. The incremental variant's version key (Appendix C) is the principled counterpart to UCP's in-place update: each amendment is a fresh subenactment, and the application interprets the collection of versions as its domain requires (e.g., treating the highest sortable v as authoritative). The incremental versioning example is deliberately minimal; a richer structure—e.g., recording parent–child lineage or scoping versioning to attribute groups—could more closely reflect UCP's selective-update semantics without relaxing immutability.

Request-response and synchronous ordering (Remarks R_5 and R_6). UCP operations are conceptually request-response pairs, and UCP implicitly assumes that operations on a session are issued sequentially. Langshaw replaces both conventions with structural causal dependencies: an action that references another action's name can only occur after it. *SimpleUCP* is inherently synchronous—a single *Create*–*Complete* exchange—so no additional ordering mechanism is needed. The incremental variant makes the design choice explicit by declaring v (version) as key (Appendix C). Attribute binding authority, implicit in UCP's JSON schemas, is made explicit through **sayso**.

UCP additionally requires idempotency keys in HTTP headers to prevent the server from reprocessing a duplicate request and repeating side-effects (e.g., creating a second order). The UCP sample generates a fresh random key per request (Listing 7), which provides protection only at the network level—where the same

request is resent with its original headers intact—but not against application-level retries where a new key would be generated. In our implementation, idempotency key generation is encapsulated in the ProxyAdapter, keeping this transport concern transparent to the agent.

Table 3 summarizes how each remark is addressed.

#	UCP Model	*SimpleUCP* Model
1	*Create* lacks instance ID	Explicit key cid; *Created* introduces id
2	Scalar vs. composite fields	Uniform attributes; types orthogonal
3	Required vs. optional fields	Null values (*SimpleUCP*) or separate actions (incremental)
4	Overwrite semantics	Avoided in *SimpleUCP*; version key in incremental (Appendix)
5	Request-response convention	Structural causal dependencies
6	Synchronous ordering (implicit)	*SimpleUCP* is inherently synchronous; incremental uses v key (Appendix)

Table 3. How Langshaw addresses assumptions implicit in UCP's specification.

6.2 Fidelity: Interoperation with Google's UCP Agents

We tested interoperation end-to-end by connecting a Peach Platform agent to Google's UCP reference Business server via the ProxyAdapter. The Platform agent successfully completed checkout sessions with the Google server, with the adapter's built-in field mapping handling name translation and session ID correlation transparently.

We also verified the reverse configuration, in which a Google Platform drives a Strabo Business. Here a ProxyAdapter plays the Platform role: it accepts UCP HTTP requests from the Google client, translates them into *SimpleUCP* actions for a Peach Business agent, and returns the responses over HTTP (Appendix D, Figure 5). Both directions interoperate with Google's sample agents, so Strabo can stand in for either role.

6.3 Economy: Comparison with Google's UCP Agent

We compare the Peach agent implementation with a comparable agent built directly against Google's UCP SDK. In the Peach agent, ordering constraints, session management, and data ownership are enforced by the adapter—the agent code contains no ordering logic. In the direct HTTP agent, the developer must manually track session identifiers, check status before completing (e.g., verifying `ready_for_complete`), and construct correct HTTP requests. The Business proxy is particularly compact: the ProxyAdapter auto-generates observation handlers from route() declarations, so adding a new UCP capability (e.g., Orders) requires only a new protocol file and a few route definitions, not new agent code.

Table 4 reports lines of code for both configurations. In each, the Peach implementation is a fraction of the corresponding Google sample: the PLATFORM agent is 97 lines against a 915-line REST client, and the BUSINESS agent is 137 lines against a 2,868-line server, with the proxies (163 and 439 lines) carrying field mapping and HTTP orchestration. The difference is most visible in equivalent operations; Listings 7 and 8 show checkout creation in the direct HTTP client and the Peach agent, respectively.

Listing 7. Checkout creation using the UCP SDK directly.

```
item1 = ItemCreateRequest(id="bouquet_roses", title="Red-
    Rose")
line_item1 = LineItemCreateRequest(quantity=1, item=item1)
payment_req = PaymentCreateRequest(
    instruments=[], selected_instrument_id=None,
    handlers=supported_handlers)
buyer_req = Buyer(full_name="John-Doe",
    email="john@example.com")
create_payload = CheckoutCreateRequest(
    currency="USD", line_items=[line_item1],
    payment=payment_req, buyer=buyer_req)
headers = get_headers()  # UCP-Agent, signature,
    idempotency-key
json_body = create_payload.model_dump(
    mode="json", by_alias=True, exclude_none=True)
response = client.post(
    "/checkout-sessions", json=json_body, headers=headers)
checkout_id = response.json().get("id")
```

Listing 8. Checkout creation using the Peach adapter.

```
@adapter.on_enabled('Create')
async def on_create_enabled(fa):
    await adapter.attempt([fa.bind(
        line_items=SAMPLE_LINE_ITEMS, currency="USD",
        buyer=SAMPLE_BUYER, payment_pref={})])
```

7 Discussion

Agent2Agent (A2A) [1], originally developed by Google and now managed by the Linux Foundation, is a widely adopted protocol for interagent task delegation. A client discovers remote agents via *AgentCard*s and delegates tasks; tasks progress through states (`submitted`, `working`, `completed`, `failed`, `canceled`) and may involve multiturn conversations. A2A and UCP are complementary: A2A handles discovery and delegation while UCP handles domain-specific interactions such as checkout. Like UCP, A2A is mostly informally specified; its state machine is described in prose, and constraints such as whether a canceled task may be resumed are ambiguous. A2A's *context identifier* is analogous to a Langshaw

		Peach			Google
Scenario	**Component**	*SimpleUCP*	Incremental	Direct	baseline
Strabo PLATFORM	PLATFORM agent	97	113	128	915
	BUSINESS proxy	163	171	134	—
Strabo BUSINESS	BUSINESS agent	137	—	—	2,868
	PLATFORM proxy	439	486	—	—

Table 4. Lines of code for the Checkout capability in both interoperation configurations. *Strabo* PLATFORM: a Peach PLATFORM agent drives the Google BUSINESS sample. *Strabo* BUSINESS: a Google PLATFORM drives a Peach BUSINESS agent (Appendix D). The **Google** column is the corresponding sample's REST client (915) and server (2,868); Peach counts exclude the shared runtime, and proxy rows include inline route() field-mapping declarations.

enactment key, and the bridge architecture presented here could equally apply: model the A2A task lifecycle as a Langshaw protocol and bridge to A2A's JSON-RPC transport.

Baldoni et al. [2] recently demonstrated BSPL integration with the SARL agent language, exploring a different point in the implementation design space.

Chopra et al. [6] survey the IOP toolkit Strabo builds on: protocol languages, the Tango verifier, and enablement-based programming models such as Kiko [9] and Mandrake [8]. That toolkit targets *greenfield* systems of IOP agents; Strabo inherits its expressiveness, semantics, and programming model from Langshaw and Peach, and makes an orthogonal contribution: interoperation with agents built to informal HTTP/JSON specifications that know nothing of information protocols. The ProxyAdapter and route() DSL map declarative actions onto such APIs, so a new capability needs only a protocol and a few routes, not new agent code, making a path to adopting declarative protocols incrementally rather than wholesale.

Strabo is concerned with bridging formal protocols to existing HTTP APIs. Chopra and Singh [7] recently proposed Fluid, which brings social norms to web-based multiagent systems—a complementary direction that could inform how normative constraints are layered atop protocols like those modeled here.

Historically, agent communication was standardized through FIPA-ACL [11] and KQML [10], which focused on speech-act semantics for individual messages. UCP and A2A represent a new generation of industry protocols that emphasize structured interactions, but share with their predecessors the challenges of informal specification and inadequate support for meaning [12].

The end-to-end test demonstrates feasibility but is not an exhaustive evaluation of protocol conformance.

8 Conclusion

We have shown that formal, declarative MAS protocol techniques are directly applicable to industry e-commerce protocols. Modeling UCP's Checkout capability in Langshaw made explicit several assumptions—most notably around attribute binding authority and ordering—that UCP's informal specification leaves implicit. The Langshaw formalism naturally supports protocol variants that UCP does not distinguish, exposing a design space that practitioners can reason about; an incremental variant is explored in Appendix C. Peach agents built against the Langshaw protocol interoperate with Google's UCP reference implementation via Strabo's ProxyAdapter, which bridges protocol actions to HTTP round-trips with inline field mapping, demonstrating that formal specification and practical interoperability are not at odds. Industry protocols like UCP and A2A are defining the interaction patterns for AI-assisted commerce and multiagent coordination; our exercise demonstrates that MAS protocol research can contribute precision and verifiability to these efforts.

This exercise was focused on operations in the UCP *Checkout* capability; UCP additionally includes *Order*, *Identity Linking*, and extensible capabilities, each of which would need its own protocol and route definitions. We model only the Platform–Business interactions; the protocol also involves roles corresponding to credential and payment providers. Extending Strabo to address additional capabilities and roles should be a rewarding direction, potentially testing Langshaw's support for protocol composition (a core facility in BSPL) and dynamic role bindings (supported in Splee [4], a BSPL extension).

Several directions remain for future work. Adding `optional` attribute support to Langshaw would address the null-value convention that *SimpleUCP* currently relies on, making optionality a first-class protocol concept rather than an implementation convention. The bridge architecture generalizes beyond UCP: A2A's task lifecycle could be modeled as a Langshaw protocol and bridged to its JSON-RPC transport, testing the approach against a complementary industry standard with a different interaction style. Finally, a benefit of Langshaw that we did not exploit is decentralized enactments—where each agent maintains its own local view of the protocol state without a shared server. Exploiting this property would enable peer-to-peer agent interaction more naturally aligned with the distributed nature of agentic AI deployments, and would test whether the ProxyAdapter architecture extends to settings without central coordination.

Reproducibility. Our software is available at https://gitlab.com/masr/strabo.

Bibliography

[1] A2A: Agent2Agent protocol (2025), https://a2aprotocol.ai/, last accessed: 2025-08-11

[2] Baldoni, M., Baroglio, C., Galland, S., Micalizio, R., Outay, F., Tedeschi, S.: Interaction protocols in an imperative agent-oriented programming language: The case of BSPL and SARL. In: Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 2426–2427. IFAAMAS, Detroit (May 2025). https://doi.org/10.5555/3709347.3743891

[3] Baldoni, M., Christie V, S.H., Singh, M.P., Chopra, A.K.: Orpheus: Engineering multiagent systems via communicating agents. In: Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI). pp. 23135–23143. AAAI, Philadelphia (Feb 2025). https://doi.org/10.1609/aaai.v39i22.34478

[4] Chopra, A.K., Christie V, S.H., Singh, M.P.: Splee: A declarative information-based language for multiagent interaction protocols. In: Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1054–1063. IFAAMAS, São Paulo (May 2017). https://doi.org/10.5555/3091125.3091274

[5] Chopra, A.K., Christie V, S.H., Singh, M.P.: Peach: Program each agent and communicate howsoever. In: Proceedings of the 25th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). IFAAMAS, Paphos, Cyprus (May 2026), accepted

[6] Chopra, A.K., Christie V, S.H., Singh, M.P.: Tools for implementing multiagent systems based on protocols. In: Collier, R., Mascardi, V., Ricci, A. (eds.) Agents and Multi-Agent Systems Development: Platforms, Toolkits, Technologies, chap. 8, pp. 211–230. Springer Nature, Cham, Switzerland (2026). https://doi.org/10.1007/978-3-032-01082-7_8

[7] Chopra, A.K., Singh, M.P.: Fluid: Social norms-based multiagent systems on the Web. In: In Proceedings of the 13th Engineering Multiagent Systems Workshop. LNCS, vol. 16407, pp. 62–79. Springer (2025)

[8] Christie V, S.H., Chopra, A.K., Singh, M.P.: Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **36**(1), 16:1–16:30 (Apr 2022). https://doi.org/10.1007/s10458-021-09540-8

[9] Christie V, S.H., Singh, M.P., Chopra, A.K.: Kiko: Programming agents to enact interaction protocols. In: Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1154–1163. IFAAMAS, London (May 2023). https://doi.org/10.5555/3545946.3598758

[10] Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the 3rd International Conference on Information and Knowledge Management. pp. 456–463. ACM Press,

Gaithersburg, Maryland (Dec 1994). https://doi.org/10.1145/191246.191322

[11] FIPA: FIPA agent communication language specifications. http://www.fipa.org/repository/aclspecs.html (2002), FIPA: The Foundation for Intelligent Physical Agents. Accessed 2025-01-20

[12] Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer **31**(12), 40–47 (Dec 1998). https://doi.org/10.1109/2.735849

[13] Singh, M.P.: 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. IFAAMAS, Taipei (May 2011). https://doi.org/10.5555/2031678.2031687

[14] Singh, M.P.: Semantics and verification of information-based protocols. In: Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1149–1156. IFAAMAS, Valencia, Spain (Jun 2012). https://doi.org/10.5555/2343776.2343861

[15] Singh, M.P., Christie V, S.H.: Tango: Declarative semantics for multiagent communication protocols. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI). pp. 391–397. IJCAI, Online (Aug 2021). https://doi.org/10.24963/ijcai.2021/55

[16] Singh, M.P., Christie V, S.H., Chopra, A.K.: Langshaw: Declarative interaction protocols based on sayso and conflict. In: Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI). pp. 202–210. IJCAI, Jeju, Korea (Aug 2024). https://doi.org/10.24963/ijcai.2024/23

[17] UCP: Universal commerce protocol (Jan 2026), https://ucp.dev, last accessed: 2026-02-20

A *Checkout* Capability

Table 5 gives the complete specification of the *Checkout* capability.

Field	Type	Required	Description
ucp	ResponseCheckout	Yes	Protocol metadata and checkout response root.
id	string	Yes	Unique identifier of the checkout session.
line_items	LineItemResponse[]	Yes	List of line items being checked out.
buyer	Buyer	No	Representation of the buyer.
status	string	Yes	Checkout state indicating the current phase and required action.
currency	string	Yes	ISO 4217 currency code.
totals	TotalResponse[]	Yes	Different cart totals.
messages	Message[]	No	List of messages with error and info about the checkout session state.
links	Link[]	Yes	Links to be displayed by the PLATFORM (e.g., Privacy Policy, TOS).
expires_at	string	No	RFC 3339 expiry timestamp; default TTL often ∼6 hours.
continue_url	string	No	URL for checkout handoff and session recovery.
payment	PaymentResponse	Yes	Payment configuration and handler info.
order	OrderConfirmation	No	Details about an order created for this session.

Table 5. The *Checkout* capability (extracted from [17]).

B Mapping Between UCP Fields and *SimpleUCP* Attributes

Table 6 gives the complete field-to-attribute mapping used by the ProxyAdapter. Identity mappings use the same name on both sides. Explicit mappings arise where UCP and *SimpleUCP* use different names for the same data. Fields marked *dropped* have no protocol equivalent in *SimpleUCP* because their information is either implicit in the action sequence or unnecessary for the Langshaw model.

C Incremental *Checkout* Protocol

The incremental protocol (Listing 9) generalizes *SimpleUCP* to support UCP's multistep update flow. Rather than submitting all information in a single *Create*, PLATFORM may issue separate actions for each concern: *SetBuyer*, *ApplyDiscounts*, *SetFulfillment*, and *SetPayment*, each carrying only its own attributes.

UCP Field	*SimpleUCP* Attribute	Mapping
Checkout response fields		
ucp	—	dropped (protocol metadata)
id	id	identity
line_items	line_items	identity
buyer	buyer	identity
status	—	dropped (implicit in action sequence)
currency	currency	identity
totals	totals	identity
messages	reason	partial (error messages → *Failed*)
links	—	dropped
expires_at	—	dropped
continue_url	—	dropped
payment	payment	identity (in *Created*)
order	order	identity (in *Completed*)
Request fields (sent in Update/Complete)		
discounts	discount_codes	explicit (route() DSL)
fulfillment	fulfillment_pref	explicit (route() DSL)
payment	payment_pref	explicit (route() DSL)
payment_data	payment_data	identity
risk_signals	risk_signals	identity
Langshaw-internal (not in UCP)		
—	cid	enactment key; stripped by proxy

Table 6. Mapping between UCP fields and *SimpleUCP* attributes. UCP's payment field is overloaded: in request context it carries payment preferences (payment_pref); in response context it carries payment configuration (payment).

Business responds to each with a corresponding acknowledgment. This decomposition makes data flow explicit—each action pair affects only the attributes it names. Regarding optional attributes (Remark R_3): each concern has its own action, so the agent simply omits any action it does not need. No null values are required; the protocol specification itself captures which combinations of actions are valid.

The incremental protocol introduces a second key, v, which serves as a version identifier. Each mutation action and its response carry v, and since an attribute may only be bound once per composite key, successive mutations are distinguished by their v values—each operates within its own subenactment. This also addresses overwrite semantics (Remark R_4): successive updates produce distinct bindings rather than overwriting previous ones, and agents can observe all bindings across subenactments to merge values and selectively override at the application level. The version key does not itself impose ordering; however, the developer may choose sortable values (e.g., ULIDs) and operate on the latest version, achieving the sequential behavior that UCP assumes by convention (Remark R_6). The protocol makes this design choice explicit.

Listing 9. Incremental UCP checkout protocol in Langshaw.

```
IncrementalUCP
who Platform, Business
what eid key, v key, Completed or Canceled
do
  Platform: Create(eid, line_items, currency, buyer, payment)
  Business: Created(eid, Create, id, payment_config, messages)
  Business: StatusChange(eid, Created, status, totals,
      messages)
  Platform: Update(eid, v, Created, id, line_items, currency)
  Business: Updated(eid, v, Update, line_items, messages)
  Platform: SetBuyer(eid, v, Created, id, buyer)
  Business: BuyerSet(eid, v, SetBuyer, buyer, messages)
  Platform: ApplyDiscounts(eid, v, Created, id,
      discount_codes)
  Business: DiscountsApplied(eid, v, ApplyDiscounts,
      discounts_applied, messages)
  Platform: SetFulfillment(eid, v, Created, id,
      fulfillment_pref)
  Business: FulfillmentSet(eid, v, SetFulfillment,
      fulfillment, messages)
  Platform: SetPayment(eid, v, Created, id, payment_pref)
  Business: PaymentSet(eid, v, SetPayment, payment_config,
      messages)
  Platform: Complete(eid, v, Create, id, payment_data,
      risk_signals)
  Business: Completed(eid, v, Complete, order, messages)
  Platform: Cancel(eid, v, Created, id)
  Business: Canceled(eid, v, Cancel, messages)
  Business: Shipped(eid, Completed, order_id,
      tracking_number, tracking_url, carrier)
  Business: Delivered(eid, Shipped, order_id)
sayso
  Platform: line_items, currency, buyer, discount_codes,
      fulfillment_pref, payment_pref, payment_data,
      risk_signals, payment
  Business: id, status, totals, discounts_applied,
      fulfillment, payment_config, order, order_id,
      tracking_number, tracking_url, carrier, messages
nono
  Completed Canceled
nogo
  Cancel -/> Complete
```

D Interaction Sequences

This appendix collects interaction sequences that complement the running example in Section 3. Figure 4 shows UCP *Checkout* as the operation-level request–

response exchanges that its informal specification describes. Figures 5 and 6 show two further bridged enactments: the reverse configuration in which a Google PLATFORM drives a Strabo BUSINESS, and a canceled checkout.

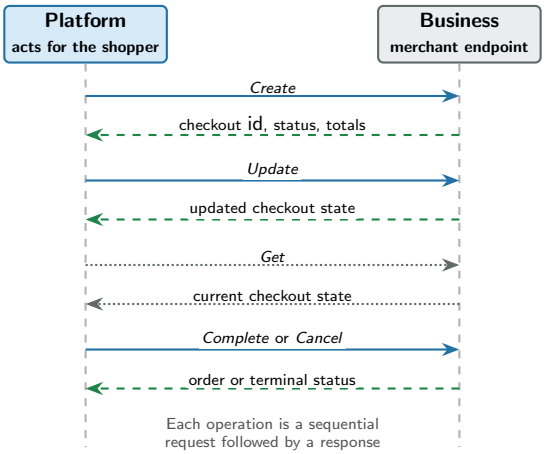

Fig. 4. UCP *Checkout* as operation-level request–response exchanges. Solid arrows are PLATFORM requests, dashed arrows are BUSINESS responses, and dotted arrows mark the optional *Get*.

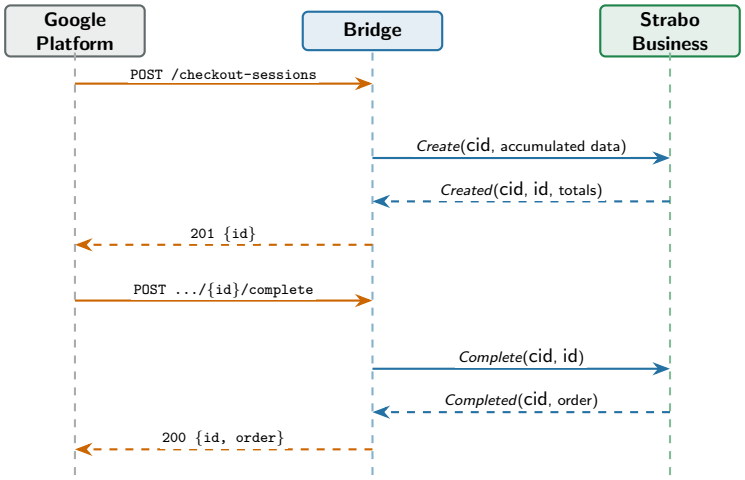

Fig. 5. Reverse configuration: a Google PLATFORM drives a Strabo BUSINESS through the bridge. HTTP requests and responses (orange) are translated to and from *SimpleUCP* actions (blue).

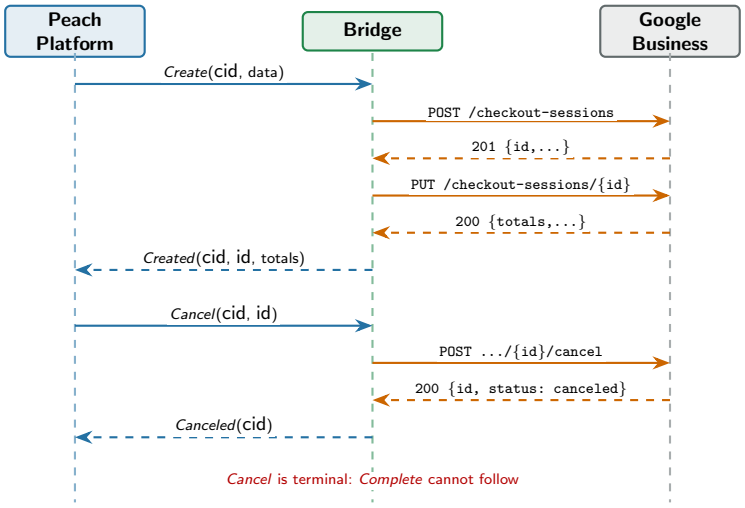

Fig. 6. A canceled checkout through the bridge. After the merchant confirms cancellation, the bridge emits the *SimpleUCP Canceled* response; the *nogo* constraint then blocks any subsequent *Complete*.