JFP **34**, e12, 69 pages, 2024.
doi:10.1017/S0956796824000091

Programming and reasoning about actors that share state

SAM CALDWELL
Northeastern University, Boston, MA 02115, USA
(*e-mail:* `samc@ccs.neu.edu`)

TONY GARNOCK-JONES
Maastricht University, Maastricht, Netherlands
(*e-mail:* `tony.garnock-jones@maastrichtuniversity.nl`)

MATTHIAS FELLEISEN
Northeastern University, Boston, MA 02115, USA
(*e-mail:* `matthias@ccs.neu.edu`)

Abstract

Actor languages realize concurrency via message passing, which most of the time is easy to use. Empirical code inspection provides evidence, however, that on occasion, programmers wish to have an actor share some of its state with others. The dataspace model adds a tightly controlled state-exchange mechanism, dubbed dataspace, to the actor model for just this purpose. Experience with dataspaces suggests that this form of sharing calls for linguistic constructs that allow programmers to state temporal aspects of actor conversations. In response, this paper presents the facet notation: its theory, its type system, its behavioral type system, and some first experiences with an implementation.

1 Introduction

Actor languages and libraries support an easy-to-use mechanism for expressing concurrent computations. The program spawns as many actors as needed, and actors strictly communicate via messages. Pure message passing eliminates the possibility of race conditions arising from a lack of mutual exclusion, a major problem in shared-state concurrency. An investigation (Tasharofi *et al.*, 2013) of actively used actor programming libraries suggests, however, that programmers often use actors with threads and shared state, if available. When interviewed, programmers tend to point to the ease of sharing some state between computational components running in parallel. In short, programmers wish for some amount of sharing among actors.

The dataspaces actor model supplements the message-exchanges of the actor model with a space for sharing state in a controlled manner (Garnock-Jones *et al.*, 2014; Garnock-Jones & Felleisen, 2016; Caldwell *et al.*, 2020). It accepts the actor model's premise that computation is conversation—exchange of messages—but it also embraces the demand of conversations for context. Roughly speaking, a context consists of the public information—corresponding to fragments of an actor's current state—that the participants remain aware of for a period of time. Such public information, dubbed an assertion, is placed into a shared dataspace. From a programmer's perspective, an assertion is a part of an actor's state that it wishes to share publicly.

From the dataspace's point of view, an actor is a function that reacts to changes in the published states of other actors. Even a message is a change to this shared state: it shows up and disappears immediately. Indeed, basic message-passing may be expressed with dataspaces in just this manner. The functional, black-box interface for actors lends itself to system-level reasoning but, when it comes to expressing the behavior of an individual actor, a purely functional notation is *not* a good fit. The core programming tasks—situationally engaging and disengaging in behavior and managing and reacting to changes in shared state—are inherently temporal. Hence, the model calls for a novel way of expressing a concurrent actor in a context-sensitive conversation.

In response, this paper presents the *design of a language of facets* and a system for *reasoning about their temporal behavior.* From the perspective of this facet language, an actor consists of several pieces—the facets—each of which represents the actor as a participant in distinct (parts of) conversations. The concept addresses the central concerns of actors:

- engaging and disengaging in conversations;
- maintaining local state;
- permitting access to shared, public state; and
- reacting to external events.

The benefits of facets are evident in improving the organization of dataspace actor programs and supporting a conversational style of concurrency. Furthermore, facets directly enable type-based static reasoning about communication.

Dining Concurrency Researchers. To illustrate our meaning of conversational concurrency, imagine a table of dining and conversing attendees at a conference on concurrency research. Experience suggests a number of common characteristics for such a scenario:

- The conversation is likely to branch into several subconversations as new topics are broached. Previous topics may be returned to—or not.
- The participants may engage in one subconversation or several simultaneously; they may leave one conversation to join another, only to return later to the first one.
- The set of participants changes over time. Late-arrivals may join the table after the conversation commences while early-departers exit the conversation before its conclusion.
- The conversation is contextual. A newcomer to the conversation is likely to listen for a while to pick up the context. Alternatively, this newcomer may explicitly state an interest in the contextual background.
- Finally, these participants identify one another through a number of characteristics, including their presentations at the conference, their interests, their association

with particular institutions, and so on. They may or may not know each others' names. Moreover, a particular participant's identifying characteristic may vary from conversation to conversation.

While Djikstra's dining philosophers illustrate how coordination avoids certain pitfalls of concurrency, namely deadlocked program states, the scenario of dining conference attendees paints a portrait of how developers desire well-designed concurrent programs to behave.

The notions of dataspace and facet equip developers with mechanisms to program in this conversational style: means for representing and sharing context, evolving actor behavior in the face of shifting demands, and a flexible notion of identity and participation in a conversation. Sections 2 and 3 return to these ideas.

Roadmap. The presentation proceeds in three steps: Sections 2 through 4 introduce the ideas via examples; the formal models of facet actors and dataspace programs are the subject of Sections 5, 6, and 7, respectively; and Section 8 states the key theorem concerning the temporal type checking of facet actors. Section 9 briefly explains the implementation, and Section 10 reports the results of evaluating the ability to verify behavioral properties of realistic programs. The remaining sections are about related and future work.

2 Background: Dataspaces, actors

The dataspace actor model generalizes the publish/subscribe protocol of Linda's Tuplespaces (Carriero *et al.*, 1994). When actors wish to share pieces of state with others, they deposit those with a dedicated actor, the *dataspace*. All actors connected to a dataspace can query this shared space. The dataspace thus controls how published state is shared within a group of conversing actors.

A simulation of a smart home allows a comprehensive illustration. Let us say the smart home supports three kinds of devices: smart lights; a presence sensor, which detects how many people are in a room; and a control hub for accessing and controlling the other devices. The simulation expresses each of these devices as an actor (`Light`, `Sensor`, `Hub`) embedded in a common dataspace.

Figure 1 depicts a moment in such a simulation. The connecting dataspace is the large rectangle. Each actor has deposited aspects of its state pertinent to the others; these are called *assertions*.[1] For example, the `Sensor` actor has published the assertion `inRoom[3]`, denoting the number of people in the room. Published pieces of state—assertions—are made up from values (e.g., numbers, strings) as well as immutable data structures (lists, structs).

An actor subscribes to publications of state of others via *assertions of interest*. In the simulation, the `Light` actor turns a bulb on or off based on the occupancy of the room, meaning it needs to know about the `inRoom` assertion. To get this information and keep it up-to-date, it registers `?(inRoom[⋆])` with the dataspace: `inRoom[⋆]` is an *assertion*

[1] The term assertion is inspired by the resemblance to a fact in the database of a Prolog program (Clocksin & Mellish, 1981).

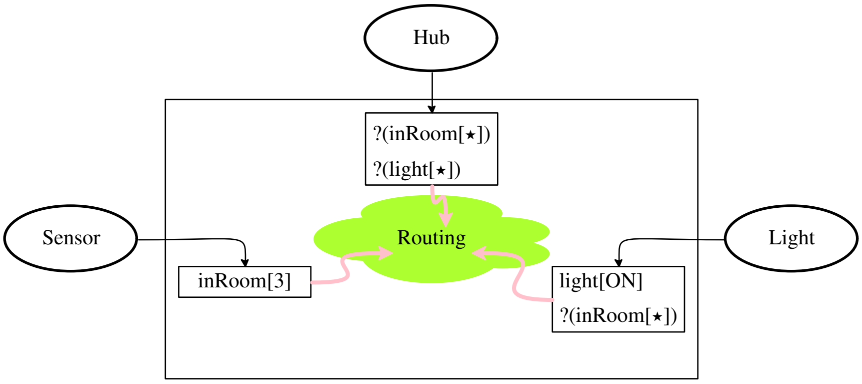

Fig. 1. Dataspace with actors engaged in a smart home conversation.

pattern where ⋆ is a match-anything wildcard and `?` turns it into an assertion of interest, or just interest for short.

Communication in dataspace programs arises from matches between interests and assertions. The dataspace provides a routing mechanism, depicted in Figure 1 as a cloud, for detecting such matches. Routing tracks changes to the set of assertions matching each actor's interests. When this set changes, the dataspace generates an *event* describing the change and informs the interested actor(s). In the running example of the smart home, the presence `Sensor` actor updates its assertion from `inRoom[3]` to `inRoom[2]` if someone leaves. In turn, the dataspace informs the `Light` actor of this change, because of its interest.

An actor's response to an event is one of the following actions:

- it may add and/or subtract assertions from its published state;
- it may spawn additional actors; and
- it may exit, by choice or by an uncaught run-time exception.

When an actor exits, the dataspace removes all of its assertions. Since this removal is a change to the assertion set, it triggers an event for all interested actors. In response, other actors may react to component failures in a graceful manner. For example, when a light burns out in the smart home simulation, its corresponding `Light` actor exits. This triggers the removal of its `light[ON]` assertion. In response, the dataspace informs the `Hub` actor because it is interested in such assertions. This actor may then take appropriate action, such as display a notification.

Dataspace communication is anonymous. That is, the model does not provide actors with an address or process-ID (PID) to serve as a communication handle. Rather, correlation information—that is, data dictating to which actor(s) some data is relevant—must be included in the exchanged assertions and interests. This allows identifying information to be encoded in a domain-specific fashion. For example, the `Light` actor could include its serial number in its assertion `light[`*serial*`, ON]`, allowing future communications specific to that device to refer to it by its serial number.

```
1  (define (light-actor)
2    (start-facet light-facet
3      (field [light-state : Bool ON])
4      (assert (light (! light-state)))
5      (on (asserted (in-room 0)) (:= light-state OFF))
6      (on (retracted (in-room 0)) (:= light-state ON))))
7
8  (spawn (light-actor))
```

Fig. 2. First faceted actor.

Dining Concurrency Researchers. The dataspace model addresses several needs of the conversational style introduced in the dining concurrency setting of the introduction:

- Assertions directly represent conversational context.
- Anonymous actors give a flexible notion of identity and conversational participant.
- The notion of a change-in-state as an event merges the means through which a new participant receives prior context in a conversation with normal communication.
- Similarly, notification of withdrawn assertions is the same—and automatic—whether via normal behavior or exceptional crashes.

In sum, the dataspace model abstracts over the processing of routing events by actors. The model requires only that an actor has a functional event–transducer interface:

$$Event \times State \rightarrow Actions \times State$$

How to express an implementation of this interface is an entirely different question.

3 Facets

With the facet notation, a programmer expresses an individual actor as a tree of facets. Roughly speaking, a facet groups some of the actor's state with the behavior related to a particular conversation. The behavior must specify the contributions (assertions and interests) to a conversation together with reactions to the utterances of other participants. The tree structure reflects the tendency of conversations to branch from one another while accruing context. This section introduces the facet language with the code for some of the actors from the smart home simulation of the preceding section.

Figure 2 presents the code for the `Light` actor written in our Racket-based implementation. The listing includes the `spawn` expression that launches the actor (line 8). The actor engages in a single conversation concerning the presence of people in the room and the state of the light. Hence, it starts a single facet (line 2), named `light-facet`. The `start-facet` form imperatively extends the actor's facet tree with additional behavior. The new facet joins the tree as a child of the enclosing facet. The first facet started by the actor, as is the case for `light-facet`, becomes the root of the tree.

A facet keeps the actor-relevant state in *fields*. The `light-facet` facet creates a single `field`, which keeps track of the current state of the light (line 3). This `light-state` field holds values of type `Bool` and is initialized to `ON`, a constant defined elsewhere in the program. The expression `(! light-state)` accesses the current value of the field, while

`(:= light-state OFF)` updates it. The type of the `light-state` field handle itself is `(Field Bool)`, analogous to the type of a mutable reference cell. A type annotation on a field is optional; when elided, it defaults to the type of the initialization expression.

A facet interacts with the connected dataspace via *endpoints*. Endpoints come in two varieties—assertions and event handlers—corresponding to incoming and outgoing information. The `light-facet` facet comes with three endpoints:

- The first one is an assertion endpoint (line 4). It shares information about the state of the light with the dataspace. The assertion is a `struct` of type `light`. The `light` assertion carries one value, the contents of the `light-state` field accessed via `!`. The endpoint establishes an assertion in the dataspace while the facet it belongs to is active. In this case, since the actor boots with the `light-facet` facet, it initially makes such an assertion. Moreover, the facet keeps its assertion up to date. When an assertion endpoint refers to a field, the endpoint and field establish a dataflow link. The dataflow mechanism propagates field updates in order to keep the actor's assertions up-to-date with respect to its fields. Thus, every time `light-state` changes, the actor automatically withdraws the previous `light` assertion and replaces it with a version reflecting the new `light-state`.
- The other endpoints are `on` (event) handlers (lines 5-6). Each consists of two pieces:
  1. The event specification describes the nature and structure of the event. The nature of an event is expressed in terms of a keyword—`asserted` or `retracted`—corresponding to the appearance or disappearance of assertions in the dataspace. The structure of the event is expressed with a pattern, articulating a query for specific structures in the dataspace. The actor automatically issues an assertion of interest corresponding to the event handler's pattern. In the running example, the event handler wishes to know about `in-room` assertions in the dataspace; it does so with placing `?(in-room 0)` in the dataspace. An event handler's assertion of interest is initialized in the same manner as an `assert` endpoint. Thus, the actor boots with assertions of interests corresponding to its initial event handlers. Moreover, if the patterns of the event handlers reference the values of any fields, the assertions of interest are kept up to date through the same dataflow mechanism.
  2. The body part of the handler is code that executes in response to every matching event. In the running example, each event handler's body updates the facet's state via the `light-state` field. Recall that updates to this field automatically propagate to the facet's assertion via dataflow.

The `light-actor` function is effectful. When called from inside an actor `spawn` statement (line 8), or the body of a facet's event handler endpoint, it has the effect of starting a new facet with the described behavior. In our implementation, programmers may mix facet operations, field creation, and endpoint creation together with definitions and expressions. With this flexibility, any aspect of an actor's behavior may be defined inside a procedure. This can improve the readability of the code as well as facilitate re-use and sharing of code among different actor definitions.

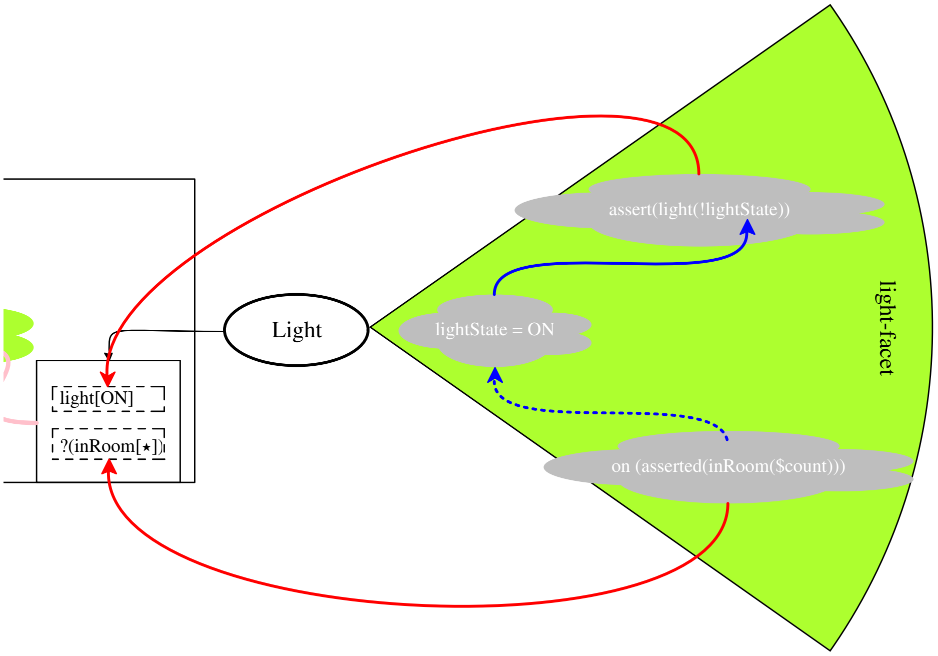

Fig. 3. Visualization of an actor’s facets and facet’s endpoints.

Figure 3 visualizes the `light-facet` facet of the `Light` actor. The facet encapsulates the `light-state` field and the two endpoints. The solid-blue arrow indicates the dataflow link between the field and the assertion endpoint; the dotted-blue arrow indicates that a value flows from the body of the on handler into the field. Finally, the red arrows show the manifestation of the endpoints in the dataspace as an assertion and an interest for the two endpoints, respectively.

Let’s equip the smart home simulation with additional features and observe how the language of facets facilitates the corresponding adaptation of the `Light` actor:

Wall Switches control the power to lights in a room. When a switch is off, the light must be off and is unable to communicate with other devices. Flipping the switch on turns the light on and allows for communication with, and control by, other devices.

Multiple Rooms complicate the simulation, too. Certain interactions are possible only among devices located within the same room.

Configuration is essential for software systems. Initially, the only devices are the control hub and the wall switches. The user may install lights and presence sensors. After installing a device, the user must assign it to a specific room via the hub.

Figure 4 shows how to adapt the implementation of the `Light` actor to this revised scenario. An unrelated difference to Figure 2 is the use of `spawn` inside of the `spawn-light-actor` function—a simplification of the presentation. This function is now called by a UI component when the user installs a light. It takes one `String`-typed argument. The argument designates to which wall switch the light is connected; that is, the

```
1  (define (spawn-light-actor [wall-switch-id : String])
2    (define my-id (generate-unique-id "light"))
3    (spawn
4      (start-facet light-facet
5        (during (wall-switch wall-switch-id ON)
6          (field [light-state ON])
7          (assert (light my-id (! light-state)))
8          (during (room-assignment my-id $room)
9            (on (asserted (in-room room 0)) (:= light-state OFF))
10           (on (retracted (in-room room 0)) (:= light-state ON))))
```

Fig. 4. Implementation of the `Light` actor, draft.

argument represents the inherent connection between elements on a power circuit. Once the function is called, it generates a unique ID for the light (line 2), akin to a manufacturer serial number. After that, it spawns the light actor with one initial facet (lines 3-4).

Equipped with a basic understanding of this function, let us look at the revised `light-facet` facet. Recall that the actor should *not* engage in any behavior unless the connected switch is flipped on. Once it is on, the actor participates in conversation(s) until the switch is turned off. A basic facet-oriented actor would start a facet in response to the first kind of event and shut it down when the second kind occurs. This pattern is so common that the facet language comes with a `during` expression (line 5), which combines an assertion pattern that can be used with either `asserted` or `retracted`.[2]

Here the facet operates while the assertion of `(wall-switch wall-switch-id ON)` exists in the dataspace—representing the interval of time during which the room's switch is in the on position. As the switch is flipped into the on position, the facet announces its existence and (`ON`) state using a field and assertion endpoint as before (lines 6-7). This `light` assertion allows the `Hub` actor to detect the installation of a new device. If this is the first time the `Hub` actor encounters this device ID, it prompts the user to assign it to a room. Once the assignment is complete, the `Hub` actor deposits a `room-assignment` assertion (line 8) to inform the `Light` actor. The subpattern `$room` binds the corresponding portion of the `room-assignment` assertion to the name `room`. Knowing what room it is in allows the `Light` actor to converse with any presence sensor in the same room, turning on and off accordingly (lines 9-10).

Multifaceted Actors. The revised implementation of the `Light` actor starts several facets. Eventually it may run three concurrent facets: `light-facet` and one per `during` expression. Each of these facets corresponds to a particular context within an ongoing conversation. Furthermore, the facets are nested, just like actor conversations. When one facet starts another, the first is the parent to the second. Stopping a facet (via `stop`) means terminating it and all of its children. In the context of the `Light` actor, the `wall-switch` facet may shut down when the switch is flipped off; in this case, shutting down this facet also shuts down the nested facets associated with `during` expressions. A `stop` form may optionally specify some continuation behavior, as in `(stop` *orig* `(start-facet` *cont* `...))`. In that case, the *cont* facet takes the place of *orig* in the tree. Continuation facets address the need to transition from one task to another.

[2] In Racket, `during` is just a notational definition (macro); see Section 3.1.

```
1  (define-type LightDict (Dictionary String Light))
2
3  (define (launch-hub)
4   (spawn
5    (start-facet hub
6     (field [lights : LightDict (create-dictionary)])
7     (during (light $id _)
8       (on start
9        (match (dict-ref (! lights) id #false)
10         [(light room _) (control-light lights id room)]
11         [#false         (get-room-from-user id)]))))))
12
13 ;; auxiliary functions:
14 (define (get-room-from-user [id : String])
15   (start-facet get-room
16     (define room ... interact with user to assign a room ...)
17     (stop get-room (control-light lights id room))))
18
19 (define (control-light [lights : (Field LightDict)] [id : String]
20                        [room : String])
21   (start-facet control
22     (assert (room-assignment id room))
23     (on (asserted (light id $o?))
24       (set-light-state lights id room o?))
25     (on stop (set-light-state id room OFF))))
26
27 (define (set-light-state [lights : (Field LightDict)] [id : String]
28                          [room : String] [on? : Bool])
29   (:= lights (dict-set (! lights) id (light room on?))))))))
```

Fig. 5. Implementation of the `Hub` actor.

Wider Facet Trees. While the facets in the `Light` actor are nested in a linear fashion, bushy trees are equally common. Consider the `Hub` actor. A user interacts with the `Hub` to configure each light. Once configured, the `Hub` both monitors and controls these lights. To accomplish this task, the actor starts a facet for each separate conversation with the various `Light` actors.

Figure 5 shows a portion of the actor's implementation. The actor starts a facet named `hub` (line 5), which creates one field (`lights`, line 6) for storing information about the lights in the system. The field is a dictionary. For each registered light, this dictionary associates the light's ID with a `light` struct that records its room and its known state. The auxiliary function `set-light-state` (lines 27–29) takes a reference to the `lights` field as an argument, of type `(Field LightDict)`, and updates the dictionary for a particular light.

Using `during`, the actor converses with each individual light. The during reacts in response to new `light` assertions.[3] Such assertions can mean one of two things: the installation of a light or the restoration of power to an already installed light. In the first case, `lights` does not contain an entry with that ID; in the second case, it does. Each of these situations calls for different behavior, but no matter what, the reaction should happen only when the facet starts up.

[3] The "don't care" pattern _ means the `during` reacts to the initial appearance of a `light` assertion with a particular ID, but not to subsequent updates to the on/off state of that light.

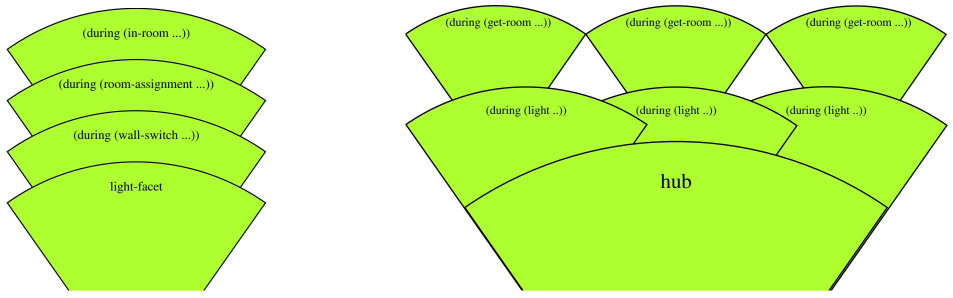

Fig. 6. Visualization of the `Light` and `Hub` actors' facet trees.

To express this idea directly, the language enables facets to react to the `start` and `stop` events, respectively. That is, instead of a reaction to a dataspace assertion, a facet's endpoints may specify a start or stop specific reaction. Here the `during` facet uses `start` to decide how to act (line 8). In the handler (lines 9–11), it uses `lights` to look up what it knows about the light. In the first case and `match` branch, the facet retrieves the light's room assignment and starts another facet to control it via `control-light` (lines 19–25). In the second case, the actor calls `get-room-from-user`, which engages in yet another dialog with the user to assign the new light to a room. This nested facet `stop`s itself, with some explicit continuation behavior: a call to `control-light`.

The `control-light` function starts a facet to assert the light's `room-assignment` and to react to changes in its state. When the state changes, the facet updates `lights`. Finally, when the facet stops—due to termination of the outer `during`—it records the light as off. Stop handlers, like exception handlers, enable separation between the decision to disengage in behavior and the definition of how to clean up in that event. In this case, the decision is implicit in the `during` on line 7's `retracted` event handler. Meanwhile, the code for cleaning up is in the `control-light` function (line 24). Crucially, the need to clean up only arises after an initial provisioning step, so it would not make sense to have the functionality at the same level as the `during`. In the case of a crash, an actor does not run any `stop` event handlers. Since the state of the actor could be inconsistent, with fields partially updated, orderly shutdown may not be possible.

Figure 6 visualizes the facets of the `Light` and `Hub` actors. In the case of the `Light` actor, the tree is deeply nested, as the actor engages in new behaviors as it accumulates conversational context. By contrast, the `Hub` actor's facet tree exhibits breadth, as it starts a new facet in reaction to each installed light. In turn, each of those facets have sub-trees either to interact with the user or to control the light. Each sub-tree of the `hub` facet is independent of the others, evolving based on its interactions with its controlled light and the user without needing to know the current state (or existence) of the others.

3.1 Derived forms

Figure 7 displays derived forms for common patterns. One exceedingly common pattern is for a facet to be constructed in response to the appearance of an assertion and torn down in response to its disappearance. The `Light` and `Hub` actors both fall into this category.

```
(during pattern during-body ...)
  ≝
(on (asserted pattern)
    (start-facet theBody
      (on (retracted pattern′)
          (stop theBody))
      during-body ...))
```

```
(during/spawn pattern during-body ...)
  ≝
(on (asserted pattern)
    (spawn (start-facet theBody
      (on (retracted pattern′)
          (stop theBody))
      during-body ...)))
```

```
(define/query-value id expr0 pattern expr)
  ≝
(field [id expr0])
(on (asserted pattern)
    (:= id expr))
(on (retracted pattern)
    (:= id expr0))
```

```
(define/query-set id pattern expr)
  ≝
(field [id (Set τ) empty-set])
(on (asserted pattern)
    (:= id (set-add (! id) expr)))
(on (retracted pattern)
    (:= id (set-remove (! id) expr)))
```

Fig. 7. Derived forms.

We abstract this pattern into the `during` form. Each time the `asserted` event fires, any pattern variables bound in *pattern* are instantiated to yield a concrete assertion, *pattern*′. The `(on (retracted ...))` endpoint monitors this assertion in the new facet.

Figure 7 also shows a related derived form, `during/spawn`. In contrast to `during`, this form does not create a facet within the actor but spawns an actor in the dataspace. The critical difference concerns failure. While a failure in `during` tears down the current actor, a failure in `during/spawn` terminates the separate actor but leaves the spawning one alone.

Both `during` and `during/spawn` allow the programmer to directly express *matching of supply to demand*. Assertions matched by the *pattern* are interpreted as demand; the *during-body* constitutes the supply. As demand increases via new matching assertions in the dataspace, the supply expressions are executed to match. As demand *decreases* again due to the withdrawal of assertions, `(on (retracted ...))` endpoints automatically terminate the facet or actor in response. Resources are allocated via `(on start ...)` clauses and released in `(on stop ...)` clauses in response to changing needs.

Figure 7 defines a second family of derived forms concerning the integration of newly arrived assertions into local fields. While actors often just deal with singleton assertions, other scenarios call for a set, a hash table, or even an aggregate summary of several assertions. To support this idiom, the facet language provides derived constructs called *queries*. Queries define and subsequently update fields, making their results available for use in neighboring facet endpoints.

One such form, `define/query-value`, is useful for local tracking of an assertion kind in which there may be zero or one instance in the dataspace. For example, we could extend the smart home program with persistence, so that the user's assignments of devices to rooms are saved between runs of the program. A separate actor would handle persisting such configurations to the file system or other location and loading them when the program starts. Then, when the `Hub` actor boots, it needs to operate based on a reloaded configuration or start with a default fresh one. It may use

```
(define/query-value config DC (configuration $lights ...)
    (configuration lights ...))
```

to create a field that defaults to `DC` (short for default configuration) value unless there is a `configuration` assertion, in which case it uses the value of that assertion.

The second such form, `define/query-set`, collects information from relevant assertions in a set.[4] A user-interface component for the smart home may use it to show the user all of the lights currently installed in the system:

```
(define/query-set lights (light $id _) id)
```

The query scheme directly generalizes to structures such as hash tables, etc., and also allows *aggregations* analogous to SQL's `COUNT(*)` and `GROUP BY`.

3.2 The case for facets

Returning to the dining concurrency researchers example, facets support a conversational style in several ways:

- Engaging in a new conversation is as simple as starting a facet. Likewise, stopping a facet disengages from a conversation.
- The `during` form directly connects context to behavior.
- The tree of facets mirrors the way (sub)conversations branch from existing conversations.

Defining actors using facets provides additional advantages over basic functions and objects, as well. Appendix A provides an implementation of the light actor using a procedural notation as a concrete basis of comparison, demonstrating and explaining these benefits.

Generally put, the facet notation combines and extends a number of concepts from other languages and empowers programmers to use them in a synergistic manner. For example, end-point programming may remind the reader of the event-handling and reactive programming style of Esterel (Berry & Gonthier, 1992) and CRIME (Mostinckx *et al.*, 2008). Dataflow exists in many forms (Demetrescu *et al.*, 2011). Updates to behavior have been around as long as the Actor model (Hewitt *et al.*, 1973). The `during` behavior is the clearest example of the synergistic use of these features. It precisely describes a temporal engagement with the dataspace; sets up and tears down local behavior as needed; and simultaneously takes into account the accumulated conversational context.

4 Static checks: Types and behavior

Programming actors requires the development of protocols. A protocol coordinates the flow of data among actors in a conversation. Designing the program requires reasoning about protocols while writing code. An actor language ought to assist programmers with this reasoning process and ideally with checking it statically.

The language of facets comes with a structural type system and a behavioral one. The first is somewhat conventional (Section 4.1); also see Caldwell *et al.* (2020)'s work on

[4] The type of the set's elements is inferred from the provided *expr*.

structural types for dataspaces. The second enables programmers to specify an actor's dynamic behavior (Section 4.2) and perform some static checking (Section 4.4). The bridge between the two is an effect-type system (Section 4.3). Section 4.5 applies this checking to an example specification from the smart home.

4.1 Basic types

The starting point is a structural type system. Base types describe data such as `Int`, `Bool`, etc. A struct-type definition such as `(struct light ([on? : Bool]))` introduces the type constructor `LightT`.[5] Thus, `(light #true)` has type `(LightT Bool)`.

Our previous work on structural types (Caldwell *et al.*, 2020) for purely functional dataspace actors validates that it prevents mistakes involving the shape of assertions. That is, type checking incorporates knowledge of how a dataspace routes assertions among actors and uses the types to confirm that assertions cannot upset routing. It is straightforward to adapt this system and its proof of soundness to facet-oriented dataspace actors.

4.2 Specification

Like for structural properties of routing assertions, programmers need a language for behavioral—that is, temporal—properties, too. Like structural types give a syntax and semantics to claims about properties of assertions, behavioral types should allow the programmer to make claims concerning the behavior of actors in a dataspace. We employ a form of linear temporal logic (LTL) for this purpose, though of course, its basic propositions must somehow involve the assertions that actors deposit into and withdraw from dataspaces. In addition to introducing this language, this section also informally explains what it means for a facet-based actor program to live up to such a specification.

LTL has a long history as a specification language for concurrent programs and the basis for mechanized checks (Manna & Pnueli, 1991; Holzmann, 1997; Pnueli, 1977). When compared to other temporal logics, such as the branching-time computation tree logic (CTL), LTL is typically considered more intuitive and at least as amenable to efficient model checking (Vardi, 2001; Rozier, 2011). Moreover, compared to typical protocol description languages such as choreographies (Carbone & Montesi, 2013) or session types (Honda *et al.*, 2008), LTL allows the focus to be on *what* is communicated rather than *who* does the communicating, suiting the anonymous style of dataspace programming.

The base LTL predicates are types that describe assertions in the dataspace. For example, the formula `(LightT Bool)` holds at any moment iff some actor has deposited an assertion `(light #true)` or `(light #false)` in the dataspace. Specifications may also use negation (`Not`), and `(Not (LightT Bool))` means no assertion of this structure type is in the dataspace.

Other connectives are conjunction (`And`), disjunction (`Or`), and logical implication (`Implies`). With temporal connectives, a programmer can specify the evolution of dataspace programs. The proposition `(Always (LightT Bool))` holds if the dataspace

[5] The final "T" is a convention for such type constructors. The `struct` form defines the plain `Light` type name to be an alias for the type constructor fully instantiated with the default type parameters, such as `(LightT Bool)`.

contains a `light` assertion at every moment during execution, regardless of which actor has deposited it there. Meanwhile,

```
(Until (LightT Bool) (Not WallSwitchOn))
```

says that eventually there will not be a (`wall-switch-on`) assertion in the dataspace, but until then there is a `light` assertion. The formula

```
(Eventually (LightT Bool))
```

describes an execution state with an assertion matching (`LightT Bool`) in the dataspace *or* the execution arrives at such a state within a finite number of steps.

A dataspace program satisfies a specification if its dataspace evolves according to the interpretation of the LTL formula. Thus, a program lives up to the just-mentioned formula if, at some point during execution, an actor makes a matching `light` assertion. In particular, any dataspace program that includes the `light-facet` actor from the preceding section satisfies this formula, because it makes a `light` assertion as it boots.

Consider another proposition:

```
(Always
   (Implies
      (LightT Bool)
      (Until
         (LightT Bool)
         (Not WallSwitchOn))))
```

It is always true that, if a `light` assertion is in the dataspace, eventually there will not be a `wall-switch-on` assertion but until that time a `light` assertion will remain in the dataspace.

It is the strong version of `Until`, requiring that `(Not WallSwitchOn)` eventually be true.

Take the dataspace program that runs the two actors on the left:

actor 1

```
(start light-facet
  (field [state ON])
  (assert (light (! state)))
  (on (retracted (wall-switch-on))
    (:= light OFF)))
```

actor 2

```
(start wall-switch
  (assert (wall-switch-on))
  (on (asserted (light _))
    (stop wall-switch)))
```

dataspace evolution

t =	assertions
0	∅
1	{`(light ON)`, `(wall-switch-on)`}
2	{`(light ON)`}
3	{`(light OFF)`}

The evolution of the dataspace is on the right. It shows which assertions are in the dataspace for the first four time steps, starting with the boot state.

After booting, actor 1 makes a `light` assertion and actor 2 adds a (`wall-switch-on`) to the dataspace. Hence, the dataspace configuration at that point matches the antecedent of the implication. Consequently, the `Until` property must also describe that state so that the latter models the complete formula. Due to actor 1's assertion, actor 2 stops its only facet, leading to the withdrawal of the (`wall-switch-on`) assertion. Since the `light` assertion remains active until that point, the implication still holds. Furthermore, actor 1 notices the disappearance of the (`wall-switch-on`), and in response, replaces its own assertion the dataspace. As a result, the next configurations also match the antecedent of the implication as does the until property.

While the notion of satisfaction is intuitive, the eventual goal is to mechanize this type-checking of specifications. In this regard, the sample explanations are also suggestive.

Actor and Facet Descriptors

Term	Type
`(spawn t ...)`	`(Spawn T ...)`
`(start-facet x t ...)`	`(StartFacet x T ...)`
`(stop x t ...)`	`(Stop x T ...)`
`(assert t)`	`(Assert T)`
`(on evt t ...)`	`(On Evt T ...)`

Event Descriptors

Term	Type
`(asserted p)`	`(Asserted P)`
`(retracted p)`	`(Retracted P)`
`start`	`Start`
`stop`	`Stop`

Also, `$x:T` and `_` have types `$T` and `_`, respectively.

Fig. 8. Effect types.

What is needed is an abstract description of the behavior of the actors and their interaction with dataspaces via assertions. And again, just like for structural type checking, it is the *shape* of exchanges that can serve in this role, not the precise assertions.

4.3 Effect types for facets

The end of the preceding section almost dictates the third step. Historically, type systems research uses effect-type systems for descriptions of program behavior (Lucassen & Gifford, 1988). In the setting of facet-oriented actors, the to-be-observed effects are caused by actors overall, facets, and endpoints. Hence, each corresponding term-level construct comes with a type-level construct for describing the type of its effects. See Figure 8 for the actual notation.

Let us illustrate the synthesis of effect types with the code for spawning the simplistic `Light` actor from Section 3 (without rooms and power switches). Ignoring fields and expressions, the code describes a single actor and a single facet nested within this actor. The facet itself comes with three end points: a plain assertion endpoint, an on-asserted one and an on-retracted one. The behavioral type mirrors this description, including the ordering of the endpoints within the facet:

```
(spawn                                      (Spawn
  (start-facet light-facet                    (StartFacet light-facet
    (field [light-state ON])                    (Assert (LightT Bool))
    (assert (light (! light-state)))    ::      (On (Asserted (InRoomT Int)))
    (on (asserted (in-room 0))                  (On (Retracted (InRoomT Int)))))
        (:= light-state OFF))
    (on (retracted (in-room 0))
        (:= light-state ON))))
```

Here is how to read the type in terms of the actor's communication behavior:

- it makes an assertion of type `(LightT Bool)`;
- it expresses interest to assertions of type `(InRoomT Int)`;
- it reacts to their appearance and disappearance; and
- its reactions do not change its communication behavior.

For this last point, note that the bodies of the `On` types are empty.

From this perspective, a behavioral type is a *simplified* actor. This simplified actor communicates via *types* of assertions rather than value assertions. By taking the behavior type of each actor in a program—say, the smart home program—we get a communication-only program, that is, a collection of simplified actors and their types of assertion exchanges with dataspaces. Accordingly, the number of possible behaviors is much lower—and that is precisely what enables a mechanical checking of (some) specifications.

4.4 Checking specifications mechanically

The transition of possible interactions from concrete value assertions to types of assertions enable specification checking. At a rather high level, the checker infers effect types from a program and compiles them to the language of a model checker. The compilation encodes effect types as simplified actors. This model-checking program is combined with the programmer's LTL formula. If the model-checking attempt fails, the error is reported in terms of a trace of these simplified actor programs.

The theory behind this idea is presented in Sections 5 and 7. Section 8 verifies why this approach works. The implementation is sketched in Section 9. Section 10 evaluates its ability to check behavioral properties of realistic programs. The following subsection illustrates the working of the model checker with an example.

4.5 Revisiting the smart-home example

To demonstrate the power of the specification language, let us revisit the smart home example from Section 3. The code presented in Figure 4 is buggy, and attempting to check it against a natural specification exposes the bug, Here is the specification of the expected behavior of the lights:

```
(Always (And (Implies (InRoomT Room NonZero)
                      (Implies WallSwitchOn (Eventually LightOn)))
             (Implies (InRoomT Room Zero)
                      (Eventually (Not LightOn)))))
```

This specification describes the expected outcomes of interactions between the presence sensor, light, and light-switch actors. It states that the light turns on and off in response to the sensor actor's `InRoomT` assertions, as long as it is powered (`WallSwitchOn`).

When analyzed with respect to this specification, our implementation of LTL checking reports the error and provides a counterexample in terms of the involved actors and their assertions:

```
...
Process Sensor RETRACTS RoomEmpty
...
Process Switch ASSERTS WallSwitchOn
...
```

If a presence sensor is installed and configured *after* the light in the same room is turned on even though it is empty, the light stays on due to the sensor's initial reading.

Based on this trace, we can identify the active assertions and thus the active facets of the `Light` actor. This assessment provides a starting point for determining where to locate the bug. A close look reveals that the `light-facet` facet in Figure 4 reacts only to the retraction of `(in-room room 0)` to determine that the room is occupied. If the

```
(define (spawn-light wall-switch-id)
  (define my-id (generate-unique-id "light"))
  (spawn
    (start-facet light-facet
      (during (wall-switch wall-switch-id ON)
        (field [light-state ON])
        (assert (light my-id (! light-state)))
        (during (room-assignment my-id $room)
          (on (asserted (in-room room 0)) (:= light-state OFF))
          (on (retracted (in-room room 0)) (:= light-state ON)))
        (start-facet init
          (on (asserted (in-room room $n))
              (:= light-state (not (zero? n)))
              (stop init)))

                                              ))))
```

Fig. 9. Implementation of the `Light` actor, fixed (see Figure 4).

room is occupied when the light regains power, no such retraction will take place. To fix this problem, we can amend the facet to query the state of `in-room` assertions on startup. The boxed code in Figure 9 is all that is needed to make the actor satisfy its LTL specification.

5 Facets: The semantics

A proper language design should come with a rigorous blueprint, especially if its rationale includes claims about the reasoning power of its type system. The presented language consists of two pieces: the facet notation and its connection to dataspaces. The former deserves a formal semantics; the latter has been presented thoroughly elsewhere (Garnock-Jones & Felleisen, 2016; Caldwell *et al.*, 2020), but Section 7.1 provides an abbreviated treatment so that the meta-theorems can be stated in a self-contained manner.

This section presents the formal model of a *single* facet-based actor (Sections 5.1 and 5.2). It describes how a facet-based actor updates its state in response to an event: the evolution of the facet tree; the invocation of event handlers; and field maintenance.

5.1 Formal syntax

Figure 10 defines the abstract syntax of facet-oriented actors. Statements Pr imperatively update the actor's state in one of several ways:

- `start` fn $\overrightarrow{e}$ ($\overrightarrow{\text{D Pr}}$)—starting a facet with name fn, assertions $\overrightarrow{e}$, and event handlers ($\overrightarrow{\text{D Pr}}$);
- `stop` fn Pr—terminating a facet, along with all of its children, and engaging in some continuation behavior Pr;
- `field` x = e `in` Pr—creating a mutable field for Pr;
- x := e—assigning a new value to a field; and
- `spawn` Pr—spawning an additional actor.

$$\begin{array}{rcl}
\mathrm{Pr} \in \textbf{Prog} & = & \texttt{start}\ \mathrm{fn}\ \overrightarrow{\mathrm{e}}\ (\overrightarrow{\mathrm{D}\ \mathrm{Pr}}) \\
& | & \texttt{stop}\ \mathrm{fn}\ \mathrm{Pr} \\
& | & \texttt{spawn}\ \mathrm{Pr} \\
& | & \texttt{field}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr} \\
& | & \mathrm{x} := \mathrm{e} \\
& | & \texttt{let}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr} \\
& | & \mathrm{Pr};\ \mathrm{Pr} \\
& | & \texttt{skip} \\
\mathrm{D} \in \textbf{EvtDsc} & = & \texttt{asserted}\ \mathrm{e} \\
& | & \texttt{retracted}\ \mathrm{e} \\
& | & \texttt{start} \\
& | & \texttt{stop}
\end{array}
\qquad
\begin{array}{rcl}
\mathrm{e} \in \textbf{Expr} & = & b \\
& | & \mathrm{x} \\
& | & !\,\mathrm{x} \\
& | & p(\overrightarrow{\mathrm{e}}) \\
& | & \texttt{?}\ \mathrm{e} \\
& | & m(\overrightarrow{\mathrm{e}}) \\
& | & \star \\
& | & \mathrm{x} : \tau \\
\mathrm{fn} \in \textbf{FName} & = & \text{facet names} \\
\mathrm{x} \in \textbf{Var} & = & \text{variables} \\
p \in \textbf{PrimOp} & = & \text{primitive operations} \\
m \in \textbf{MsgCtor} & = & \text{message labels} \\
b \in \textbf{BasicVal} & = & \text{basic values: strings, etc.}
\end{array}$$

Fig. 10. Program syntax.

Statements may introduce local, lexically scoped variables with `let` x = e `in` Pr. Otherwise, they are just like statements in other programming languages that compose sequentially (Pr; Pr). For technical reasons, the grammar includes no-op (`skip`).

The set of expressions comprises

- b—basic values;
- $p(\overrightarrow{\mathrm{e}})$—invocations of potentially partial primitive operations;
- x—references to variable names;
- ! x—field access;
- `?` e—assertion of interest in e;
- $m(\overrightarrow{\mathrm{e}})$—tuple with tag m;
- $\star$—wildcard assertion patterns/templates; and
- $\mathrm{x} : \tau$—binding patterns.

A vector of assertion templates ($\overrightarrow{\mathrm{e}}$) defines the assertions made by a facet. Templates describe families of assertions built with labels (m), interests (`?`), wildcards ($\star$), and expressions (e).

Event handlers consist of an event description (D) and a body (Pr). Events describe occurrences that are either internal to the actor (`start`, `stop`) or particular assertions in the dataspace (`asserted` e, `retracted` e). In the latter case, the description includes a pattern expression e defining the relevant assertions. In order to construct patterns, expressions additionally include match-anything wildcards ($\star$) and binding variables ($\mathrm{x} : \tau$).

In comparison to the implemented syntax, the model uses some obvious short-hands and some less obvious simplifications:

- The assertion endpoints of a facet are all declared together.
- The event handlers come as a sequence of description/handler body pairs.
- The fields shared by a facet's endpoints are declared outside the facet.
- The assertion labels replace `struct` names.
- Binding variables in patterns require a type annotation.

$$
\begin{array}{rcl}
\mathrm{M} \in \textbf{Machine} & = & \langle \mathrm{FT};\ \overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma \rangle \\
 & | & \texttt{error} \\
\mathrm{FT} \in \textbf{FctTree} & = & \varepsilon \\
 & | & \mathrm{fn}[\vec{\mathrm{e}}\,\overrightarrow{(\mathrm{D}\ \mathrm{Pr})}].\overrightarrow{\mathrm{FT}} \\
\mathrm{C} \in \textbf{Context} & = & \square \\
 & | & \mathrm{fn}[\vec{\mathrm{e}}\,\overrightarrow{(\mathrm{D}\ \mathrm{Pr})}].\overrightarrow{\mathrm{FT}} \cdot \mathrm{C} \cdot \overrightarrow{\mathrm{FT}} \\
\mathrm{I} \in \textbf{Instr} & = & \texttt{start fn}\ \vec{\mathrm{e}}\ \overrightarrow{(\mathrm{D}\ \mathrm{Pr})}\ @\ \mathrm{fid} \\
 & | & \texttt{stop fn} \\
 & | & \texttt{spawn}\ \mathrm{Pr} \\
\mathrm{PS} \in \textbf{PScript} & = & (\mathrm{fid},\ \mathrm{Pr}) \\
\mathrm{fid} \in \textbf{FacetID} & = & \langle \overrightarrow{\mathrm{fn}} \rangle \\
\mathrm{fn} \in \textbf{FName} & = & \text{facet names} \\
\sigma \in \textbf{Store} & = & \mathrm{x} \xrightarrow{fin} \mathrm{v}
\end{array}
\qquad
\begin{array}{rcl}
\mathrm{u} \in \textbf{Assertion} & = & b \\
 & | & m(\vec{\mathrm{u}}) \\
 & | & ?\ \mathrm{u} \\
\pi \in \textbf{ASet} & = & \mathscr{P}(\textbf{Assertion}) \\
\Delta \in \textbf{Patch} & = & \pi^{+}/\pi^{-} \\
 & & \text{where } \pi^{+} \cap \pi^{-} = \emptyset \\
\mathrm{Evt} \in \textbf{Event} & = & \Delta \\
 & | & \texttt{start} \\
 & | & \texttt{stop} \\
\mathrm{v} \in \textbf{Val} & = & b \\
 & | & ?\ \mathrm{v} \\
 & | & m(\vec{\mathrm{v}}) \\
 & | & \star \\
 & | & \mathrm{x} : \tau \\
l \in \textbf{Label} & = & \bullet \\
 & | & \Delta \\
 & | & \overline{\mathrm{Pr}}
\end{array}
$$

Fig. 11. Evaluation syntax.

Here is the basic `Light` actor from Figure 2 in the model's syntax:

```
field state = ON in
  start light-facet
    light(!state)
    (asserted in-room(count : Int)
     state := if(equal?(count, 0), OFF, ON))
```

5.2 Semantics

Since individual actors consume events and produce assertions, we choose to specify the semantics of facet-oriented actors via a labeled transition system. Each transition acts on a machine state and constructs the next one. The transition labels l describe either inputs in the form of event patches (Δ) or outputs in the form of spawned actors ($\overline{\mathrm{Pr}}$). An unlabeled transition relation ($\bullet$), represents actor-internal work. Patches Δ comprise two disjoint assertions sets π, representing added and removed assertions. Assertions u range over basic values, labeled tuples, and assertions of interest.

The formulation of machines requires syntax for describing the states in addition to the syntax of programs. Figure 11 defines this syntax and the machine states. The `error` state represents an actor that has crashed due to an unhandled error.

An actor machine state consists of five components: $\langle \mathrm{FT};\ \overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma \rangle$. The components track different kinds of state and knowledge:

- *the activity state of facets*
  The machine must express which facets are currently active as well as the parent-child relationship among them. This aspect takes the form of a tree: FT. A node

$$\begin{array}{llcl}
\multicolumn{4}{l}{\langle\, l\text{-}c[I_0]\ ;\ \cdot;\ \cdot;\ \emptyset;\ \{\ \texttt{state} \mapsto \texttt{ON}\ \}\ \rangle} \\
\text{where} & l\text{-}c & = & \texttt{light-facet} \\
 & I_0 & = & [\texttt{light(!state)} \\
 & & & \ (\texttt{asserted in-room(count : Int)} \\
 & & & \ \texttt{state := if(equal?(count, 0), OFF, ON))}].\varepsilon \\
\hline
\end{array}$$

$$\xrightarrow{\{\texttt{in-room(0)}\}/\emptyset} \qquad \text{(by inject)}$$

$$\begin{array}{llcl}
\multicolumn{4}{l}{\langle\, l\text{-}c[I_1]\ ;\ \cdot;\ \mathrm{PS};\ \{\texttt{in-room(0)}\};\ \{\texttt{state} \mapsto \texttt{ON}\}\ \rangle} \\
\text{where} & \mathrm{PS} & = & (\langle l\text{-}c\rangle,\ \texttt{state := if(equal?(0, 0), OFF, ON)}) \\
 & I_1 & = & [\texttt{light(!state)} \\
 & & & \ (\texttt{asserted in-room(count : Int)} \\
 & & & \ \texttt{state := if(equal?(count, 0), OFF, ON))}].\varepsilon
\end{array}$$

$$\xrightarrow{\bullet} \qquad \text{(by transfer)}$$

$$\begin{array}{llcl}
\multicolumn{4}{l}{\langle\, l\text{-}c[I_2]\ ;\ \cdot;\ \cdot;\ \{\texttt{in-room(0)}\};\ \{\texttt{state} \mapsto \texttt{OFF}\}\ \rangle} \\
\text{where} & I_2 & = & [\ \texttt{light(!state)} \\
 & & & \ (\texttt{asserted in-room(count : Int)} \\
 & & & \ \texttt{state := if(equal?(count, 0), OFF, ON))}].\varepsilon
\end{array}$$

Fig. 12. The `light-facet` facet as an initial machine state and its transitions.

$\mathrm{fn}[\overrightarrow{e}\ (\overrightarrow{D\ \ Pr})].\overrightarrow{FT}$ in this tree describes an active facet in terms of its name, its assertion and event handler endpoints, and its children, while the empty tree takes the form ϵ.

- *the instruction stream within the active facets*
  A machine state identifies a sequence of ready-to-perform instructions I, and it also maintains a queue of pending scripts PS. A pending script (fid, Pr) is a sequence of statements to be performed and pertinent context information. The context is a facet ID $\langle \overrightarrow{\mathrm{fn}} \rangle$ identifying the facet from which the script originated. A facet ID is a sequence of facet names forming the path from the root of the tree to that particular facet. Instructions I can `start` a new facet at a designated location in the tree; `stop` a running one; and `spawn` an actor.
- *the information known to the actor*
  Specifically, the store σ holds the contents of the fields, while the set of assertions π keeps track of the actor's current knowledge of assertions in the dataspace. Newly booted facets receive these assertions as an event.[6]

Finally, we need some miscellaneous pieces of syntax. An Evt is either a patch or facet `start`/`stop`. Evaluation reduces expressions to values v.

Figure 12—above the horizontal line—shows the initial machine state M for the `light-facet` facet from above. The rest of the figure displays the first two transitions of the machine.

Transition Relations. Figure 13 defines the transition relation on machine states that describes the behavior of a single facet-based actor in response to incoming events. For the

[6] Due to the particulars of dataspace routing (Section 7), the actor will not receive another event from the dataspace describing these assertions until they disappear and subsequently reappear. That is, the dataspace tracks the set of assertions known to an *actor,* not facet. A new facet of the actor may include an event handler for which an already-received assertion is relevant.

$$\langle \mathrm{FT};\ \cdot;\ \cdot;\ \pi;\ \sigma\rangle \xrightarrow{\ \Delta\ } \langle \mathrm{FT};\ \cdot;\ \overrightarrow{\mathrm{PS}};\ \pi';\ \sigma\rangle \qquad \textsf{(inject)}$$

where

$$\begin{aligned} \pi' &= \pi \oplus \Delta \\ \overrightarrow{\mathrm{PS}} &= \mathit{dispatch}(\mathrm{FT},\ \Delta,\ \pi,\ \pi',\ \sigma,\ \langle\cdot\rangle) \end{aligned}$$

$$\langle \mathrm{FT};\ \cdot;\ (\mathrm{fid},\ \mathrm{Pr})\cdot\overrightarrow{\mathrm{PS}'};\ \pi;\ \sigma\rangle \xrightarrow{\ \bullet\ } \mathrm{M}' \qquad \textsf{(transfer)}$$

where

$$\mathrm{M}' = \begin{cases} \langle \mathrm{FT};\ (\mathrm{fid},\ \overrightarrow{\mathrm{I}});\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma'\rangle & \text{if } \overrightarrow{\mathrm{I}}, \sigma' = \mathit{p\text{-}e}(\mathrm{Pr},\ \sigma,\ \mathrm{fid}) \\ \mathtt{error} & \text{otherwise} \end{cases}$$

$$\langle \mathrm{FT};\ \mathtt{start}\ \mathrm{fn}\ \overrightarrow{\mathrm{e}}\ \overrightarrow{(\mathrm{D}\ \mathrm{Pr})}\ @\ \mathrm{fid}\cdot\overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma\rangle \xrightarrow{\ \bullet\ } \langle \mathrm{FT}';\ \overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}}\cdot\overrightarrow{\mathrm{PS}_{\mathit{start}}}\cdot\overrightarrow{\mathrm{PS}_{\mathit{boot}}}\cdot\overrightarrow{\mathrm{PS}_{\mathit{stop}}};\ \pi;\ \sigma\rangle \qquad \textsf{(start)}$$

where

$$\begin{aligned} &\mathrm{fn}_{\mathit{new}}\ \text{ fresh in }\ \mathrm{FT} \\ \mathrm{FT}_{\mathit{new}} &= \mathrm{fn}_{\mathit{new}}[\overrightarrow{\mathrm{e}}\ \overrightarrow{(\mathrm{D}\,\mathrm{Pr}[\mathrm{fn} \mapsto \mathrm{fn}_{\mathit{new}}])}].\varepsilon \\ \overrightarrow{\mathrm{PS}_{\mathit{start}}} &= \mathit{dispatch}(\mathrm{FT}_{\mathit{new}},\ \mathtt{start},\ \emptyset,\ \emptyset,\ \sigma,\ \mathrm{fid}) \\ \overrightarrow{\mathrm{PS}_{\mathit{boot}}} &= \mathit{dispatch}(\mathrm{FT}_{\mathit{new}},\ \pi/\emptyset,\ \emptyset,\ \pi,\ \sigma,\ \mathrm{fid}) \\ \mathrm{FT}' &= \begin{cases} \mathrm{C}[\mathrm{FT}_{\mathit{new}}] & \text{if } \mathit{locate}(\mathrm{FT},\ \mathrm{fid}) = \mathrm{C} \\ \mathrm{FT} & \text{otherwise} \end{cases} \\ \overrightarrow{\mathrm{PS}_{\mathit{stop}}} &= \begin{cases} \cdot & \text{if } \mathit{locate}(\mathrm{FT},\ \mathrm{fid}) \text{ defined} \\ \mathit{dispatch}(\mathrm{FT}_{\mathit{new}},\ \mathtt{stop},\ \pi,\ \pi,\ \sigma,\ \mathrm{fid}) & \text{otherwise} \end{cases} \end{aligned}$$

$$\langle \mathrm{FT};\ \mathtt{stop}\ \mathrm{fn}\cdot\overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma\rangle \xrightarrow{\ \bullet\ } \langle \mathrm{FT}';\ \overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}}\cdot\overrightarrow{\mathrm{PS}_{\mathit{stop}}};\ \pi;\ \sigma\rangle \qquad \textsf{(stop)}$$

where

$$\begin{aligned} \mathrm{FT}' &= \begin{cases} \mathrm{C}[\varepsilon] & \text{if } \mathrm{FT} = \mathrm{C}[\mathrm{fn}[\ldots].\overrightarrow{\mathrm{FT}}] \\ \mathrm{FT} & \text{otherwise} \end{cases} \\ \mathrm{fid} &= \mathit{facet\text{-}context}(\mathrm{C}) \\ \overrightarrow{\mathrm{PS}_{\mathit{stop}}} &= \begin{cases} \mathit{dispatch}(\mathrm{fn}[\ldots].\overrightarrow{\mathrm{FT}},\ \mathtt{stop},\ \pi,\ \pi,\ \sigma,\ \mathrm{fid}) & \text{if } \mathrm{FT} = \mathrm{C}[\mathrm{fn}[\ldots].\overrightarrow{\mathrm{FT}}] \\ \cdot & \text{otherwise} \end{cases} \end{aligned}$$

$$\langle \mathrm{FT};\ \mathtt{spawn}\ \mathrm{Pr}\cdot\overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma\rangle \xrightarrow{\ \overline{\mathrm{Pr}}\ } \langle \mathrm{FT};\ \overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma\rangle \qquad \textsf{(spawn)}$$

Fig. 13. Transition relation.

moment we ignore how such events arrive; Section 7 covers that aspect of the semantics. We use $\mathrm{M} \longrightarrow \mathrm{M}'$ as shorthand for $\mathrm{M} \xrightarrow{\bullet} \mathrm{M}'$.

Generally speaking, the machine works in the following fashion. Initially, the machine state describes an inert actor, meaning there are no instructions to perform nor pending scripts to execute. When the machine receives an event patch, it is matched against the event handlers of the actor's facets. For each successful match, the body of the event handler is enqueued as a pending script.

Once the matching process is completed, the machine executes the scripts. It dequeues the first one and interprets the instructions, one at a time. Some instructions generate internal events (facet start and stop). For those the machine matches them again against the facet tree, and this process may enqueue additional scripts. The machine continues in this manner until it reaches inertness again. At that time, another external event may be injected.

Here are descriptions of the machine's five transition rules:

inject —The rule's purpose is to dispatch an event to an inert actor. Due to dataspace routing, the event is guaranteed to conform to the actor's interests. Incorporating the incoming patch via $\oplus$ yields a new set π' of assertions known by the actor. The *dispatch* metafunction matches the event against each handler in the tree of facets, yielding a pending script for each match.

transfer —The machine transfers the next pending script to the current register and partially evaluates (*p-e*-s) its internal statements. In doing so, it updates and creates fields in the store. The process eliminates expressions and yields a sequence of instructions that alter the active facet tree (`start`/`stop`) or correspond to an output by the actor (`spawn`). Partially evaluating the script means *eval*-ing expressions, which may include applications of partial primitives. In the event that such a primitive fails, the actor crashes. The `error` state represents a crashed actor without any behavior.

start —The next instruction demands the start of a facet. The new facet is given a fresh name; the bodies of each of its event handlers refer to the new name. The *dispatch* function is used to create two scripts: one to signal a `start` event and one to boot the event handlers that correspond to the known assertions of the actor. These scripts are enqueued and the new facet is inserted in the tree. The instruction specifies the location for the new facet in the tree as a path (fid) to the new facet's parent. The *locate* metafunction constructs the appropriate context. It is possible that no such context exists, which happens when the desired parent, or one of its anscestors, is terminated via a `stop` instruction in the time between the script containing this `start` statement being enqueued and the instruction executing. In that case, the new facet is immediately terminated. The *dispatch* function informs it of the `stop` event. The resulting scripts are enqueued, and the facet is never inserted into the tree.

stop —The machine terminates an active facet and its children, *dispatch*-ing a `stop` event to allow these facets to shut down in an orderly manner. The instruction may designate a facet that is already eliminated. In this case, there is no more.

spawn —The facet wishes to create an actor, which the machine treats as an output instruction. Formally, the transition comes with a $\overline{\mathrm{Pr}}$ label.

The top-most machine state in Figure 12 transitions to an inert state in two steps. The first one injects the script, consuming the patch $\{\texttt{in-room}(0)\}/\emptyset$ as the label indicates. The second enqueues the script's instruction and performs it.

5.2.1 Metafunctions

The transition system of Figure 13 refers to several metafunctions. This section provides their formal definitions. The definitions make use of secondary metafunctions, which are defined in Appendix B.

The $\mathit{boot}_{\mathrm{Pr}}$ function initializes the machine from an actor description. It recognizes only scripts that create some number of fields and then `start` a single facet. It creates each field and then boots the facet as a child of a synthetic *root* facet with no behavior. The synthetic root facet allows the initial facet to be replaced by any number of facets via `stop`:

$$\begin{array}{rcl}
\mathit{boot}_{\mathrm{Pr}} & : & \mathrm{Pr} \times \sigma \xrightarrow{\mathit{partial}} \mathrm{M} \\
\mathit{boot}_{\mathrm{Pr}}(\texttt{field}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr},\ \sigma) & = & \mathit{boot}_{\mathrm{Pr}}(\mathrm{Pr}[\mathrm{x} \mapsto \mathrm{x}'],\ \sigma[\mathrm{x}' \mapsto \mathrm{v}]) \\
 & & \text{if } \mathrm{v} = \mathit{eval}(\mathrm{e},\ \sigma) \\
 & & \text{where} \\
 & & \mathrm{x}' \text{ fresh in } \sigma, \mathrm{Pr} \\
\mathit{boot}_{\mathrm{Pr}}(\texttt{let}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr},\ \sigma) & = & \mathit{boot}_{\mathrm{Pr}}(\mathrm{Pr}[\mathrm{x} \mapsto \mathrm{v}],\ \sigma) \\
 & & \text{if } \mathrm{v} = \mathit{eval}(\mathrm{e},\ \sigma) \\
\mathit{boot}_{\mathrm{Pr}}(\texttt{start}\ \mathrm{fn}\ \overrightarrow{\mathrm{e}}\ \overrightarrow{(\mathrm{D}\ \mathrm{Pr})},\ \sigma) & = & \langle \mathrm{fn}_{\mathit{root}}[\cdot].\,\mathrm{FT};\ \cdot;\ \overrightarrow{\mathrm{PS}};\ \emptyset;\ \sigma \rangle \\
 & & \text{where} \\
 & & \mathrm{FT} = \mathrm{fn}[\overrightarrow{\mathrm{e}}\ \overrightarrow{(\mathrm{D}\ \mathrm{Pr})}].\epsilon \\
 & & \overrightarrow{\mathrm{PS}} = \mathit{dispatch}(\mathrm{FT},\ \texttt{start},\ \emptyset,\ \emptyset,\ \sigma,\ \mathrm{fn}_{\mathit{root}}) \\
 & & \mathrm{fn}_{\mathit{root}} \text{ free in FT}
\end{array}$$

The *dispatch* function matches an event against each handler in a tree of facets, accumulating the successes as a list of pending scripts:

$$
\begin{array}{rcl}
\mathit{dispatch} & : & \mathrm{FT} \times \mathrm{Evt} \times \pi \times \pi \times \sigma \times \mathrm{fid} \longrightarrow \overrightarrow{\mathrm{PS}} \\
\mathit{dispatch}(\epsilon,\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma,\ \mathrm{fid}) & = & \cdot \\
\mathit{dispatch}(\mathrm{fn}[\overrightarrow{\mathrm{e}}\ (\overrightarrow{\mathrm{D\ \ Pr}})].\overrightarrow{\mathrm{FT}},\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma,\ \langle\overrightarrow{\mathrm{fn}_{ctx}}\rangle) & = & \overrightarrow{\mathit{dispatch1}(\langle\overrightarrow{\mathrm{fn}_{ctx}}\cdot\mathrm{fn}\rangle,\ \mathrm{D},\ \mathrm{Pr},\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma)}\cdot \\
 & & \overrightarrow{\mathit{dispatch}(\mathrm{FT},\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma,\ \langle\overrightarrow{\mathrm{fn}_{ctx}}\cdot\mathrm{fn}\rangle)}
\end{array}
$$

It utilizes a helper function, *dispatch1*, to instantiate the body of a single event handler with the substitution(s) arising from matching against a single event. Substitutions γ map variables to values and matching a pattern against a set of assertions yields a set of substitutions $\mathcal{S}$:

$$
\begin{array}{rcl}
\mathit{dispatch1} & : & \mathrm{fid} \times \mathrm{D} \times \mathrm{Pr} \times \mathrm{Evt} \times \pi \times \pi \times \sigma \longrightarrow \overrightarrow{\mathrm{PS}} \\
\mathit{dispatch1}(\mathrm{fid},\ \mathrm{D},\ \mathrm{Pr},\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma) & = & \begin{cases} (\mathrm{fid},\ \mathit{unroll}(m)) & \text{if } \mathcal{S} = \mathit{match}_{\mathrm{D}}(\mathrm{D},\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma), \mathcal{S} \neq \emptyset \\ & \text{where } m = \{\gamma(\mathrm{Pr}) \mid \gamma \in \mathcal{S}\} \\ \cdot & \text{otherwise} \end{cases}
\end{array}
$$

Finally, *unroll* sequentializes a set of statements using an arbitrary yet fixed ordering:

$$
\begin{array}{rcl}
\mathit{unroll} & : & \mathcal{P}(\mathrm{Pr}) \longrightarrow \mathrm{Pr} \\
\mathit{unroll}(\emptyset) & = & \texttt{skip} \\
\mathit{unroll}(\{\mathrm{Pr}\} \uplus m) & = & \mathrm{Pr};\ \mathit{unroll}(m)
\end{array}
$$

The *p-e* function partially evaluates a script, yielding a sequence of instructions and an updated store:

$$
\begin{array}{rcl}
\mathit{p\text{-}e} & : & \mathrm{Pr} \times \sigma \times \mathrm{fid} \xrightarrow{\mathit{partial}} \overrightarrow{\mathrm{I}} \times \sigma \\
\mathit{p\text{-}e}(\texttt{skip},\ \sigma,\ \mathrm{fid}) & = & \cdot \\
\mathit{p\text{-}e}(\mathrm{Pr}_1;\ \mathrm{Pr}_2,\ \sigma,\ \mathrm{fid}) & = & \overrightarrow{\mathrm{I}_1}\cdot\overrightarrow{\mathrm{I}_2}, \sigma'' \\
 & & \text{if} \\
 & & \overrightarrow{\mathrm{I}_1}, \sigma' = \mathit{p\text{-}e}(\mathrm{Pr}_1,\ \sigma,\ \mathrm{fid}) \\
 & & \overrightarrow{\mathrm{I}_2}, \sigma'' = \mathit{p\text{-}e}(\mathrm{Pr}_2,\ \sigma',\ \mathrm{fid}) \\
\mathit{p\text{-}e}(\texttt{start}\ \mathrm{fn}\ \overrightarrow{\mathrm{e}}\ (\overrightarrow{\mathrm{D\ \ Pr}}),\ \sigma,\ \mathrm{fid}) & = & \texttt{start}\ \mathrm{fn}\ \overrightarrow{\mathrm{e}}\ (\overrightarrow{\mathrm{D\ \ Pr}})\ @\ \mathrm{fid}, \sigma \\
\mathit{p\text{-}e}(\texttt{stop}\ \mathrm{fn}\ \ \mathrm{Pr},\ \sigma,\ \langle\overrightarrow{\mathrm{fn}'}\ \mathrm{fn}\ \overrightarrow{\mathrm{fn}''}\rangle) & = & \texttt{stop}\ \mathrm{fn}\cdot\overrightarrow{\mathrm{I}}, \sigma' \\
 & & \text{if}\ \mathit{p\text{-}e}(\mathrm{Pr},\ \sigma,\ \langle\overrightarrow{\mathrm{fn}'}\rangle) = \overrightarrow{\mathrm{I}}, \sigma' \\
\mathit{p\text{-}e}(\texttt{spawn}\ \mathrm{Pr},\ \sigma,\ \mathrm{fid}) & = & \texttt{spawn}\ \mathrm{Pr}, \sigma \\
\mathit{p\text{-}e}(\texttt{let}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr},\ \sigma,\ \mathrm{fid}) & = & \mathit{p\text{-}e}(\mathrm{Pr}[\mathrm{x} \mapsto \mathrm{v}],\ \sigma,\ \mathrm{fid})\text{if}\ \mathrm{v} = \mathit{eval}(\mathrm{e},\ \sigma) \\
\mathit{p\text{-}e}(\texttt{field}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr},\ \sigma,\ \mathrm{fid}) & = & \mathit{p\text{-}e}(\mathrm{Pr}',\ \sigma',\ \mathrm{fid}) \qquad \text{if}\ \mathrm{v} = \mathit{eval}(\mathrm{e},\ \sigma) \\
 & & \text{where} \\
 & & \mathrm{x}'\ \text{fresh in}\ \sigma, \mathrm{Pr} \\
 & & \sigma' = \sigma[\mathrm{x}' \mapsto \mathrm{v}] \\
 & & \mathrm{Pr}' = \mathrm{Pr}[\mathrm{x} \mapsto \mathrm{x}'] \\
\mathit{p\text{-}e}(\mathrm{x} := \mathrm{e},\ \sigma \uplus \{\mathrm{x} \mapsto \mathrm{v}\},\ \mathrm{fid}) & = & \cdot, \sigma[\mathrm{x} \mapsto \mathrm{v}'] \qquad \text{if}\ \mathrm{v}' = \mathit{eval}(\mathrm{e},\ \sigma)
\end{array}
$$

The *facet-context* function produces the path of facet names (a fid) leading to the hole in a context:

$$
\begin{array}{rcl}
\mathit{facet\text{-}context} & : & \mathrm{C} \longrightarrow \mathrm{fid} \\
\mathit{facet\text{-}context}(\Box) & = & \langle\cdot\rangle \\
\mathit{facet\text{-}context}(\mathrm{fn}[\overrightarrow{\mathrm{e}}\ (\overrightarrow{\mathrm{D\ \ Pr}})].\overrightarrow{\mathrm{FT}}\cdot\mathrm{C}\cdot\overrightarrow{\mathrm{FT}'}) & = & \langle\mathrm{fn}\cdot\overrightarrow{\mathrm{fn}'}\rangle \\
 & & \text{where}\langle\overrightarrow{\mathrm{fn}'}\rangle = \mathit{facet\text{-}context}(\mathrm{C})
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{T} \in \textbf{FacetTy} & = & \texttt{Start}\ \mathrm{fn}\ \vec{\tau}\ (\overrightarrow{\mathrm{D_T}\ \mathrm{T}}) & \tau \in \textbf{Type} & = & \mathrm{B} \\
& \mid & \texttt{Stop}\ \mathrm{fn}\ \mathrm{T} & & \mid & ?\,\tau \\
& \mid & \texttt{Spawn}\ \mathrm{T} & & \mid & m(\vec{\tau}) \\
& \mid & \mathrm{T};\ \mathrm{T} & & \mid & \texttt{Field}\ \tau \\
& \mid & \texttt{Skip} & & \mid & \star \\
& & & & \mid & \mathrm{x} : \tau \\
\mathrm{D_T} \in \textbf{EvtDscTy} & = & \texttt{asserted}\ \tau & & & \\
& \mid & \texttt{retracted}\ \tau & \mathrm{B} \in \textbf{BaseTy} & = & \text{basic types: } \texttt{String}, \texttt{Int}, \text{etc.} \\
& \mid & \texttt{start} & & & \\
& \mid & \texttt{stop} & \Gamma \in \textbf{Env} & = & \cdot \\
& & & & \mid & \Gamma, \mathrm{x} : \tau \\
& & & & \mid & \Gamma, \mathrm{fn} : \texttt{FacetName}
\end{array}
$$

Fig. 14. Type syntax.

Meanwhile, *locate* performs the reverse direction, producing the context for a particular fid in a tree of facets:

$$
\begin{array}{rcl}
\mathit{locate} & : & \mathrm{FT} \times \mathrm{fid} \xrightarrow{\mathit{partial}} \mathrm{C} \\
\mathit{locate}(\mathrm{FT},\ \langle\cdot\rangle) & = & \square \\
\mathit{locate}(\mathrm{fn}[\vec{\mathrm{e}}\ (\overrightarrow{\mathrm{D}\ \mathrm{Pr}})].\vec{\mathrm{FT}},\ \langle \mathrm{fn} \cdot \vec{\mathrm{fn}'}\rangle) & = & \mathrm{fn}[\vec{\mathrm{e}}\ (\overrightarrow{\mathrm{D}\ \mathrm{Pr}})].\vec{\mathrm{FT}}_1 \cdot \mathrm{C} \cdot \vec{\mathrm{FT}}_2 \\
& & \text{if} \\
& & \mathit{locate}(\mathrm{FT}',\ \langle\vec{\mathrm{fn}'}\rangle) = \mathrm{C} \\
& & \vec{\mathrm{FT}} = \vec{\mathrm{FT}}_1 \cdot \mathrm{FT}' \cdot \vec{\mathrm{FT}}_2
\end{array}
$$

6 Types for facets

Facet types (Section 6.1) play two roles. First, they ensure basic type safety in the form of a standard soundness theorem. Second, they reflect facet operations to the type level with a type-and-effect style. By lifting the individual actor semantics to the type level (Section 6.2), and then dataspace programs, type-level programs can be assigned an operational semantics that enables model checking temporal logic claims.

6.1 Type syntax and judgment

Figure 14 defines the syntax of types. Effect types T directly reflect the core facet syntax (Figure 10), including facet `start`/`stop` and `spawn` operations of Pr. Event descriptor types $\mathrm{D_T}$ correspond directly to the term level descriptors D. Types τ summarize the result of expressions, with base types B describing primitive values. Type environments Γ associate variable names from patterns and fields with their types and facet names with the marker `FacetName`.

Figure 15 specifies the core type judgment on facets: $\Gamma \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}$. It ensures that facet and field names are used appropriately while collecting certain facet operations as an effect type T. As mentioned, this effect type is used for behavioral analysis. Auxiliary judgments apply to the different categories of syntax; Appendix D provides their definitions.

Here are explanations of the key typing rules:

T-Start assigns types to a facet-creation instruction. It appeals to an auxiliary judgment for checking the assertion expressions e, which also produces a description of the assertion types τ. The condition $\texttt{assertable}(\tau)$, defined in Appendix D, ensures

$$\boxed{\Gamma \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{Pr}} \texttt{skip} : \texttt{Skip}}\ \boxed{\text{T-SKIP}} \qquad \frac{\Gamma \vdash_{\mathrm{Pr}} \mathrm{Pr}_1 : \mathrm{T}_1 \qquad \Gamma \vdash_{\mathrm{Pr}} \mathrm{Pr}_2 : \mathrm{T}_2}{\Gamma \vdash_{\mathrm{Pr}} \mathrm{Pr}_1;\ \mathrm{Pr}_2 : \mathrm{T}_1;\ \mathrm{T}_2}\ \boxed{\text{T-SEQ}}$$

$$\frac{\overrightarrow{\texttt{assertable}(\tau)} \qquad \overrightarrow{\Gamma \vdash_{\mathrm{D}} \mathrm{D} : \mathrm{D_T}} \qquad \begin{array}{c}\overrightarrow{\Gamma \vdash_{\mathrm{e}} \mathrm{e} : \tau} \\ \overrightarrow{\Gamma, \mathrm{fn} : \texttt{FacetName}, \mathit{bindings}_{\mathrm{D}}(\mathrm{D}) \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}\end{array}}{\Gamma \vdash_{\mathrm{Pr}} \texttt{start}\ \mathrm{fn}\ \vec{\mathrm{e}}\ (\overrightarrow{\mathrm{D}\ \mathrm{Pr}}) : \texttt{Start}\ \mathrm{fn}\ \vec{\tau}\ (\overrightarrow{\mathrm{D_T}\ \mathrm{T}})}\ \boxed{\text{T-START}}$$

$$\frac{\Gamma(\mathrm{fn}) = \texttt{FacetName} \qquad \mathit{prune\text{-}up\text{-}to}(\mathrm{fn}, \Gamma) \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}{\Gamma \vdash_{\mathrm{Pr}} \texttt{stop}\ \mathrm{fn}\ \mathrm{Pr} : \texttt{Stop}\ \mathrm{fn}\ \mathrm{T}}\ \boxed{\text{T-STOP}}$$

$$\frac{\mathit{prune}(\Gamma) \vdash_{\mathrm{Pr}} \mathrm{Pr} : \texttt{Start}[\ldots]}{\Gamma \vdash_{\mathrm{Pr}} \texttt{spawn}\ \mathrm{Pr} : \texttt{Spawn}\ \texttt{Start}[\ldots]}\ \boxed{\text{T-SPAWN}}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} \mathrm{e} : \tau \qquad \Gamma, \mathrm{x} : \texttt{Field}\ \tau \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}{\Gamma \vdash_{\mathrm{Pr}} \texttt{field}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr} : \mathrm{T}}\ \boxed{\text{T-FIELD}} \qquad \frac{\Gamma \vdash_{\mathrm{e}} \mathrm{e} : \tau \qquad \Gamma(\mathrm{x}) = \texttt{Field}\ \tau}{\Gamma \vdash_{\mathrm{Pr}} \mathrm{x} := \mathrm{e} : \texttt{Skip}}\ \boxed{\text{T-ASSIGN}}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} \mathrm{e} : \tau \qquad \Gamma, \mathrm{x} : \tau \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}{\Gamma \vdash_{\mathrm{Pr}} \texttt{let}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{Pr} : \mathrm{T}}\ \boxed{\text{T-LET}}$$

Fig. 15. Facet typing.

that assertion types do not contain binding patterns or field names. The event descriptors D for each event handler are checked in the same manner, yielding a type $\mathrm{D_T}$ for each. The final aspect to check is the body Pr of each event handler. For each such body, the environment is extended with the name of the facet being started as well as the binding variables from the pattern in the event descriptor (via the $\mathit{bindings}_{\mathrm{D}}$ metafunction). Checking each body produces a type level description of its behavior, T. The product of the rule is a `Start` effect, combining the types of the assertions plus each event handler description and its body.

T-STOP is the analogue of T-START for facet termination. It consults the environment to ensure that the named facet is in context. The *prune-up-to* function removes the target facet's name from the environment, as well as the names of any of its descendants, when checking the continuation Pr. The resulting effect type is to `Stop` the same facet with a continuation behavior.

T-FIELD, T-ASSIGN check field creation and assignment, respectively. The judgment for field creation checks the type of the initial expression and associates the name with an appropriate type in the environment for the remaining statements. The judgment for field assignment looks up that type in the environment, making sure that the type of the assigned expression matches. Field updates and references are not included in effect types. Thus, field creation yields the effect type from its enclosed statements while field assignment yields a `Skip` effect.

T-SPAWN concerns the creation of actors. The rule utilizes the *prune* function (Appendix D) to remove identifiers associated with fields and facet names from the running actor, because these names are local. The spawned actor's behavior Pr is checked within this restricted context, and it has the (sole) effect of `Start`-ing a facet. The effect type of the `spawn` statement is a `Spawn` with a type description of the actor's initial facet.

6.2 Machine types and type-level machines

The judgment $\vdash_{\mathrm{M}}\ \mathrm{M}\ :\ \mathrm{M}_{\mathrm{T}}$ relates a machine M to a machine type M_{T}. A machine type $\langle \mathrm{FT}_{\mathrm{T}};\ \overrightarrow{\mathrm{I}_{\mathrm{T}}};\ \overrightarrow{\mathrm{PS}_{\mathrm{T}}};\ \pi_\tau;\ \sigma_\tau\rangle$ includes a facet tree type (FT_{T}), sequence of instruction types ($\overrightarrow{\mathrm{I}_{\mathrm{T}}}$), sequence of pending script types ($\overrightarrow{\mathrm{PS}_{\mathrm{T}}}$), set of assertion types (π_τ), and store type (σ_τ). Appendix F defines the full judgment and syntax. In brief, the syntax of machine types reflects the syntax of machines (Figure 11) in the same manner that effect types T (Figure 14) reflect the syntax of statements Pr (Figure 10).

The statement of soundness for facet-based actors relies on two auxiliary notions. First, we say M is an inert machine state, $\mathtt{inert}(\mathrm{M})$, iff $\mathrm{M} = \langle \mathrm{FT};\ \cdot;\ \cdot;\ \pi;\ \sigma\rangle$. That is, it has neither instructions to perform nor pending scripts to execute. Second, we designate a set of *internal* transition labels, $l_\bullet$. Internal labels correspond to the machine processing instructions and scripts. Thus, they consist of each $\bullet$ and $\overline{\mathrm{Pr}}$ label, but not Δ.

Theorem 1 (Soundness)**.** *If* $\vdash_{\mathrm{M}}\ \mathrm{M}\ :\ \mathrm{M}_{\mathrm{T}}$ *and* $\mathrm{M} \xrightarrow{\Delta} \mathrm{M}'$ *then either:*

- $\mathrm{M}' \xrightarrow{l_\bullet}{}^{*} \mathrm{M}''$ *and* $\mathit{inert}(\mathrm{M}'')$*; or*
- $\mathrm{M}' \xrightarrow{l_\bullet}{}^{*}$ error

Proof By the standard progress and preservation properties (Wright & Felleisen, 1994). Appendix G contains the full statement and proof of each property. To show termination of internal reduction sequences, note the lack of recursive facilities in the language, and that each event handler in a machine's facet tree may activate a maximum of one time between external events. □

If a well-typed single-actor machine configuration can take an inject step, it eventually transitions to an inert machine state or a state that represents an exceptional state (due to a misapplication of a partial primitive).

The transition system for machine states operates on machine types with only minor syntactic adjustments. Appendix F defines the full type-level transition system. Section 8 makes use of this notion for behavioral analysis of actors. Here, we note that transitions on type machines are well-defined.

Theorem 2 (Well-definedness)**.** *Either* $\mathrm{M}_{\mathrm{T}} \xrightarrow{l_\tau} \mathrm{M}'_{\mathrm{T}}$ *or* $\mathit{inert}(\mathrm{M}_{\mathrm{T}})$.

Proof Follows that of Theorem 1. □

7 Dataspaces: The semantics of programs

While the preceding section describes the semantics of a *single* actor, a full understanding necessitates a semantics of entire dataspace programs, that is, programs with many, dynamically created and terminated, actors. Figure 16 defines the abstract syntax (left) and the semantics (right) of dataspace programs.

A dataspace configuration consists of a queue of pending patches (ℓ, Δ), labeled by the originating actor; a table of assertions (R); and a series of actors (A). The table R is a set of pairs associating each assertion with the identity ℓ of the originating actor. An

$$
\begin{array}{rcl}
\mathrm{DS} \in \textbf{Dataspace} & = & [\overrightarrow{(\ell, \Delta)}; \mathrm{R}; \overrightarrow{\mathrm{A}}] \\
\mathrm{R} \in \textbf{Table} & = & \mathscr{P}(\mathrm{u} \times \ell) \\
\ell \in \textbf{Address} & = & \mathscr{X} \\
\mathrm{A} \in \textbf{Actor} & = & \ell \mapsto \Sigma \\
\Sigma \in \textbf{ActorState} & = & \Delta \triangleright \mathrm{B} \\
\mathrm{act} \in \textbf{Action} & = & \Delta \mid \mathrm{P} \\
\mathrm{B} \in \textbf{Behavior} & = & (\mathrm{f},\ \mathrm{v}) \\
\mathrm{P} \in \textbf{ProcDesc} & = & \texttt{actor}\ \mathrm{f}\ \mathrm{v}\ \pi \\
\mathrm{f} \in \textbf{BehFun} & = & \exists \mathit{State}. \Delta \times \mathit{State} \longrightarrow \mathit{State} \times \overrightarrow{\mathrm{act}}
\end{array}
$$

$$
\boxed{
\begin{array}{l}
[\overrightarrow{(\ell_{\mathrm{act}}, \Delta)}; \mathrm{R}; \overrightarrow{\mathrm{A}}] \longrightarrow_{\mathrm{DS}} [\overrightarrow{(\ell, \Delta')} \cdot \overrightarrow{(\ell', \pi/\emptyset)}; \mathrm{R}'; \overrightarrow{(\ell_{\mathrm{A}} \mapsto \Sigma')} \cdot \overrightarrow{(\ell' \mapsto \Sigma'')}] \\
\text{where} \\
\quad \begin{array}{lcl}
\mathrm{R}', \overrightarrow{\ell_{\mathrm{A}} \mapsto \Sigma} & = & \texttt{foldL}\ \mathit{update}\ \overrightarrow{(\ell_{\mathrm{act}}, \Delta)}\ (\mathrm{R}, \overrightarrow{\mathrm{A}}) \\
\overrightarrow{\Sigma', \Delta', \overrightarrow{\mathrm{P}}} & = & \overrightarrow{\mathit{dispatch}_{\mathrm{DS}}(\Sigma)} \\
\overrightarrow{\Sigma'', \pi} & = & \overrightarrow{\mathit{boot}_{\Sigma}(\mathrm{P})} \\
 & & \text{with } \overrightarrow{\ell'} \text{ fresh in } \overrightarrow{\ell_{\mathrm{A}}}
\end{array}
\end{array}
}
$$

Fig. 16. Dataspace syntax and semantics.

active actor in the dataspace $\ell \mapsto \Sigma$ associates an address with an actor state, Σ. The state consists of a pending event Δ and behavior B. The pending event accumulates a patch for the actor between its turns. An actor behavior (f, v) pairs a behavior function with a private state. The behavior function f is a total mapping between a pending event and the private state of the actor to a new state and possibly some actions (act). The model abstracts over the language for specifying behavior functions so both facet-based code as well as alternatives work. Actions include actor specifications P and patches Δ. An actor specification `actor` f v π consists of three pieces: a behavior function; its initial state, which is any value from the language for expressing behavior functions; and its initial assertions.

The semantics is defined in terms of a transition relation on dataspace configurations. The relation interprets actor actions and dispatches events in a single rule (step-par). It starts by interpreting the queue of pending patches. The *update* metafunction handles a single queue item, updating the dataspace's assertion table and dispatching events. The pending event for each actor aggregates a patch describing relevant updates. Next, the $\mathit{dispatch}_{\mathrm{DS}}$ metafunction applies the behavior of each actor to its pending patch, producing an updated actor state and some outputs. Any spawn actions are booted, yielding a new address, actor state, and assertions. Finally, all of the patches and boot assertions are collected into a new queue for the next configuration. It includes the updated assertion table and freshly booted actors. Appendix E contains the full definition of these metafunctions.

Definition 3 (DS $_i$)**.** A *run* of a dataspace configuration DS is a potentially infinite sequence of dataspaces configurations, where each neighboring pair is related by the transition relation:

$$\mathrm{DS} \longrightarrow_{\mathrm{DS}} \mathrm{DS}_{\,1} \longrightarrow_{\mathrm{DS}} \mathrm{DS}_{\,2} \longrightarrow_{\mathrm{DS}} \ldots$$

It is a run that represents the temporal behavior of a dataspace program and its individual actors, and it is thus the foundation for verifying the value of our behavioral types.

Definition 4 ($\texttt{inert}_{\mathrm{DS}}$)**.** A dataspace configuration $[\overrightarrow{(\ell, \Delta)}; \mathrm{R}; \overrightarrow{\mathrm{A}}]$ is inert iff each patch and pending event is empty.

Theorem 5 (Dataspace Soundness)**.** *Either* $\texttt{inert}_{\mathrm{DS}}(\mathrm{DS})$ *or there exists* DS$'$ *such that* $\mathrm{DS} \longrightarrow_{\mathrm{DS}} \mathrm{DS}'$.

Proof Garnock-Jones's dissertation proves the soundness of dataspace systems, as well as a number of desirable properties (Garnock-Jones, 2017, theorem 4.17).

7.1 Facet dataspace programs

Caldwell *et al.* (2020) show how to link a semantics for individual actors with the dataspace semantics and prove the soundness of the composite system. Their semantics coincides with ours, so that the soundness of the system follows from Theorem 5, because each actor implements the functional interface of a **BehFun**. The $\mathit{interp}_{\mathrm{M}}$ function is the key component connecting the semantics of dataspaces with the semantics of facet actors. It implements the dataspace behavior interface **BehFun** for facet machine states, and it applies a pending patch to a machine state via an inject-step, then allows the machine to continue via internal transitions until reaching either inertness or an error state. The result of the function is the new machine state together with a patch describing the difference(s) in assertions between the initial and final states. Furthermore, any $\overline{\mathrm{Pr}}$ transitions are translated to dataspace specifications P:

$$\begin{array}{rcl}
\mathit{interp}_{\mathrm{M}} & : & \Delta \times \mathrm{M} \longrightarrow \mathrm{M} \times \overrightarrow{\mathrm{act}} \\
\mathit{interp}_{\mathrm{M}}(\Delta,\ \mathtt{error}) & = & (\mathtt{error}, \cdot) \\
\mathit{interp}_{\mathrm{M}}(\Delta,\ \mathrm{M}) & = & \begin{cases} (\mathtt{error}, \mathit{patch}(\mathrm{M},\ \mathtt{error})) & \text{if } \mathrm{M} \xrightarrow{\Delta} \mathrm{M}' \xrightarrow{l_\bullet}{}^{*}\ \mathtt{error} \\ (\mathrm{M}'', \mathit{patch}(\mathrm{M},\ \mathrm{M}'') \cdot \overrightarrow{\mathit{label\text{-}to\text{-}action}(l_\bullet)}) & \text{if } \mathrm{M} \xrightarrow{\Delta} \mathrm{M}' \xrightarrow{l_\bullet}{}^{*}\ \mathrm{M}'', \mathtt{inert}(\mathrm{M}'') \end{cases}
\end{array}$$

The $\mathit{boot}_{\mathrm{P}}$ metafunction translates a facet-level spawn to a dataspace process description:

$$\begin{array}{rcl}
\mathit{boot}_{\mathrm{P}} & : & \mathrm{Pr} \longrightarrow \mathrm{P} \\
\mathit{boot}_{\mathrm{P}}(\mathrm{Pr}) & = & \mathtt{actor}\ \mathit{interp}_{\mathrm{M}}\ \mathrm{M}_{\mathit{boot}}\ \mathit{assertions\text{-}of}_{\mathrm{M}}(\mathrm{M}_{\mathit{boot}}) \\
 & & \text{where } \mathrm{M}_{\mathit{boot}} = \begin{cases} \mathrm{M} & \text{if } \mathit{boot}_{\mathrm{Pr}}(\mathrm{Pr},\ \emptyset) = \mathrm{M} \\ \mathtt{error} & \text{otherwise} \end{cases}
\end{array}$$

7.2 Type-level dataspace programs

Section 6.2 sketches how to lift the transition system on a facet machine to operate on a facet machine type. Similarly, the dataspace transition system above naturally lifts to the type level with only minor adjustments. A type-level dataspace consists of actors communicating in terms of assertion types τ. Facet machine types provide behavior to actors in a type-level dataspace the same way that facet machines give behavior to term-level dataspace actors. We write DS_{T} to refer to a dataspace configuration with type assertions.

7.3 Example

Figure 17 shows a dataspace configuration based on the smart home example and its evolution. For readability, the example employs a notation different from that of Figure 16, giving each actor a name rather than an address and aggregating various other parts of the syntax. Each actor is displayed as a triple, as in $\Delta \triangleright \boxed{nm} \triangleright \pi$ for an actor with name *nm*. The leftmost element is the current pending event for the actor, while on the right are its current set of assertions. The example treats the behavior and private state of each actor as a black box, displaying them as a box containing the actor's name. For brevity, we use two

$$\begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star)\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", on)},\ ?\,\texttt{assign("l1", }\star\texttt{)}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow \begin{array}{lll} [\ \texttt{light("l1", on)}\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star)\} \\ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", on)},\ ?\,\texttt{assign("l1", }\star\texttt{)}\} \\ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow$$

$$\begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", on)},\ ?\,\texttt{assign("l1", }\star\texttt{)}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow \begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \texttt{assign("l1", "bed")}\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", on)},\ ?\,\texttt{assign("l1", }\star\texttt{)}\} \\ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow$$

$$\begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", on)},\ ?\,\texttt{assign("l1", }\star\texttt{)},\ ?\,\texttt{empty("bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow \begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \texttt{empty("bed")}\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", on)},\ ?\,\texttt{assign("l1", }\star\texttt{)},\ ?\,\texttt{empty("bed")}\} \\ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow$$

$$\begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", off)},\ ?\,\texttt{assign("l1", }\star\texttt{)},\ ?\,\texttt{empty("bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow \begin{array}{lll} [\ \frac{\{\texttt{light("l1", off)}\}}{\{\texttt{light("l1", on)}\}}\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", off)},\ ?\,\texttt{assign("l1", }\star\texttt{)},\ ?\,\texttt{empty("bed")}\} \\ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array} \longrightarrow$$

$$\begin{array}{lll} [\ \cdot\ \triangleright & \boxed{\mathit{hub}} & \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign("l1", "bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{l1}} & \triangleright \{\texttt{light("l1", off)},\ ?\,\texttt{assign("l1", }\star\texttt{)},\ ?\,\texttt{empty("bed")}\} \\ \ \ \cdot\ \triangleright & \boxed{\mathit{ps1}} & \triangleright \{\texttt{empty("bed")}\}\ ] \end{array}$$

Fig. 17. Example Dataspace Transitions

additional notational shorthands. As a pending event, · represents the empty patch ∅/∅, while a single assertion u stands for the patch { u }/∅.

The configuration contains three actors, representing the control hub, a light with the ID `"l1"`, and a presence sensor with the ID `"ps1"`. Each transition of the relation from Figure 16 is broken into two steps. The first routes assertions among the actors and updates pending events when there is relevant information. The second dispatches each actor on its pending event and updates its current assertions.

The system proceeds in the following manner:

1. Initially, the light actor announces its presence and on/off state, as well as a subscription to a room assignment.

$$\begin{array}{rl} [\;\cdot \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star)\} \\ \cdot \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star)\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array} \rightarrow \begin{array}{rl} [\;\texttt{light}(\texttt{Str},\ \texttt{Bool}) \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star)\} \\ \cdot \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star)\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array} \rightarrow$$

$$\begin{array}{rl} [\;\cdot \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign}(\texttt{Str},\ \texttt{Str})\} \\ \cdot \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star)\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array} \rightarrow \begin{array}{rl} [\qquad \cdot \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign}(\texttt{Str},\ \texttt{Str})\} \\ \texttt{assign}(\texttt{Str},\ \texttt{Str}) \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star)\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array} \rightarrow$$

$$\begin{array}{rl} [\;\cdot \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign}(\texttt{Str},\ \texttt{Str})\} \\ \cdot \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star),\ ?\,\texttt{empty}(\texttt{Str})\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array} \rightarrow \begin{array}{rl} [\qquad \cdot \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign}(\texttt{Str},\ \texttt{Str})\} \\ \texttt{empty}(\texttt{Str}) \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star),\ ?\,\texttt{empty}(\texttt{Str})\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array} \rightarrow$$

$$\begin{array}{rl} [\;\cdot \triangleright & \boxed{\mathit{Hub}} \triangleright \{?\,\texttt{light}(\star,\ \star),\ \texttt{assign}(\texttt{Str},\ \texttt{Str})\} \\ \cdot \triangleright & \boxed{\mathit{L1}} \triangleright \{\texttt{light}(\texttt{Str},\ \texttt{Bool}),\ ?\,\texttt{assign}(\texttt{Str},\ \star),\ ?\,\texttt{empty}(\texttt{Str})\} \\ \cdot \triangleright & \boxed{\mathit{PS1}} \triangleright \{\texttt{empty}(\texttt{Str})\}\;] \end{array}$$

Fig. 18. Example type dataspace transitions.

2. The hub actor's interest in `light` assertions yields a corresponding event.
3. In response, the hub actor introduces a room assignment to the `"bed"` room.
4. Routing results in an event informing the light actor of the assignment.
5. The light actor's reaction is to introduce a subscription for assertions that the `"bed"` room is `empty`.
6. The subscription matches the existing `empty` assertion, immediately yielding an event for the light actor.
7. The light actor updates its `light` assertion to the `off` state in response to the room being empty.
8. The hub actor receives a patch describing the change to the `light` assertion.
9. Finally, the hub actor processes the latest patch without changing its assertions, yielding an inert system.

Figure 18 shows the corresponding type-level system. The primary difference is that the assertions in the dataspace are types rather than values; the names of the actors have also been capitalized to emphasize the difference between the two examples. The transitions follow the same logic as those of Figure 17, with one exception. The example encodes the on/off state of the light as a `Bool`. Consequently, the types of assertions in the dataspace do not change when the light switches from on to off in response to an `empty` assertion (type). Thus, the type-level system reaches inertness in fewer transitions than the term-level one.

8 The predictive nature of behavioral types

Equipped with a semantics of facets and dataspaces, we are now in a position to verify the predictive nature of our behavioral types. Specifically, we wish to show that if a program's behavioral types satisfy a specification, the program itself does, too. We start with a model-appropriate definition of our LTL specifications.

Definition 6 (LTL). $\psi \in \mathbf{LTL} = \tau \mid \bullet\psi \mid \psi\ \mathbf{U}\ \psi \mid \neg\psi \mid \psi \vee \psi$

The atomic propositions are types τ of dataspace assertions. Our basic syntax comes with two temporal operators: $\bullet\psi$ (next) and $\psi_1\ \mathbf{U}\ \psi_2$ (strong-until). The set of formulas includes negation ($\neg\psi$) and disjunction ($\psi \vee \psi$). **Note** Standard constructs such as conjunction ($\psi \wedge \psi$), eventually ($\diamond\psi$), and always ($\Box\psi$) are derived forms.

Formulating the meaning of the LTL specifications takes two steps. The first introduces the set of atomic propositions of a dataspace.

Definition 7 (**AP**(DS))**.**

$$\begin{array}{ll} \tau \in \mathbf{AP}([\overrightarrow{(\ell, \Delta)}; \mathrm{R}; \overrightarrow{\mathrm{A}}]) & \text{iff there exists } (\mathrm{u}, \ell) \text{ s.t. } (\mathrm{u}, \ell) \in \mathrm{R},\ \vdash_{\mathrm{e}}\ \mathrm{u}\ :\ \tau \\ \tau \in \mathbf{AP}([\overrightarrow{(\ell, \Delta_\tau)}; \mathrm{R_T}; \overrightarrow{\mathrm{A}}]) & \text{iff there exists } \ell \text{ s.t. } (\tau, \ell) \in \mathrm{R_T} \end{array}$$

That is, an atomic proposition belongs to a dataspace configuration, if its table of assertions u contains an assertion (by any actor l) that type checks as τ. We overload the notation for type-level dataspace configurations, **AP**($\mathrm{DS_T}$). At the type level, an atomic proposition τ belongs to the configuration if such a type is present in the configuration's assertions ($\mathrm{R_T}$).

The second step concerns the notion of satisfaction, that is, when a dataspace configuration—and its run—satisfies an LTL specification.

Definition 8 (DS $\models \tau$)**.** For both term and type-level configurations,

$$\begin{array}{lll} \mathrm{DS} \models \tau & \text{iff} & \tau \in \mathbf{AP}(\mathrm{DS}) \\ \mathrm{DS} \models \neg\psi & \text{iff} & \mathrm{DS} \not\models \psi \\ \mathrm{DS} \models \psi_1 \vee \psi_2 & \text{iff} & \mathrm{DS} \models \psi_1 \text{ or } \mathrm{DS} \models \psi_2 \\ \mathrm{DS} \models \bullet\psi & \text{iff} & \mathrm{DS}_1 \models \psi \\ \mathrm{DS} \models \psi_1\ \mathbf{U}\ \psi_2 & \text{iff} & \text{there exist an } i \text{ s. t. } \mathrm{DS}_i \models \psi_2 \text{ and } \forall j, 0 \leq j < i, \mathrm{DS}_j \models \psi_1 \end{array}$$

At the term level, atomic propositions hold in dataspace configurations where an actor is making an assertion of that type. At the type level, they hold for configurations where an actor is asserting that type. The temporal constructor $\bullet\psi$ holds in a dataspace configuration when ψ holds starting from the next step in the run. Similarly, the other temporal operator,

Formula	Term Meaning	Type Meaning
`light(Bool)`	Either `light(true)` or `light(false)` is currently asserted in the dataspace.	`light(Bool)` asserted.
$\neg$`light(Bool)`	Neither `light(true)` nor `light(false)` is currently asserted in the dataspace.	`light(Bool)` not asserted.
$\diamond$`light(Bool)`	The execution reaches a `light(Bool)` configuration in a finite number of steps.	Same as term meaning.
$\square$`light(Bool)` $\Rightarrow$ (`light(Bool)` **U** $\neg$`wall-switch-on()`)	Whenever `light(Bool)` is true, there is eventually no `wall-switch-on()` assertion in the dataspace, and `light(Bool)` stays true until then.	Same as term meaning.

Fig. 19. Formal LTL examples.

strong until (ψ_1 **U** ψ_2), holds when ψ_1 is true for some finite number of steps in a dataspace run, at which point ψ_2 is true. All other syntactic forms have their usual meanings.

At this point, we can define what it means for a program—a description of the initial facet-based actors in the dataspace, or their types—to satisfy a specification.

Definition 9 ($\cdot \models \psi$).

$$\begin{array}{lll} \overrightarrow{\mathrm{Pr}} \models \psi & \text{iff} & \mathit{boot}_{\mathrm{DS}}(\overrightarrow{\mathrm{Pr}}) = \mathrm{DS} \wedge \mathrm{DS}_{\ 1} \models \psi \\ \overrightarrow{\mathrm{T}} \models \psi & \text{iff} & \mathit{boot}_{\mathrm{DS_T}}(\overrightarrow{\mathrm{T}}) = \mathrm{DS_T} \wedge \mathrm{DS_{T\ 1}} \models \psi \end{array}$$

Note how the definitions use the first successor of the initial dataspace configuration to check the desired formula. They ignore the initial one because all of the actors' initial assertions are in the pending action queue, waiting to be interpreted. The first transition step moves these assertions into the configuration's assertion table.

Figure 19 illustrates how to translate the example specification from Section 4.2 into the formal syntax. It also indicates what it means at the term and the type level.

The type system and the behavioral type system properly predict the behavior of actors and programs. Naturally, the first is the basis for the second. That is, all correspondence between terms and types relies on standard soundness (Theorem 1). The point of soundness is to confirm that there are distinct classes of actor programs: those that live up to (type) expectations and those that raise exceptions. We collect the first kind in a set `safe`(M) for a configuration M that does not error due to the application of a partial primitive, such as division by zero. We write `safe`(Pr) if $\mathit{boot}_{\mathrm{Pr}}$-s evaluates to a safe machine state.

Next we establish a correspondence between the behavior of a single actor and its type. Specifically, a term-level actor machine is related to a type-level actor machine if they make the same assertion types; the type-level machine types the term-level one; they react to the same (types of) events; and their reactions to those events yield related states.

Definition 10. $\mathrm{M} \approx \mathrm{M_T}$ if and only if:

- $\vdash_\pi$ *assertions-of*$_{\mathrm{M}}(\mathrm{M})$: *assertions-of*$_{\mathrm{M_T}}(\mathrm{M_T})$
- $\forall l$ such that $\mathrm{M} \xrightarrow{l} \mathrm{M}', \exists l_\tau, \vdash_l l : l_\tau$ such that $\mathrm{M_T} \xrightarrow{l_\tau} \mathrm{M}'_\mathrm{T}$ and $\mathrm{M}' \approx \mathrm{M}'_\mathrm{T}$
- $\forall l_\tau$ such that $\mathrm{M_T} \xrightarrow{l_\tau} \mathrm{M}'_\mathrm{T}, \exists l, \vdash_l l : l_\tau$ such that $\mathrm{M} \xrightarrow{l} \mathrm{M}'$ and $\mathrm{M}' \approx \mathrm{M}'_\mathrm{T}$

Appendix F provides the straightforward definitions of the $\vdash_\pi$ and $\vdash_l$ judgments.

Assuming the machine belongs to the set of safe ones, typing implies correspondence.

Theorem 11. *If* $\vdash_\mathrm{M}\ \mathrm{M} : \mathrm{M_T}$ *and* $\mathit{safe}(\mathrm{M})$ *then* $\mathrm{M} \approx \mathrm{M_T}$.

Proof The initial machine states for a term and its type are related, as are their assertions. The key meta functions preserve typing, and a suitably related event can always be constructed. See Appendix G for the full statement of these properties and their proofs. □

Theorem 11 establishes a similarity between term and type level machines based on their inputs (injected events) and output labels (spawned actors).

The point of this theorem is that an actor machine and its type implement the same interface, up to typing. A dataspace program consisting of actor machines interacts with its context in the same way as a dataspace program consisting of those actor machines' types.

Finally, we can state the key theorem, namely, that type-level behavior carries over to term-level dataspace programs. That is, if a collection of actor terms $\overrightarrow{\mathrm{Pr}}$ have types $\overrightarrow{\mathrm{T}}$ and boots to exception-free dataspace programs, then LTL properties satisfied by the type-level dataspace program also hold for the term-level one.

Theorem 12 (LTL Transference). *If*

- $\overrightarrow{\vdash_{\mathrm{Pr}}\ \mathrm{Pr} : \mathrm{T}}$
- $\overrightarrow{\mathit{safe}(\mathrm{Pr})}$
- $\overrightarrow{\mathrm{T}} \models \psi$

then $\overrightarrow{\mathrm{Pr}} \models \psi$

Proof Due to the simulation property (Theorem 11), the table of assertions in each step of the term and type level dataspace runs is related by typing. Therefore, the same atomic predicates (**AP**) hold in each, so the same LTL properties apply. □

Note The assumption of safety for the single-actor machines is standard for the statement of partial correctness. In essence, it is equivalent to the conventional assumption "if the statement terminates properly" in Floyd-Hoare logic (Floyd, 1967; Hoare, 1969).

In detail, our type analysis does not seek to capture the behavior of actors in the case of uncaught exceptions. While dealing with failures is a core principle of the actor and dataspace model, the current type system does not capture this notion.

If we were to account for the possibility of unexpected exceptions, every non-trivial actor would include the behavior of immediately crashing. Every program would then include the situation where all its actors crash and other such degenerate cases, vastly increasing the difficulty of stating LTL properties of the system. Even the addition of supervision actors would not avoid this collapse in reasoning capability.

We conjecture that by generalizing work on exception analysis via type systems (Yi & Ryu, 2002) it might be possible to extend our type system to reason about such cases precisely, at least for some limited class of programs. We leave this problem as future work.

9 Implementation

An implementation of the theoretical design requires three pieces: the language itself (its syntax and semantics); a type checker that derives effect types; and a compilation of these effect types plus the specification to a model-checking system. The following three subsections sketch these three pieces of our implementation.

9.1 Implementing the behavior of facet-oriented actors

The basic language implementation of facet-oriented actors consists of three layers:

- a syntax layer, which provides the facet notation of the first half of this paper;
- a runtime system, which provides data structures and functions that implement the behavior of facets and endpoints; and
- an imperative dataflow network for tracking changes to fields and scheduling the re-evaluation of dependent computations.

These pieces are built atop the existing implementation of the dataspace model. The dataspace implementation does not require any modifications in order to support these elements, just as the dataspace model predicts (Garnock-Jones & Felleisen, 2016).

The syntax layer essentially turns the surface forms into calls to functions provided by the runtime system. It makes use of `syntax/parse`, Racket's high-level syntax extension system (Culpepper & Felleisen, 2010, 2012). Implementing the surface language with `syntax/parse` greatly simplifies the addition of actor syntax. Moreover, the resulting interface can be extended and grown as the language develops, and the implementation can benefit from any improvements to the underlying dataspace library. It also facilitates the addition of a type checker.

The run-time system is the most sophisticated of the three layers. The key elements strongly mirror the data structures of the machine and the meta-functions of the formal model of Section 5. Technically speaking, the run-time system maintains data structures corresponding to the different parts of machine states M and defines functions for dealing with facets, enqueuing scripts, etc. The implementation differs from the model primarily in its support for efficient re-evaluation of the assertions of an actor as fields are updated.

Finally, the runtime maintains a bipartite, directed dataflow graph for each facet-oriented actor. A source node represents a field, a target node an endpoint, and edges the dependencies of the endpoints on the fields. Each endpoint representation contains a procedure that is used to compute the set of assertions to be associated with the endpoint. By recording field dependencies during the execution of such procedures, the implementation learns which endpoints must have their assertion sets recomputed in response to a field change.

9.2 Implementing the type checker

The type checker is implemented with the Turnstile meta-DSL (Chang *et al.*, 2017). Roughly speaking, Turnstile macros supplement `syntax/parse` with a mechanism to check types first and elaborate the given source syntax into target terms later. That is, each syntactic form is defined as a macro that performs some type checking before elaborating to syntax that implements the run-time behavior. Type information is propagated via metadata on syntax objects.

For example, the `define-typed-syntax` macro for `start-facet` looks like this:

```
(define-typed-syntax (start-facet name:id ep:expr ...+) ≫
  [[name : FacetName] ⊢ ep ≫ ep- (⇒ ν effs)] ...
  #:fail-unless (all-endpoint-effects? #'(effs ...))
                "only endpoint installation effects allowed"
  -------------------------------------------------------------------------------
  [⊢ (install-facet! name (lambda () ep- ...)) (⇒ ν (StartFacet name effs ...))])
```

Line by line, this definition reads the following way:

- The macro applies to syntax that matches the `(start-facet id expr ...+)` pattern.[7] The pattern expects an identifier and a nonempty sequence of endpoint expressions.
- The code between ≫—pronounced “elaborates to”—and the dashed line analyzes and checks these endpoint forms:
  1. The first clause elaborates each endpoint expression, `ep`, individually in an environment extension that records `name` as the name of a facet. This allows Turnstile to resolve each identifier reference within each `ep` to the proper type.
  2. The result of a successful[8] elaboration is bound to the name `ep-`, while the types of effects performed by `ep` are bound to the name `effs`.
  3. The trailing ellipse dictates that each `ep` form is analyzed in this manner.
- The `#:fail-unless` clause enforces the side-condition on the corresponding type-checking rule. It makes sure that the body has only endpoint installation effects, such as from the use of `on`, `assert`, and `field`; it disallows the following effect types: `start-facet`, `stop`, and `spawn`. The actual work is left to a helper function: `all-endpoint-effects?`. If the check fails, the checker signals a type error.
- The clause below the dashed line consists of the macro’s two outputs:
  1. The first output is the target expression that implements the behavior of `start-facet`. The produced expression calls the `install-facet!` procedure, which is provided by the run-time support system for facets. The procedure consumes the name and a thunk that runs the results of elaborating the endpoint forms—in the specified order. These endpoint forms then elaborate to calls to other procedures from the run-time system, and so on.

[7] For expressions that use the keyword `start-facet` and don’t match the pattern `syntax/parse` raises an error with an informative error message.

[8] Any type error discovered while checking an `ep` is reported as soon as it is discovered, aborting the rest of the elaboration of `start-facet`.

2. The second output attaches a type to the elaboration. Here it describes the effect type of the form: `StartFacet`, with endpoint types described by `effs ...`.

Generally speaking, the `define-typed-syntax` macros are almost verbatim transliterations of the typing rules presented in Figure 15.

9.3 Implementing the model checker

A programmer initiates a system verification with respect to some specification by adding the following formula to the program:

```
(verify-actors spec actor-type ...)
```

In terms of Theorem 12, this formula requests a check of $\overrightarrow{T} \models \psi$ where the LTL formula ψ corresponds to `spec`, and T is the series of `actor-types`, one per actor in the dataspace program. If the check succeeds, then the actors of these types jointly realize the specification.

The implementation of `verify-actors` utilizes the model checker SPIN (Holzmann, 1997) to check such uses. That is, it translates each use of `verify-actors` into a Promela program, that is, SPIN's input forms.

More precisely, the translation turns each `actor-type` into a Promela process with a state-machine driven behavior. Each state corresponds to a set of active facets within the actor. An encoding of the dataspace routing algorithm allows processes to react to the appearance and disappearance of assertions, while an additionally generated process performs message dispatching.

SPIN is invoked on this Promela program, plus a translation of the LTL ψ. If verification fails, the trace provided by SPIN is back-translated into the actions of the type-level actors.

Figure 20 displays a sample Promela program. It is the result of translating the `Light` actor's type. The figure annotates the Promela code with in-line explanations. We italicize the annotations to distinguish them from the actual code and comments, which are enclosed within `/*...*/`. The rest of the Promela process follows the same form.

Blocks of code within the actor (lines 15, 23) execute `atomic`ally to avoid any unnecessary interleaving with other actor's initialization in the SPIN search space. Additionally, comments within the code, such as those on lines 16 and 30, embed information from the Racket source program; this information is used to back-translate counterexamples of the SPIN model to the source level.

10 Evaluation

The usefulness of this model-checking approach depends on two primary factors:

- Whether the specification language can express important properties of dataspace communication as they arise in actual programs; and

```
Declare an enumeration type for the possible combinations of active facets:
mtype = {dur_im_lgt_dur, dur_lgt, dur_lgt_dur, lgt, dur_im_lgt}
Create a global variable that controls scheduling:
bool light_proc_clock = true;
Name a Promela process that is active when the program starts:
active proctype light_proc() {
  Use a local variable to track the actor's currently active facets:
  mtype current = lgt;
  Local variables for tracking the assertions known to the actor:
  bool know_WallSwitchOnT_Symbol = false;
  bool know_RoomOccupiedT_Symbol = false;
  bool know_RoomAssignmentT_Symbol_Symbol = false;
  Issue the actor's initial assertions:
  atomic {
    ASSERT(Obs_WallSwitchOnT_Symbol); /*#s(assert #s(Observe ...))*/
  }
  Launch an infinite loop for the actor's ongoing behavior:
  do
    Wait until the actor is enabled:
    :: true ->
      light_proc_clock == GLOBAL_CLOCK;
      atomic {
        light_proc_clock = !light_proc_clock;
        Dispatch based on which facets are active:
        do
          :: current == dur_lgt_dur ->
            Dispatch to the event handlers within the active facets:
            if
              The condition for one of the active event handlers.
              A retracted event fires if there are no matching assertions:
              :: RETRACTED(RoomAssignmentT_Symbol_Symbol) && /*#s(Retracted ...)*/
              and the actor must have seen such a prior assertion:
                 know_RoomAssignmentT_Symbol_Symbol ->
                  Perform effects dictated by the type of the event handler's body:
                  stop a facet by updating current:
                  current = dur_lgt;
                  Update the actor's knowledge of the retracted assertion,
                  and no-longer-relevant ones:
                  know_RoomAssignmentT_Symbol_Symbol = false;
                  know_RoomOccupiedT_Symbol = false;
                  Retract assertions from the stopped facet:
                  RETRACT(Obs_LightOnCmdT_Symbol); /*#s(retract #s(Observe ...))*/
                  RETRACT(Obs_RoomOccupiedT_Symbol); /*#s(retract ...)*/
                  RETRACT(Obs_LightOffCmdT_Symbol); /*#s(retract ...)*/
              The beginning of the next event handler:
              :: ASSERTED(RoomOccupiedT_Symbol) && /*#s(Asserted ...)*/
                 !know_RoomOccupiedT_Symbol ->
                  ...
      }
}
```

Fig. 20. Annotated excerpt of a Promela program for the SPIN model checker

- Whether the implementation can successfully check those properties that are expressible.

This section presents the results of an analysis of these factors in the context of a corpus of facet-based dataspace programs implemented in Racket. Inspecting each program yields a number of behavioral properties important to program correctness. These properties provide the basis of the evaluation.

10.1 Corpus

This section describes the programs in the corpus, as well as several behavioral properties for each program that comprise the subject of the evaluation. One property—deadlock-freedom—is relevant to several examples and thus factored into its own Section (10.2), which discusses the topic in the context of dataspace programs and our behavioral checks. Each program consists of about 500 lines of code, comprising the concurrent core of a larger system.

Smarthome. This program is an extended version of the earlier examples, including a text-based user interface and temperature sensors and controls.

- *Light Presence.* This property describes the desired interaction between presence sensors and lights in the smarthome program. The specification essentially states that once a presence sensor and light have been installed in a room, the light turns on or off as the user moves in or out of the room, respectively.[9] Note that while on the surface this property only talks about light and presence sensor actors, the behavior of those actors depends on additional interactions from the user interface, the actor representation of the user, and the hub actor.
- *Steady State Temperature.* This property specifies the desired behavior of the thermostat in the smarthome: when the user sets a desired temperature, the system gradually heats or cools the home until the target is reached, before turning off.

Data Processing. This program implements the core of a streaming data processing framework, inspired by Flink.[10] Clients submit jobs consisting of some number of interdependent tasks. Several actors collaborate to manage the underlying computational resources for executing tasks, tracking intermediate progress, and delivering the end result.

- *Task Delegation.* This property checks the working of three actors that collaborate to perform tasks by treating them as a black box: each assigned task is eventually performed. Performing a task involves delegating it from the job manager to a task manager, and then from a task manager to a task runner, and finally propagating the results back in the opposite direction.
- *Job Completion.* This is another black-box correctness property: when a client submits a job, the job's results eventually become available. When a client submits a job, a job manager actor analyzes the request and computes a directed, acyclic graph of tasks. It then processes the task graph by assigning tasks as they become ready to other actors, before finally announcing the results of the job. A weak variant of this property states that each job either completes or processes tasks in an infinite loop.
- *Load Balancing.* Task-runner actors are capable of performing one task at a time. A task manager actor monitors the status of task performers and assigns them tasks when they are idle. Likewise, each task manager has a capacity based on the number

[9] This property is a response to a bug uncovered through manual testing, where a light installed in a room after a presence sensor sometimes did not properly react to the initial presence information.

[10] `https://flink.apache.org`.

of task runners it manages. The job manager assigns tasks to task managers based on their current free capacity. This property states that the task manager never assigns a task to a busy runner and the job runner never assigns a task to a task manager that is at capacity. A weak variant of the property pertains to the scenario where each task manager oversees exactly one task runner actor.

Caucus. This program represents a geographically distributed, iterative election in the style of a caucus, inspired by the "two-buyer problem" from the behavioral types literature (Honda *et al.*, 2008). A distinct actor represents each voter and candidate. After a registration period, voting commences across a collection of regions. Within each region, voting proceeds in rounds until a single candidate receives a majority share of the vote. Once every region has reached a decision, the winner of the most regions is pronounced the winner of the election. In each region, an actor guards against various forms of misbehavior from voters and candidates, such as voting twice.

- *Resolution.* The election completes. A weak variant of the property states that either the election completes or holds infinitely many rounds of voting.
- *Candidate (Mis)Behavior.* This property is the specification for a well-behaved candidate actor in the election. A well-behaved candidate actor announces its candidacy as an assertion and maintains it until either the election is over or an election agent informs it that it is no longer in the running.
- *Voter (Mis)Behavior.* This property is the specification for a well-behaved voter actor. A well-behaved voter actor registers in exactly one region and then participates in each round of voting in that region by casting exactly one vote for one of the eligible candidates. A weak variant of the property states that the voter registers and always votes at least once in each round.

Windowing System. This program implements a basic graphical windowing system. A collection of driver actors collaborate to provide the primary interface. A layout-solving actor powers graphical output by combining requested sizes and layout styles (such as vertical or tabular) with the actual dimensions of the system window to compute the actual size and location of each item. Another interface actor provides descriptions of mouse events, including when the mouse is touching a window, mouse presses, and releases. The mouse interface allows for an actor *mix-in* (a function that starts facets or endpoints) to implement drag-and-drop behavior, which can then be instantiated and reused freely. The rest of the program consists of actors implementing rudimentary windowed applications and menus.

- *Layout.* This property describes the behavior of the layout engine, which spawns solvers for horizontal, vertical, and tabular layouts on demand. The specification is based on a comment in the original, untyped program, describing the desired behavior; it essentially demands that requests for layouts are answered with solutions:

```
;;       (Observe LayoutSolution)+ ==>
;;         RequestedLayoutSize ==>
;;           ComputedLayoutSize ∧ LayoutSolution+
```

This comment closely corresponds to the specification language of assertion types and LTL connectives provided by the implementation.

- *Menu Duration.* This property checks the lifetime of menu items: that they appear in response to selecting a menu and that they remain until either a selection is made or a mouse click occurs, either selecting a menu item or closing the menu.

Web Chat. This program implements the server for a chat service loosely based on Slack.[11] It allows a user to connect, sign up for an account, and then create and join conversations. Connected users may request to follow one another. Accepted follow requests lead to each user being added to the contact list of the other. Users may join the conversations of their contacts. The system implements an option for permission delegation, so that if user A chooses to invite user B to a conversation, user B may then invite user C, and so on.

- *Conversation Release.* This property is based on a comment in the original, untyped program, essentially calling for the release of resources in the event that a request is canceled before the response has materialized:

```
;; TODO: CHECK THE FOLLOWING: When the ‘invitation‘ vanishes (due
;; to satisfaction or rejection), this should remove the question
;; from all eligible answerers at once
(during (invitation $cid $inviter $invitee)
  ...)
```

  A weak version of the property relaxes the need for the resources to all be removed "at once."
- *Contact Release.* This property is also based on a similar comment in the code to the conversation release property, but in the module for managing the contacts list for users:

```
;; TODO: CHECK THE FOLLOWING: When the ‘permission-request‘
;; vanishes (due to satisfaction or rejection), this
;; should remove the question from all eligible answerers
;; at once
(during (permission-request $who $grantee ($ p (p:follow _)))
  ...)
```

  A weak version of the property relaxes the need for the resources to all be removed "at once."

10.2 Deadlock freedom

Dataspaces implement a form of asynchronous message-passing, so in a technical sense every dataspace program is free from deadlocks. However, dataspace programs may reach a stuck state where each actor waits for further communication before continuing. Such a situation is sometimes referred to as a soft deadlock. The corpus includes several programs that feature numerous kinds of actors participating in interwoven conversations, making freedom from such soft deadlocks a desirable property to check.

Figure 21 demonstrates a simple program simulating two friends attempting to make plans. One friend actor, once it knows the time of the meeting, is ready to suggest a

[11] https://slack.com.

```
(define (friend1)
  (spawn
    (start-facet f1
      (on (asserted (time $t))
          (start-facet (assert (place "yours")))))))

(define (friend2)
  (spawn
    (start-facet f2
      (on (asserted (place $p))
          (start-facet (assert (time 0)))))))

(run-dataspace (friend1) (friend2))
```

Fig. 21. Soft deadlocked actors.

location. Meanwhile, the other friend actor, is ready to suggest a time depending on the location. The result is no communication at all. Such situations can arise from poorly designed protocols and buggy implementations. Figure 21 is a case of the former.

One way of describing soft deadlocks is that an actor states an interest, that interest is never withdrawn (that is, the interest remains relevant to the actor), and no matching assertion ever arises. The following LTL formula states this property with respect to the `friend1` actor's interest in `Time` assertions:

```
(define-ltl friend1-deadlock
  (Always (Implies ?(TimeT ⋆)
                   (Eventually (Or Time
                                   (Not ?(TimeT ⋆)))))))
```

Checking this LTL property against the implementations of the `friend1` and `friend2` actors fails, as should be expected for Figure 21. The resulting trace illustrates the two actors waiting for one another with active subscriptions but no matching assertions.

A similar property describes the deadlock from the perspective of the `friend2` actor's interest in `Place` assertions, which fails a similar check:

```
(define-ltl friend2-deadlock
  (Always (Implies ?(PlaceT ⋆)
                   (Eventually (Or Place
                                   (Not ?(PlaceT ⋆)))))))
```

Generalizing, a program may be checked for deadlocks by taking each assertion of interest `?`τ and checking the following LTL formula:

```
(Always (Implies ?τ
                 (Eventually (Or τ
                                 (Not ?τ)))))
```

Such a check succeeds if the model checker can show that the program *never* deadlocks. This property may be too strong for the level of type precision. An alternative approach is to state in LTL that the program *definitely* deadlocks, and see if model checking produces a counterexample, implying that the program has some non-deadlocking executions:

```
(Eventually (And (Always ?τ)
                 (Not (Eventually τ))))
```

This section refers to the former property as *strong* (soft) deadlock freedom and the latter as *weak* (soft) deadlock freedom.

Table 1. Evaluation results

Program	Property	Expressible	Checkable
Smarthome	Light Presence	✓	✓
	Steady State Temperature	✓	✓
	Deadlock Freedom	✓	✓
Data Processing	Task Delegation	✓	✓
	Job Completion	✓	
	Job Completion (Weak)	✓	✓
	Load Balancing		
	Load Balancing (weak)	✓	✓
	Deadlock Freedom	✓	✓
Caucus	Resolution	✓	
	Resolution (Weak)	✓	✓
	Cand. Misbehavior	✓	✓
	Voter Misbehavior		
	Voter Misbehavior (weak)	✓	✓
	Deadlock Freedom	✓	✓
Windowing System	Layout	✓	✓
	Menu Duration	✓	✓
Web Chat	Conv. Release		
	Conv. Release (weak)	✓	✓
	Cont. Release		
	Cont. Release (weak)	✓	✓

10.3 Results

Table 1 summarizes the results. Each row of the table describes a property from the corresponding program. A ✓ in the "Expressible" column indicates that the implementation can express the property and a corresponding ✓ in the "Checkable" column indicates that the implementation successfully checked the property. Otherwise, the process of stating and checking could not be completed. Section 10.4 discusses these cases.

Table 2. Running time (seconds) and memory usage (megabytes) of each completed check. The checks ran on a laptop with an Apple M1 Pro CPU, 32GB of memory, and MacOS Ventura 13.5 installed

Program	Property	Time (s)	Mem. (MB)
Smarthome	Light Presence	4.6	285
	Steady State Temperature	14.3	338
	Deadlock Freedom	131.2	333
Data Processing	Task Delegation	2.6	382
	Job Completion (weak)	2.9	467
	Load Balancing (weak)	2.7	467
	Deadlock Freedom	11.2	382
Caucus	Resolution (Weak)	1.6	620
	Cand. Misbehavior	1.7	626
	Voter Misbehavior (weak)	1.8	626
	Deadlock Freedom	6.9	624
Windowing System	Layout	11.0	894
	Menu Duration	11.1	898
Web Chat	Conv. Release (weak)	8.5	1252
	Cont. Release (weak)	1.7	251

10.3.1 Performance

One possibility is that, even if the implementation *can* express and check a desired property, the result may not return within a reasonable amount of time. The potential for a program's state space to grow exponentially is unavoidable. Eventually, even the most optimized checkers will require enormous amounts of memory and face slowdowns. Determining a threshold for a "reasonable" amount of time is tricky, as different developers have different conceptions. This evaluation sets the upper bound for a check that could still be useful at eight hours. That is a long time, but short enough to be a part of continuous integration (CI) tests that run on a nightly basis, with the results ready for review the next morning. Table 2 details the running time and peak memory usage for each successfully checked property.

The performance clears the "integration test" threshold by a wide margin. All of the properties take seconds or a few minutes rather than hours to check. The property that takes the longest to check, deadlock freedom for the smarthome program, consists of 22 individual properties, one for each type of subscription in the program. Each sub-property leads to a separate invocation of the model checker, one for each type of interest in the program. These results put the performance in the realm of "unit test" acceptability, where the checks can be run along with a unit test suite before committing each change to a project.

10.4 Interpretation and discussion

This section discusses each of the properties that the implementation failed to support and the primary reason(s). In a few cases, interesting details for successfully checked properties are provided as well.

Data Processing.

- *Job Completion.* The implementation is not able to check this property. The checker is not able to reason about the termination of the task-processing loop in the protocol. Even though the loop follows an inductive structure—processing a DAG—this information is not propagated to the type level. Moreover, since an LTL formula always talks about all possible executions of a system, the specification language cannot express that it is at least possible to reach the desired end state.
- *Load Balancing.* The implementation is unable to state and check the property. For a task manager with an arbitrary number of associated task runners, stating the specification requires stating relations between numbers. Checking the property would similarly require reasoning about arithmetic operations. Moreover, the property inherently deals with the multiplicity of assertions of a certain types, namely task-assignment assertions. Violating the property means having more than the expected number of such assertions.
  Checking the weak variant of the specification makes use of a coincidence in the program for working around the inability to reason about the multiplicity of assertions. Because there happen to exist two distinct types of tasks, a limited form of multiplicity can be checked—namely, if there are some of one type of task assigned to a manager, there are none of the other, and vice versa.

Caucus.

- *Resolution.* Checking this property also encounters difficulty due to the potential non-termination of a loop in the protocol. The implementation is unable to reason about the termination of the loop, even though the loop matches a simple inductive structure (one candidate is removed each round).
- *Voter (Mis)Behavior.* The implementation lacks the precision to state the full specification. The primary impediments are the inability to correlate information from one assertion to another, such as the name of the candidate the voter chooses with the names of the candidates on the ballot, as well as reason about the multiplicity of assertions. However, the weak specification is still able to distinguish a correct implementation of a voter from several malicious ones.
- *Deadlock Freedom.* While Section 10.2 discusses deadlock freedom, one interesting point to note is that this check uncovered a bug in the program: at the end of each round of voting, a region manager actor announces intermediary results that inform

the candidates of their status in the race, except for the final round of voting, the manager actor simply announces the winner. Due to an oversight, the original implementation of the candidate actors only listened for the intermediate results, not the winner announcement. As a result, they were soft deadlocked waiting for an intermediate update that never materializes. The fix to the bug is to have the candidate actor listen and react to the final election result.

Web Chat.

- *Conversation Release.* This property is checkable with the implementation, with one caveat: the phrase "at once" entails a level of precision that is beyond the implementation. That is, it can check that the required assertions disappear, but it does not have a way of expressing that the concerned parties, "all eligible answerers," all react within a set timeframe. However, that part of the specification is of dubious importance when scrutinized: dataspace routing will always dispatch an event to all interested actors when the `question` is withdrawn. Each such actor will have the opportunity to react to the event in due time. Since dataspace scheduling is fair, there is little reason to worry about such timing details. Thus, the full specification is beyond the implementation, but the weak version, without the timing constraint, is checkable.
- *Contact Release.* Just as with the conversation release property, the "at once" phrasing creates difficulty but is of questionable importance. So again, the full specification is not checkable but a weak variant without the timing constraint is.

Threats to Validity. There are several threats to validity of this evaluation. The corpus comprises a small number of programs, authored by only three individuals. The programs for the corpus were not selected randomly. Rather, we sought programs whose implementations involved a high degree of concurrency and communication, as opposed to business logic or other concerns. Moreover, the identification of properties and subsequent verification efforts were conducted by the authors of the tools. We are highly familiar with both the capabilities and limitations, potentially biasing the search for properties that are a good match for the current capabilities.

11 Related work

The design of the facet language has two distinct goals. First, the notation greatly facilitates the expression of the temporal relationships with respect to—their conversations with—other actors. See Appendix A for an extended comparison of facets to an implementation of the same behavior in a functional style. Second, in contrast to higher order functional programming of actors, the first-order character of the facet language allows the construction of reasoning tools that assist programmers with the validation of essential protocol properties. That is, the facet language forms capture the input/output behavior of an actor without the need for additional data- or control-flow analysis. See Section 10 for a basic illustration. Our results are thus related to two distinct bodies of research: design of concurrent languages (Section 11.1) and reasoning with automated behavioral types (Section 11.2). The dissertations of Garnock-Jones (2017) and Caldwell (2023) also feature extended discussions of related languages and analysis tools.

11.1 Facets and the design of concurrent languages

Programming languages provide limited support for organizing conversations among groups of actors and their internals. Garnock-Jones et al. (2014, 2016) compare dataspace communication with related coordination mechanisms such as the Conversation Calculus (Vieira *et al.*, 2008), the Mobile Ambient Calculus (Cardelli & Gordon, 2000), the join calculus (Fournet & Gonthier, 2000), and tuplespaces (Carriero *et al.*, 1994; Murphy *et al.*, 2006), as well as various actor systems and process calculi. Here, we focus on linguistic features for organizing the code of individual actors.

State-Machine Actors. Actor systems such as Akka (Akka Project, 2022) and Erlang/OTP (Armstrong, 2003) provide means of organizing actors as state machines. Erlang's `gen_statem` interface exemplifies these mechanisms. The programmer describes each state in the machine as a procedure mapping an incoming message to the next state and some actions to carry out. These state transition functions also operate on the actor's private store, handled separately from the state of the machine.

Actors organized as event-driven state machines lose access to contextual information. Instead, such notations force programmers to encode context into each state and save related information explicitly in the private store. Such encodings, however, make it cumbersome to create actors that simultaneously participate in multiple conversations or alternatively engage and disengage in multiple intertwined behaviors, such as the `Hub` and `Light` actors from Section 3.

The `gen_statem` interface and its sibling `gen_server` address only one of the concerns of facets in a limited fashion: abstracting control state. Language support ends at instantiating the interface with callbacks. An actor that deals with multiple types of messages in a single state must often manually demultiplex the message to match it with the proper response. Callbacks use the familiar state-passing style, operating on a monolithic store. Though callbacks may coordinate startup and shutdown of the actor, initiating and closing a conversation comes without linguistic support.

Fact Spaces and CRIME. The inspirational *fact spaces model* (Mostinckx *et al.*, 2007) shares many similarities with the dataspace model. Fact spaces build on TOTAM (Scholliers *et al.*, 2009, 2010; González Boix *et al.*, 2014), a form of tuplespace, to equip actors with a means of moving and sharing state. In fact spaces, programs react to both the appearance and disappearance of facts from a shared repository. Reactions are described in a logic coordination language, allowing the computation of new facts based on current facts via forward-chaining. The language implementation (implicitly) records the dependencies between facts and invokes application actions specified by logic rules.

The implementation of the fact space model, dubbed CRIME (Mostinckx *et al.*, 2008), integrates the fact spaces model with the AmbientTalk language (Van Cutsem *et al.*, 2007, 2014). AmbientTalk is an instance of the Actor model (Agha, 1986; De Koster *et al.*, 2016) in the mold of E (Miller *et al.*, 2005). In E and AmbientTalk, objects are organized into *vats*; each vat runs its own event loop, dispatching incoming messages to the contained objects. Combining AmbientTalk and CRIME requires bridging the gap between the events corresponding to assertion and retraction of facts and the message exchange of AmbientTalk. The solution incorporates reactive programming, in the mold of FrTime (Cooper & Krishnamurthi, 2006). The result is that actors may define time-varying *behaviors*, with values shifting based on the available facts.

The differences between CRIME and facet-oriented dataspace actors stem from the absence of a unifying notion of a conversational frame. CRIME comes without any representation of a conversation, hence programs lack the conversational structure of (nested) facets. Time-varying collections do not offer a way of propagating changes to the tuples in the shared space, unlike the fields and query forms that connect to an actor's endpoints. Because CRIME does not group the components of conversational behavior, no automatic support exists for the release of associated resources. Programmers must carefully reason about the relationships between components to ensure that the state of the actor and associated tuples remain consistent. The underlying E language offers object references to denote specific conversations within the heap of a given actor and employs method dispatch as a limited pattern matcher over received messages.

Concurrency Within Active Objects. Some Active Objects languages allow for concurrency *within* an Actor or Actor-like entity (de Boer *et al*., 2007; Schäfer & Poetzsch-Heffter, 2008, 2010). Typically, this concurrency is achieved by allowing multiple message dispatches to be active within a single active object. This allows for splitting control and state among numerous objects, resembling the way our facets and fields individually contribute to the actor's overall behavior.

Much like with conventional object-oriented programming, communication between active objects forms a graph structure. Each active object is a node in the graph and has an edge to each object it communicates with via asynchronous method invocation. Indeed, many active object languages support both the communication-level graph and the traditional object-oriented "refers to" relationship with support for passive objects (Boer *et al*., 2017). Each active object (node in the communication graph) may implement its behavior using any number of such passive objects following traditional object-oriented design (Gamma *et al*., 1994).

Dataspaces and faceted actors possess the same graph-like communication structure. Each actor (node) is connected to each other actor that it communicates with via assertions (edge). Thinking in terms of the graph abstracts away the differences between the message-passing paradigms employed by active objects (point-to-point) and dataspaces (publish/subscribe). Just as how an active object may implement its behavior using any number of passive objects with their own fields and method definitions, a facet-based actor defines its behavior across a collection of facets, fields, and endpoints. The primary difference is that passive objects form a graph, while facets form a (dynamically changing) tree. The tree structure of facets enables the ability to program directly to the concerns of startup and, especially, shutdown (via the stop statement and on stop event handlers).

One interpretation of this analysis is that it may be fruitful to adapt the facet notation to the domain of active objects. Combining facets for defining *behavior* with passive objects for defining *state* has the potential to synthesize the best of both worlds.

ConGoLog. While communication-by-assertion in dataspaces is inspired by Prolog fact databases, ConGoLog (De Giacomo *et al*., 2000) is a more direct adaptation of logic programming to concurrency. In ConGoLog, a programmer first models a particular domain—typical applications include robotic control systems—with a collection of axioms describing relevant properties and the effects of performing particular actions. The main program then specifies some number of concurrent agents—as logic programs—emitting effects, each of which may update the stated properties, and reacting to the truthiness of such properties.

ConGoLog and dataspaces share a deep connection but also differ in significant ways with respect to their conceptions of state and agent. As a variant of the actor model, dataspaces provide each actor with its own isolated memory, while the dataspace itself contains information shared with the rest of the program. Information shared in the dataspace is tied to the identity of some particular actor, imposing a rigid structure on updates and a connection to component failure. Meanwhile, ConGoLog lacks the distinction between local state of an agent and globally visible state information of the situation. Concurrent ConGoLog agents come without the structure provided by facets for grouping related behavior and state, as well as orchestrating startup, shutdown, and component evolution—partly because these concerns are not a goal of the design.

Sparrow. The Sparrow DSL (Avila *et al.*, 2020) extends Erlang with the ability to react to complex event patterns. That is, rather than just handling one message at a time from its mailbox, a programmer may utilize Sparrow to specify an actor's behavior dependent on particular combinations of messages, timing constraints, and so on. Reactions to a given pattern may be dynamically added to and removed from an actor's behavior. A reaction is like a facet with a single event-handler endpoint. They lack the other features of our language, like the hierarchical structure, start-up and shut-down behavior, and grouping of facets, as well as localized state of fields.

Dataflow. The simple dataflow system described here is most similar to the simple dependency tracking approach to object-oriented reactive programming described by Salvaneschi & Mezini (2014, section 2.3) and was in fact directly inspired by the dependency tracking of JavaScript frameworks such as Knockout[12] (Sanderson, 2010) and Meteor.[13]

11.2 Behavioral type systems

In general, the design and theory of a type system is intimately tied to the underlying language. This is doubly true of behavioral type systems, which express control-flow aspects of the program, not just the static relationships found in purely structural type systems. Consequently, applying ideas around the actor model to dataspaces and facet-oriented actors poses a serious challenge. Dataspaces provide a form of publish/subscribe communication (Eugster *et al.*, 2003) that has received little attention in the context of behavioral type systems. While encoding dataspaces in a traditional message-passing (or channel-based) setting is possible, the encoding would obscure a program's communication patterns too much for the targeted behavioral type system to be of use. Application-specific information could potentially be recovered by utilizing more powerful

[12] `http://knockoutjs.com/`.
[13] `https://docs.meteor.com/api/tracker.html`.

type-level reasoning, such as dependent session types (Toninho *et al.*, 2011), but presently bringing such machinery to bear in a usable programming language remains unsolved.

Our work specifically draws inspiration from studying the approach to type checking more so than the specifics of the systems. In that sense, our design follows that of Igarashi & Kobayashi (2001) and Chaki *et al.* (2002), in which the type checker constructs type-level processes as abstractions for term-level behavior. The type-level processes serve as the basis for behavioral analysis. In the case of Igarashi & Kobayashi (2001), the checks are generic. Subtyping and other parts of the type checker may be instantiated to yield checks for races, deadlocks, etc. Chaki *et al.* (2002) use the SPIN model checker for simulation-based subtyping and analyzing conformance to LTL specifications. In their case, LTL formulae state properties of channels in the system.

Effpi. The Effpi message-passing framework (Scalas *et al.*, 2019) reflects core process and communication operations to the type level and uses dependent function types to precisely track which channels are used. By model checking these type-level descriptions, it becomes possible to verify certain properties stated in a temporal logic, including deadlock-freedom and some communication patterns. By contrast, we seek to verify any LTL formula, allowing application-specific correctness properties. The biggest difference to our work is the communication paradigm. While Effpi uses message-passing along channels in a process calculus, our actors share knowledge via a dataspace.

Conversation Types. Conversation types (Caires & Vieira, 2009) add behavioral checking to the Conversation Calculus (Vieira *et al.*, 2008). A conversation type describes a sequence of message exchanges within a particular context, i.e., conversation. Like the global types of Honda *et al.* (2008), a conversation type may be decomposed to type(s) describing the actions of the individual processes participating in the conversation. Crucially, the decomposition is flexible. A given conversation type may be realized as the composition of numerous different combinations of participant types. This flexibility meshes well with the anonymous communication in the conversation calculus, allowing a degree of agnosticism with regard to the number of participants in a conversation and their individual roles when looking from the global perspective. At first glance, this approach may work for dataspace actors, but because every communication is between a single sender and a single receiver, the model remains similar to channel-based models rather than publish/subscribe communication. In addition to structural message safety, their types ensure a degree of deadlock-freedom.

Active Object Languages. There are several notable behavioral type systems and efforts to perform static verification on active object languages. Behavioral type systems for active objects tend to focus on the problem of detecting deadlocks (Henrio *et al.*, 2017). They employ a similar types-as-processes technique, capturing key information, such as dependencies between futures, as effect types. Proof rules (i.e., type checking) can then determine whether the program may deadlock. Our behavioral checks of facet programs instead treat effect types as abstracted, but still executable, processes, with the goal of both (model)checking properties of programs and validating the design of the DSL.

The Rebeca modeling language (Sirjani *et al.*, 2004) is the closest to our own behavioral checking. With Rebeca, a programmer may use communication and abstraction facilities of active objects to define a modes of their program or system. The Rebeca model checker, Modere (Jaghoori *et al.*, 2006), can then verify properties of the model specified as temporal logic formulae. Rebeca has numerous extensions, notably broadcasting (Yousefi *et al.*, 2015). The goal of our work is to check properties of programs versus models of programs and to thus validate the design of the facet DSL. It might be possible to target Rebeca as a backend for the implementation, rather than SPIN, and benefit from its message-passing-specific optimizations.

ConGoLog. In bounded contexts, it is possible to decide some temporal properties of ConGoLog programs (De Giacomo *et al.*, 2016). Thus, it ought to be possible to develop a model checker based on these decision procedures that could express and check temporal properties similar to the ones that we explore for dataspace programs. No such model-checker implementation appears in the literature.

Types for the Join Calculus. The join calculus (Fournet & Gonthier, 2000) shares some similarities with dataspace actors. It has a soup of messages versus the table of assertions in a dataspace. The event handler endpoints of facets resemble join patterns, especially in the objective join calculus (Fournet *et al.*, 2000). Recent work (Crafa & Padovani, 2017) has developed behavioral types for the join calculus around the notion of type state (Strom & Yemini, 1986). They describe the types of messages understood by an object and use connectives such as conjunction, disjunction, and repetition to determine how a reference to an object can and must be used. The resulting design is able to elegantly track the dynamic interface of, e.g., a lock object. Unfortunately, they note the challenge of implementing such connectives, as well as the necessary substructural support.

Mailbox Types. Similarly to the behavioral type systems for the Join Calculus, mailbox types (de'Liguoro & Padovani, 2018) use a commutative regular expression of messages to describe the inputs to an actor. Though the description of messages is unordered, sequencing is still expressible using mailbox-passing. Following a familiar technique in behavioral systems, type analysis collects information from the program as the basis for analysis. In this case, it is a dependency graph among mailboxes. The dependency graphs are vital for detecting and preventing programs that may deadlock.

Session Types. Type systems for the π-calculus and related process calculi have been widely studied. Session type systems (Honda *et al.*, 1998) in particular have been utilized to describe and verify the communication properties of many kinds of systems. Our methods are closely related to the generic π-calculus type system of Igarashi & Kobayashi (2001). They relate processes with abstract process types, including the primary process constructors: channel send and receive, parallel composition, and so on. Their system is parametric over constraints on process types, allowing different instantiations targeted for different properties, such as deadlock prevention.

Multiparty session types (Honda *et al.*, 2008; Scalas & Yoshida, 2019) come closer to describing the group oriented communications between dataspace actors. Global types provide a perspective for describing the protocol among a group of participants. Though efforts toward adapting session type theories to actor systems have found some success (Mostrous & Vasconcelos, 2011; Crafa, 2012; Neykova & Yoshida, 2014), much work remains to be done to handle the full spectrum of actor programming and its relatives, such as the dataspace model.

12 Conclusion

The notation of facets for dataspace actors is good for concurrent actor programming. Our examples show how the notation directly addresses the most common concerns: engaging and disengaging in behavior, managing local state and resources, sharing information during a conversation, etc. Furthermore, the language design captures the key components of behavior, allowing a types-as-processes approach to analyzing temporal behavior.

The facet notation is also good for reasoning about programs. A type system can ensure basic soundness and provides the basis of behavioral verification. The evaluation on realistic programs illustrates the system's usefulness. Our formal work proves that this approach is well-founded; our evaluation supplies initial evidence of the practicality of the approach.

The work itself suggests several directions for future improvements. Concerning the facet notation, the design could benefit from Avila *et al.* (2020)'s language of communication patterns. Concerning the behavioral type system, precision could be improved by using a dependency notation for correlation information in assertions similar to the work of Scalas *et al*. (2019) for channel names. Finally, an additional validation could come from porting our work to a widely used actor system, such as Akka. The broadened context would introduce additional research challenges, such as accounting for the possibility of communication failures, as well as shed additional light on the usefulness and usability of our design.

Acknowledgments

We would like to thank Mitch Gamburg for his assistance in developing programs for the evaluation.

Financial support

This work was partially supported by NSF grant SHF 2315884.

Conflicts of interest

None.

References

Agha, G. (1986) *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press. Massachusetts.

Akka Project. (2022) Akka actors. `https://akka.io/`. Accessed: 2022-06-24.

Armstrong, J. (2003) *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD dissertation. Royal Institute of Technology, Stockholm.

Avila, H. R., De Koster, J. & De Meuter, W. (2020) Advanced join patterns for the actor model based on CEP techniques. *Art Sci. Eng. Program*. **5**(2), Article 10.

Berry, G. & Gonthier, G. (1992) The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program*. **19**(2), 87–152.

Boer, F. d., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C. C., Johnsen, E. B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K. & Yang, A. M. (2017) A survey of active object languages. *ACM Comput. Surv*. **50**(5), 1–39, Article 76.

Caires, L. & Vieira, H. T. (2009) Conversation types. In *Proceedings of the European Symposium on Programming (ESOP)*.

Caldwell, S. (2023) *Reasoning About Actors that Share State*. PhD dissertation. Northeastern University.

Caldwell, S., Garnock-Jones, T. & Felleisen, M. (2020) Typed dataspace actors. *J. Funct. Program*. **30**, e27.

Carbone, M. & Montesi, F. (2013) Deadlock-freedom-by-design: Multiparty asynchronous global programming. In Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).

Cardelli, L. & Gordon, A. D. (2000) Mobile ambients. *Theoret. Comput. Sci*. **240**(1), 177–213.

Carriero, N. J., Gelernter, D., Mattson, T. G. & Sherman, A. H. (1994) The Linda alternative to message-passing systems. *Parallel Comput*. **20**(4), 633–655.

Chaki, S., Rajamani, S. K. & Rehof, J. (2002) Types as models: Model checking message-passing programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Chang, S., Knauth, A. & Greenman, B. (2017) Type systems as macros. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Clocksin, W. & Mellish, C. (1981) *Programming in Prolog*. Springer.

Cooper, G. H. & Krishnamurthi, S. (2006) Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the European Symposium on Programming (ESOP)*.

Crafa, S. (2012) *Behavioural Types for Actor Systems*. Technical report. https://arxiv.org/abs/1206.1687.

Crafa, S. & Padovani, L. (2017) The chemical approach to typestate-oriented programming. In *ACM Transactions on Programing Languages and Systems*, vol. 39, 1–45. A preliminary version appeared in the proceedings of OOPSLA'15.

Culpepper, R. & Felleisen, M. (2010) Fortifying macros. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Culpepper, R. & Felleisen, M. (2012) Fortifying macros. *J. Funct. Program*. **22**, 439–476.

de Boer, F. S., Clarke, D. & Johnsen, E. B. (2007) A complete guide to the future. In *Proceedings of the European Symposium on Programming (ESOP)*.

De Giacomo, G., Lespérance, Y. & Levesque, H. J. (2000) ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell*. **121**(1–2), 109–169.

De Giacomo, G., Lespérance, Y., Patrizi, F. & Sardina, S. (2016) Verifying ConGolog programs on bounded situation calculus theories. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

De Koster, J., Van Cutsem, T. & De Meuter, W. (2016) 43 years of Actors: A taxonomy of Actor models and their key properties. In *Proceedings of the International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE)*.

de'Liguoro, U. & Padovani, L. (2018) Mailbox types for unordered interactions. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Demetrescu, C., Finocchi, I. & Ribichini, A. (2011) Reactive imperative programming with dataflow constraints. In *Proceedings of the ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

Eugster, P. T., Felber, P. A., Guerraoui, R. & Kermarrec, A.-M. (2003) The many faces of publish/subscribe. *ACM Comput. Surv*. **35**(2), 114–131.

Felleisen, M. (1988) The theory and practice of first-class prompts. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Flatt, M., Yu, G., Findler, R. & Felleisen, M. (2007) Adding delimited and composable control to a production programming environment. In Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP).

Floyd, R. W. (1967) Assigning meaning to programs. In *Proceedings of Symposium on Applied Mathematics*, vol. 19.

Fournet, C. & Gonthier, G. (2000) The join calculus: A language for distributed mobile programming. *Appl. Semant. Summer School*.

Fournet, C., Laneve, C., Maranget, L. & Rémy, D. (2000) Inheritance in the join calculus. In Foundations of Software Technology and Theoretical Computer Science.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. Massachusetts.

Garnock-Jones, T. (2017) *Conversational Concurrency*. PhD dissertation. Northeastern University.

Garnock-Jones, T. & Felleisen, M. (2016) Coordinated concurrent programming in Syndicate. In *Proceedings of the European Symposium on Programming (ESOP)*.

Garnock-Jones, T., Tobin-Hochstadt, S. & Felleisen, M. (2014) The network as a language construct. In *Proceedings of the European Symposium on Programming (ESOP)*.

González Boix, E., Scholliers, C., De Meuter, W. & D'Hondt, T. (2014) Programming mobile context-aware applications with TOTAM. *J. Syst. Softw*. **92**(1), 3–19.

Henrio, L., Laneve, C. & Mastandrea, V. (2017) Analysis of synchronisations in stateful active objects. In Integrated Formal Methods.

Hewitt, C., Bishop, P. & Steiger, R. (1973) A universal modular actor formalism for artificial intelligence. In International Joint Conference on Artificial Intelligence.

Hoare, C. A. R. (1969) An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580.

Holzmann, G. (1997) The model checker SPIN. *IEEE Trans. Softw. Eng*. **23**(5), 279–295.

Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming (ESOP)*.

Honda, K., Yoshida, N. & Carbone, M. (2008) Multiparty asynchronous session types. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Igarashi, A. & Kobayashi, N. (2001) A generic type system for the Pi-calculus. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Jaghoori, M. M., Movaghar, A. & Sirjani, M. (2006) Modere: The model-checking engine of Rebeca. In Proceedings of the ACM Symposium on Applied Computing (SAC).

Lucassen, J. M. & Gifford, D. K. (1988) Polymorphic effect systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Manna, Z. & Pnueli, A. (1991) *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag.

Miller, M. S., Tribble, E. D. & Shapiro, J. (2005) Concurrency among strangers. In International Symposium on Trustworthy Global Computing, Edinburgh, Scotland.

Mostinckx, S., Lombide Carreton, A. & De Meuter, W. (2008) Reactive context-aware programming. In Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS), vol. 10, Electronic Communications of the EASST, DisCoTec.

Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C. & De Meuter, W. (2007) Fact spaces: Coordination in the face of disconnection. In International Conference on Coordination Languages and Models.

Mostrous, D. & Vasconcelos, V. T. (2011) Session typing for a featherweight Erlang. In International Conference on Coordination Languages and Models.

Murphy, A. L., Picco, G. P. & Roman, G.-C. (2006) LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol*. **15**(3), 279–328.

Neykova, R. & Yoshida, N. (2014) Multiparty session actors. In International Conference on Coordination Languages and Models.

Pnueli, A. (1977) The temporal logic of programs. In Symposium on Foundations of Computer Science (SFCS).

Rozier, K. Y. (2011) Linear temporal logic symbolic model checking. *Comput. Sci. Rev.* **5**(2), 163–203.

Salvaneschi, G. & Mezini, M. (2014) Towards reactive programming for object-oriented applications. In Transactions on Aspect-Oriented Software Development.

Sanderson, S. (2010) Introducing Knockout, a UI library for JavaScript. http://blog.stevensanderson.com/2010/07/05/introducing-knockout-a-ui-library-for-javascript/. Accessed: 2022-07-25.

Scalas, A. & Yoshida, N. (2019) Less is more: Multiparty session types revisited. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Scalas, A., Yoshida, N. & Benussi, E. (2019) Verifying message-passing programs with dependent behavioural types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Schäfer, J. & Poetzsch-Heffter, A. (2008) CoBoxes: Unifying active objects and structured heaps. In Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science.

Schäfer, J. & Poetzsch-Heffter, A. (2010) JCoBox: Generalizing active objects to concurrent components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science.

Scholliers, C., González Boix, E. & De Meuter, W. (2009) TOTAM: Scoped tuples for the Ambient. In Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS).

Scholliers, C., González Boix, E., De Meuter, W. & D'Hondt, T. (2010) Context-aware tuples for the Ambient. In *Proceedings of On the Move to Meaningful Internet Systems*, OTM'10.

Sirjani, M., Movaghar, A., Shali, A. & de Boer, F. S. (2004) Modeling and verification of reactive systems using Rebeca. *Fundam. Inform*. **63**(4), 385–410.

Strom, R. E. & Yemini, S. (1986) Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng*. SE-12(1), 157–171.

Tasharofi, S., Dinges, P. & Johnson, R. E. (2013) Why do Scala developers mix the actor model with other concurrency models? In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Toninho, B., Caires, L. & Pfenning, F. (2011) Dependent session types via intuitionistic linear type theory. In International Symposium on Principles and Practice of Declarative Programming (PPDP).

Van Cutsem, T., Gonzalez Boix, E., Scholliers, C., Lombide Carreton, A., Harnie, D., Pinte, K. & De Meuter, W. (2014) AmbientTalk: Programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct*. **40**(3-4), 112–136.

Van Cutsem, T., Mostinckx, S., González Boix, E., Dedecker, J. & De Meuter, W. (2007) AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In International Conference of the Chilean Society of Computer Science.

Vardi, M. Y. (2001) Branching vs. linear time: Final showdown. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS).

Vieira, H. T., Caires, L. & Seco, J. C. (2008) The conversation calculus: A model of service-oriented computation. In *Proceedings of the European Symposium on Programming (ESOP)*.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput*. **115**(1), 38–94.

Yi, K. & Ryu, S. (2002) A cost-effective estimation of uncaught exceptions in standard ML programs. *Theoret. Comput. Sci*. **277**(1–2), 185–217.

Yousefi, B., Ghassemi, F. & Khosravi, R. (2015) Modeling and efficient verification of broadcasting actors. In *Fundamentals of Software Engineering*.

A Comparing Facet-based and Functional Actors

This section provides additional motivation for the facet notation via a comparison to the results of implementing actor behaviors in a procedural manner.

Figure A1 implements the behavior of the `light-actor` (Figure 4) using a procedural interface to the Racket dataspace library. In the procedural interface, the behavior of an individual actor is specified as a state transducer function.

The `BEHAVIOR` function (lines 3–8) implements this interface. Its inputs are a representation of an `event` and the `current-state` of the actor. The body of the `BEHAVIOR` function dispatches to one of several sub-behavior functions for the actor. The `current-state` is an instance of the `light-actor-state struct` containing two pieces of information. The first is the actor's current control state, dictating which conversation(s) to engage in response to an event. The actor's interactions are simple enough that its control may be encoded as a switch.[14] The second holds any additional data relevant to the active sub-behavior.

Placing the `light-actor` into a dataspace as an instance of an `actor struct` (lines 9–11) requires specifying the initial `light-actor-state` as well as the actor's initial assertions, establishing interest in receiving `wall-switch` information. Note that the actor's `ctrl-state`—`UNPOWERED`—is expressed separately but must be in sync with the initial assertion(s).

The `unpowered`, `powered`, and `located` sub-routines implement the actor's various sub-behaviors. Each routine computes a `transition-to struct` as its result. The first slot of the `transition-to` specifies a new private state for the actor, while the second requests updates to the actor's current assertions. The updates are a list, with each item calling for the introduction of a new assertion (`assert`) or withdrawal of an existing one (`retract`).

The `unpowered` routine (lines 13–19) responds to notification that the `wall-switch` is `ON` with a `transition-to` the `POWERED` state. In the `POWERED` state, the actor uses the `powered-state struct` as its private state to keep track of whether the light is currently `ON` or `OFF`. Additionally, it introduces the `light` assertion and expresses interest in a `room-assignment`.

The `powered` routine (lines 21–32) implements two possibilities. The first is the event that the wall switch is turned off (line 23). The resulting transition returns to the `"unpowered"` state, `retract`ing the `powered`-specific assertions. The second possibility is the appearance of a `room-assignment` assertion (line 28). In that case, the actor transitions to the `LOCATED` state. The private state of the actor must then keep track of both the current `ON/OFF` status of the light as well as the assigned room. The `located-state struct` serves this purpose. Finally, the actor introduces a new interest in presence sensor information for the assigned room.

[14] An actor engaged in truly complex conversations requires a more sophisticated encoding or the use of (delimited) continuations. Note that while it may seem attractive to represent the control state as a function, allowing the body of `BEHAVIOR` to simply be the application of the `ctrl-state` to `event`, that approach makes composing the behaviors of different states more difficult, as when the `located` behavior inspects the desired `ctrl-state` of the `powered` behavior (lines 40–41). While additional programming patterns may be employed to address this deficiency, the facet notation is the result of considering such patterns and designing a DSL around them.

```
1  (define (spawn-light wall-switch-id)
2    (define my-id (generate-unique-id "light"))
3    (define (BEHAVIOR event current-state)
4      (match-define (light-actor-state ctrl-state data) current-state)
5      (match ctrl-state
6        [UNPOWERED (unpowered event current-state)]
7        [POWERED (powered event current-state)]
8        [LOCATED (located event current-state)]))
9    (define STATE0 (light-actor-state UNPOWERED (unpowered-state)))
10   (define ASSERTIONS0 (assertions (? (wall-switch wall-switch-id ON))))
11   (actor BEHAVIOR STATE0 ASSERTIONS0))
12
13 (define (unpowered event current-state)
14   (match event
15     [(asserted (wall-switch _ ON))
16      (transition-to (light-actor-state POWERED
17                                        (powered-state ON))
18                     #:with-updates (list (assert (light my-id ON))
19                                          (assert (? (room-assignment my-id ⋆)))))]))
20
21 (define (powered event current-state)
22   (match event
23     [(retracted (wall-switch _ ON))
24      (transition-to (light-actor-state UNPOWERED
25                                        (unpowered-state))
26                     #:with-updates (list (retract (light my-id ⋆))
27                                          (retract (? (room-assignment my-id ⋆)))))]
28     [(asserted (room-assignment my-id $room))
29      (transition-to (light-actor-state LOCATED
30                                        (located-state (powered-state-on? current-state)
31                                                       room))
32                     #:with-updates (list (assert (? (in-room room 0)))))]))
33
34 (define (located event current-state)
35   (match-define (located-state on? room) current-state)
36   (define powered-transition (powered event (powered-state on?)))
37   (match powered-transition
38     [(transition-to (light-actor-state UNPOWERED _) $updates)
39      (transition-to (light-actor-state UNPOWERED (unpowered-state))
40                     #:with-updates (append updates
41                                            (list (retract (? (in-room room 0))))))]
42     [_
43      (match event
44        [(asserted (in-room room 0))
45         (transition-to (light-actor-state LOCATED (located-state OFF room))
46                        #:with-updates (list (retract (light my-id ⋆))
47                                             (assert (light my-id OFF))))]
48        [(retracted (in-room room 0))
49         (transition-to (light-actor-state LOCATED (located-state ON room))
50                        #:with-updates (list (retract (light my-id ⋆))
51                                             (assert (light my-id ON))))])]))
```

Fig. A1. Procedural light actor.

The `located` routine (lines 34–54) must still react to the withdrawal of `wall-switch` assertions, in the manner of `powered`. To accomplish this goal, it invokes the `powered` function on the event and a synthesized instance of `powered-state`. A procedure-based actor must determine how to combine the state-changes and actions from each of its conversations on a case-by-case basis. Here, it inspects the resulting transition. If the transition is to the `"unpowered"` state, the `located` function augments the transition with the `retraction` of interest in presence sensor assertions. Otherwise, the actor ignores the transition and proceeds by analyzing the `event`. Depending on the presence sensor information, the actor either updates the light's private state and public assertion to `ON` or `OFF`.

Analysis. The facet-based implementation enjoys a number of advantages.

Spatial locality. The procedural actor describes its initial assertion in one place and updates to its assertions throughout different locations in the code. The actor manipulates its `light` assertion in four different locations (lines 18, 26, 46, and 50). The problem is not unique to assertion-manipulating dataspace actors. Consider the code for a networked actor. The code for connecting is typically in one place, while code for using the connection or tearing it down is somewhere else. By contrast, in the facet language, an actor accomplishes the same behavior with a single endpoint—which is less error prone than when functionality is distributed over many locations.

Conversational structure. The organization of the facet code directly reflects the relationships among the actor's conversations. The interaction with the presence sensor takes place as a sub-conversation in the context of a `room-assignment` assertion, which is dependent on the conversation with the wall switch. Recovering the same relationships from the implementation of the procedural actor requires a careful inspection of its code.

Localized State and *Automatic Demultiplexing.* Facet-based actors define state item-by-item and in the relevant context, and dataflow keeps public and private state automatically in sync. Likewise, the control state of the facet-based actor is maintained by the language implementation, which automatically routes incoming assertion patches to the (pattern)matching endpoints.

The contrast between the code in a facet notation and a procedural language is large, even when the latter is enriched with imperative assignment or objects. For simple conversations, switches like those in the light actor combined with assignments or objects will simplify the procedural code. Complications arise for an actor conducting parallel conversations. While each conversation's state could be encapsulated in an object, doing so would have the unfortunate consequence that the code for manually demultiplexing incoming events and the code for maintaining a local view would be at separate places.

Similarly, a procedural language forces an actor to maintain its control state explicitly. In our concrete example, the `ctrl-state` field is the symptom of this problem. Continuations (delimited or unconstrained) (Felleisen, 1988; Flatt *et al.*, 2007) do not solve the problem either, as code (or patterns) for managing the continuations must still be deployed to keep track of the state.

Brevity. Finally, the facet-based implementation requires far fewer lines of code.

B Facet Metafunctions

The semantics of Section 5.2 refers to several secondary metafunctions. This section provides their formal definitions.

The $\mathit{match}_{\mathrm{D}}$ function matches an event against an event descriptor. Successful matches yield a set of substitutions for the binding variables of the event descriptor's pattern:

$$\begin{array}{rcll}
\mathit{match}_{\mathrm{D}} & : & \mathrm{D} \times \mathrm{Evt} \times \pi \times \pi \times \sigma \xrightarrow{\mathit{partial}} \mathcal{S} & \\
\mathit{match}_{\mathrm{D}}(\texttt{start},\ \texttt{start},\ \pi,\ \pi',\ \sigma) & = & \{\emptyset\} & \\
\mathit{match}_{\mathrm{D}}(\texttt{stop},\ \texttt{stop},\ \pi,\ \pi',\ \sigma) & = & \{\emptyset\} & \\
\mathit{match}_{\mathrm{D}}(\texttt{asserted}\ \mathrm{e},\ \pi^{+}/\pi^{-},\ \pi,\ \pi',\ \sigma) & = & \mathit{project}(\mathrm{v},\ \pi,\ \pi',\ \pi^{+}) & \text{if } \mathrm{v} = \mathit{eval}(\mathrm{e},\ \sigma) \\
\mathit{match}_{\mathrm{D}}(\texttt{retracted}\ \mathrm{e},\ \pi^{+}/\pi^{-},\ \pi,\ \pi',\ \sigma) & = & \mathit{project}(\mathrm{v},\ \pi,\ \pi',\ \pi^{-}) & \text{if } \mathrm{v} = \mathit{eval}(\mathrm{e},\ \sigma)
\end{array}$$

The *project* function handles matching a pattern against a set of assertions, yielding a set of substititutions. The set is empty when there are no matching assertions in the set:

$$\begin{array}{rcl} \mathit{project} & : & \mathrm{v} \times \pi \times \pi \times \pi \longrightarrow \mathcal{S} \\ \mathit{project}(\mathrm{v}, \pi_i, \pi_i', \pi) & = & \{\mathit{match}_{\mathrm{v}}(\mathrm{v}, I) \mid I \in \{\mathit{inst}(\mathrm{v}, \mathrm{u}) \mid \mathrm{u} \in \pi\}, \\ & & \qquad\qquad \mathit{known}(I, \pi_i) \neq \mathit{known}(I, \pi_i')\} \\ & & \text{where} \\ & & \mathit{known}(I, \pi) = 1 \text{ if } \exists\, \mathrm{u} \in \pi . \mathit{match}_{\mathrm{v}}(I, \mathrm{u}) \text{ defined; else, } 0 \end{array}$$

The helper *inst* partially matches a pattern against a value, leaving wildcards and binders in place:

$$\begin{array}{rcl} \mathit{inst} & : & \mathrm{v} \times \mathrm{u} \xrightarrow{\mathit{partial}} \mathrm{v} \\ \mathit{inst}(\star, \mathrm{u}) & = & \star \\ \mathit{inst}(\mathrm{x} : \tau, \mathrm{u}) & = & \mathrm{x} \\ \mathit{inst}(\mathrm{v}, \mathrm{v}) & = & \mathrm{v} \\ \mathit{inst}(m(\overrightarrow{\mathrm{v}}), m(\overrightarrow{\mathrm{u}})) & = & m(\overrightarrow{\mathrm{v}'}) \\ & & \text{if } |\overrightarrow{\mathrm{v}}| = |\overrightarrow{\mathrm{u}}| \\ & & \quad \overrightarrow{\mathrm{v}'} = \overrightarrow{\mathit{inst}(\mathrm{v}, \mathrm{u})} \\ \mathit{inst}(?\ \mathrm{v}, ?\ \mathrm{u}) & = & ?\ \mathrm{v}' \qquad \text{if } \mathrm{v}' = \mathit{inst}(\mathrm{v}, \mathrm{u}) \end{array}$$

The $\mathit{match}_{\mathrm{v}}$ function implements basic value-against-pattern matching. We assume without loss of generality that all binders in a given pattern are unique:

$$\begin{array}{rcl} \mathit{match}_{\mathrm{v}} & : & \mathrm{v} \times \mathrm{u} \xrightarrow{\mathit{partial}} \gamma \\ \mathit{match}_{\mathrm{v}}(\star, \mathrm{u}) & = & \emptyset \\ \mathit{match}_{\mathrm{v}}(\mathrm{x} : \tau, \mathrm{u}) & = & \{\mathrm{x} \mapsto \mathrm{u}\} \\ \mathit{match}_{\mathrm{v}}(\mathrm{v}, \mathrm{v}) & = & \emptyset \\ \mathit{match}_{\mathrm{v}}(m(\overrightarrow{\mathrm{v}}), m(\overrightarrow{\mathrm{u}})) & = & \bigcup \overrightarrow{\gamma} \\ & & \text{if } |\overrightarrow{\mathrm{v}}| = |\overrightarrow{\mathrm{u}}| \\ & & \quad \overrightarrow{\gamma} = \overrightarrow{\mathit{match}_{\mathrm{v}}(\mathrm{v}, \mathrm{u})} \\ \mathit{match}_{\mathrm{v}}(?\ \mathrm{v}, ?\ \mathrm{u}) & = & \mathit{match}_{\mathrm{v}}(\mathrm{v}, \mathrm{u}) \end{array}$$

The $\oplus$ operation updates an actor's knowledge of assertions based on a patch:

$$\begin{array}{rcl} \oplus & : & \pi \times \Delta \longrightarrow \pi \\ \pi \oplus \pi^+/\pi^- & = & (\pi \cup \pi^+) - \pi^- \end{array}$$

The evaluator for expressions is *eval*. The function is partial due to the possibility of unbound field names. A primitive interpretation δ is a partial function from an operation p and a vector of values the result value:

$$\begin{array}{rcl} \mathit{eval} & : & \mathrm{e} \times \sigma \xrightarrow{\mathit{partial}} \mathrm{v} \\ \mathit{eval}(b, \sigma) & = & b \\ \mathit{eval}(!\,\mathrm{x}, \sigma) & = & \sigma(\mathrm{x}) \\ \mathit{eval}(\star, \sigma) & = & \star \\ \mathit{eval}(\mathrm{x} : \tau, \sigma) & = & \mathrm{x} : \tau \\ \mathit{eval}(p(\overrightarrow{\mathrm{e}}), \sigma) & = & \delta(p, \overrightarrow{\mathrm{v}}) \text{if } \overrightarrow{\mathrm{v}} = \overrightarrow{\mathit{eval}(\mathrm{e}, \sigma)} \\ \mathit{eval}(m(\overrightarrow{\mathrm{e}}), \sigma) & = & m(\overrightarrow{\mathrm{v}}) \ \text{ if } \overrightarrow{\mathrm{v}} = \overrightarrow{\mathit{eval}(\mathrm{e}, \sigma)} \\ \mathit{eval}(?\ \mathrm{e}, \sigma) & = & ?\ \mathrm{v} \qquad \text{if } \mathrm{v} = \mathit{eval}(\mathrm{e}, \sigma) \end{array}$$

The *assertions-of* function produces the current assertions made by an actor given its active facet tree and field store:

$$\begin{aligned}
\mathit{assertions\text{-}of} &: \mathrm{FT} \times \sigma \longrightarrow \pi \\
\mathit{assertions\text{-}of}(\epsilon,\ \sigma) &= \emptyset \\
\mathit{assertions\text{-}of}(\mathrm{fn}[\overrightarrow{\mathrm{e}}\ \overrightarrow{(\mathrm{D}\ \mathrm{Pr})}].\overrightarrow{\mathrm{FT}},\ \sigma) &= \bigcup \overrightarrow{\mathit{assertions\text{-}of}_{\mathrm{e}}(\mathrm{e},\ \sigma)}\ \cup \\
&\quad\ \bigcup \overrightarrow{\mathit{assertions\text{-}of}_{\mathrm{D}}(\mathrm{D},\ \sigma)} \cup \\
&\quad\ \bigcup \overrightarrow{\mathit{assertions\text{-}of}(\mathrm{FT},\ \sigma)}
\end{aligned}$$

It utilizes a family of helper functions for determining the assertions associated with endpoints. The $\mathit{assertions\text{-}of}_{\mathrm{e}}$ function does the work for assertion endpoints:

$$\begin{aligned}
\mathit{assertions\text{-}of}_{\mathrm{e}} &: \mathrm{e} \times \sigma \longrightarrow \pi \\
\mathit{assertions\text{-}of}_{\mathrm{e}}(\star,\ \sigma) &= \textbf{Assertion} \\
\mathit{assertions\text{-}of}_{\mathrm{e}}(\mathrm{x} : \tau,\ \sigma) &= \textbf{Assertion} \\
\mathit{assertions\text{-}of}_{\mathrm{e}}(m(\overrightarrow{\mathrm{e}_i}),\ \sigma) &= \{m(\overrightarrow{\mathrm{u}_i}) \mid \mathrm{u}_i \in \mathit{assertions\text{-}of}_{\mathrm{e}}(\mathrm{e}_i,\ \sigma)\} \\
\mathit{assertions\text{-}of}_{\mathrm{e}}(?\ \mathrm{e},\ \sigma) &= \{?\ \mathrm{u} \mid \mathrm{u} \in \mathit{assertions\text{-}of}_{\mathrm{k}}(\mathrm{e},\ \sigma)\} \\
\mathit{assertions\text{-}of}_{\mathrm{e}}(\mathrm{e},\ \sigma) &= \{\mathit{eval}(\mathrm{e},\ \sigma)\}
\end{aligned}$$

While the $\mathit{assertions\text{-}of}_{\mathrm{D}}$ metafunction produces the assertion of interest when needed by an event-handler endpoint:

$$\begin{aligned}
\mathit{assertions\text{-}of}_{\mathrm{D}} &: \mathrm{D} \times \sigma \longrightarrow \pi \\
\mathit{assertions\text{-}of}_{\mathrm{D}}(\texttt{start},\ \sigma) &= \emptyset \\
\mathit{assertions\text{-}of}_{\mathrm{D}}(\texttt{stop},\ \sigma) &= \emptyset \\
\mathit{assertions\text{-}of}_{\mathrm{D}}(\texttt{asserted e},\ \sigma) &= \{?\ \mathrm{u} \mid \mathrm{u} \in \mathit{assertions\text{-}of}_{\mathrm{e}}(\mathrm{e},\ \sigma)\} \\
\mathit{assertions\text{-}of}_{\mathrm{D}}(\texttt{retracted e},\ \sigma) &= \{?\ \mathrm{u} \mid \mathrm{u} \in \mathit{assertions\text{-}of}_{\mathrm{e}}(\mathrm{e},\ \sigma)\}
\end{aligned}$$

C The Extended Light Actor

The full implementation of the light actor in Figure 4 is a good example for illustrating how the facet machine deals with loading and manipulating states. Here is the fixed version of the actor description, assuming suitable bindings for `wall-switch-id` and `my-id`:

```
start light-facet
  ∅
  (asserted wall-switch(wall-switch-id, ON)
   field state = ON in
     start during<wall-switch>
       light(my-id, !state) ∪ ∅
       (retracted wall-switch(wall-switch-id, ON)
        stop during<wall-switch>)
       (asserted room-assignment(my-id, room:String)
        start during<room-assignment>
          ∅
          (retracted room-assignment(my-id, room)
           stop during<room-assignment>)
           (asserted in-room(room, ⋆)
            start during<in-room>
              ∅
              (retracted in-room(room, ⋆)
               stop during<in-room>)
               (asserted room-empty(room)
                ·)
               (asserted room-occupied(room)
                ·))))
```

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle\, \textit{l-c}[I_{10}]\ ;\ (\textit{l-c},\ \cdot);\ \cdot;\ \{\texttt{wall-switch}(\ldots)\};\ \{\texttt{state} \mapsto \texttt{ON}\}\,\rangle} \\
\text{where}\ \ I_{10} & = & [\emptyset\ \mathit{WS}^{+}_{EH}]. \\
 & & \quad \mathrm{fid}_{w\text{-}s}[\texttt{light}(\ldots)\ \cup\ \emptyset \\
 & & \qquad \mathit{WS}^{-}_{EH} \\
 & & \qquad \mathit{RA}^{+}_{EH}[\texttt{during<w-s>} \mapsto \mathrm{fid}_{w\text{-}s}]].\cdot
\end{array}$$

$$\xrightarrow{\frac{\{\texttt{room-assignment}(\cdot)\}}{\emptyset}} \qquad \text{(by inject)}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle\, \textit{l-c}[I_{11}]\ ;\ (\textit{l-c},\ \cdot);\ \mathrm{PS}_{11}\ ;\ \pi_{11};\ \{\texttt{state} \mapsto \texttt{ON}\}\,\rangle} \\
\text{where}\ \ \mathrm{PS}_{11} & = & (\mathrm{fid}_{w\text{-}s},\ \texttt{start during<r-a>}\ \ldots) \\
\pi_{11} & = & \{\texttt{wall-switch}(\ldots), \texttt{room-assignment}(\ldots)\} \\
I_{11} & = & [\emptyset\ \mathit{WS}^{+}_{EH}]. \\
 & & \quad \mathrm{fid}_{w\text{-}s}[\texttt{light}(\ldots)\ \cup\ \emptyset \\
 & & \qquad \mathit{WS}^{-}_{EH} \\
 & & \qquad \mathit{RA}^{+}_{EH}[\texttt{during<w-s>} \mapsto \mathrm{fid}_{w\text{-}s}]].\cdot
\end{array}$$

$$\xrightarrow{\bullet} \qquad \text{(by start)}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle\, \textit{l-c}[I_{12}]\ ;\ (\mathrm{fid}_{w\text{-}s},\ \cdot);\ \cdot;\ \pi_{12}\ ;\ \{\texttt{state} \mapsto \texttt{ON}\}\,\rangle} \\
\text{where}\ \ \pi_{12} & = & \{\texttt{wall-switch}(\ldots), \texttt{room-assignment}(\ldots)\} \\
I_{12} & = & [\emptyset\ \mathit{WS}^{+}_{EH}]. \\
 & & \quad \mathrm{fid}_{w\text{-}s}[\texttt{light}(\ldots)\ \cup\ \emptyset \\
 & & \qquad \mathit{WS}^{-}_{EH} \\
 & & \qquad \mathit{RA}^{+}_{EH}[\texttt{during<w-s>} \mapsto \mathrm{fid}_{w\text{-}s}]]. \\
 & & \qquad \mathrm{fid}_{r\text{-}a}[\emptyset\ \ldots\ ].\cdot
\end{array}$$

$$\xrightarrow{\frac{\emptyset}{\{\texttt{wall-switch}(\cdot)\}}} \qquad \text{(by inject)}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle\, \textit{l-c}[I_{13}]\ ;\ (\textit{l-c},\ \cdot);\ \mathrm{PS}_{13};\ \pi_{13};\ \{\texttt{state} \mapsto \texttt{ON}\}\,\rangle} \\
\text{where}\ \ \mathrm{PS}_{13} & = & (\mathrm{fid}_{w\text{-}s},\ \texttt{stop}\ \mathrm{fid}_{w\text{-}s}) \\
\pi_{13} & = & \{\texttt{room-assignment}(\ldots)\} \\
I_{13} & = & [\emptyset\ \mathit{WS}^{+}_{EH}]. \\
 & & \quad \mathrm{fid}_{w\text{-}s}[\texttt{light}(\ldots)\ \cup\ \emptyset \\
 & & \qquad \mathit{WS}^{-}_{EH} \\
 & & \qquad \mathit{RA}^{+}_{EH}[\texttt{during<w-s>} \mapsto \mathrm{fid}_{w\text{-}s}]].\cdot
\end{array}$$

$$\xrightarrow{\bullet} \qquad \text{(by transfer)}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle\, \textit{l-c}[I_{14}\ ;\ (\textit{l-c},\ \cdot);\ \mathrm{PS}_{14};\ \pi_{14};\ \{\texttt{state} \mapsto \texttt{ON}\}\,\rangle} \\
\text{where}\ \ \mathrm{PS}_{14} & = & (\mathrm{fid}_{w\text{-}s},\ \texttt{stop}\ \mathrm{fid}_{w\text{-}s}) \\
\pi_{14} & = & \{\texttt{room-assignment}(\ldots)\} \\
I_{14} & = & [\emptyset\ \mathit{WS}^{+}_{EH}].
\end{array}$$

Fig. C1. The `light-facet` facet as an initial machine state and its transitions.

Figure C1 shows the complete machine transition sequence, starting from the initial state. From there, the machine executes a labeled inject transition to get the start instruction of the facet script. A transfer transition launches the script, which then immediately issues an action patch via another labeled inject transition. At this point, the facet is ready to shut down, which the machine realizes with stop transition.

D Type Judgments

The following definitions complete the type judgment for facet actor terms (section 6.1).

The judgment for event descriptions is $\Gamma\ \vdash_{\mathrm{D}}\ \mathrm{D}\ :\ \mathrm{D}_{\mathrm{T}}$:

$$\frac{}{\Gamma \vdash_{\mathrm{D}} \texttt{start} : \texttt{start}}\ \boxed{\textsc{D-Start}} \qquad \frac{}{\Gamma \vdash_{\mathrm{D}} \texttt{stop} : \texttt{stop}}\ \boxed{\textsc{D-Stop}}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} \texttt{e} : \tau \qquad \texttt{pattern}(\tau)}{\Gamma \vdash_{\mathrm{D}} \texttt{asserted e} : \texttt{asserted}\ \tau}\ \boxed{\textsc{D-Asserted}}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} \texttt{e} : \tau \qquad \texttt{pattern}(\tau)}{\Gamma \vdash_{\mathrm{D}} \texttt{retracted e} : \texttt{retracted}\ \tau}\ \boxed{\textsc{D-Asserted}}$$

The judgment for expressions is $\Gamma \vdash_{\mathrm{e}} \texttt{e} : \tau$:

$$\frac{}{\Gamma \vdash_{\mathrm{e}} b : \mathrm{B}}\ \boxed{\textsc{E-Base}} \qquad \frac{\Gamma(\mathrm{x}) = \tau}{\Gamma \vdash_{\mathrm{e}} \texttt{x} : \tau}\ \boxed{\textsc{E-Var}} \qquad \frac{}{\Gamma \vdash_{\mathrm{e}} \star : \star}\ \boxed{\textsc{E-Wildcard}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{e}} \texttt{x}:\tau : \texttt{x}:\tau}\ \boxed{\textsc{E-Bind}} \qquad \frac{\Gamma(\mathrm{x}) = \texttt{Field}\ \tau}{\Gamma \vdash_{\mathrm{e}} \texttt{!x} : \tau}\ \boxed{\textsc{E-Ref}}$$

$$\frac{\overrightarrow{\Gamma \vdash_{\mathrm{e}} \texttt{e} : \tau} \qquad \delta_\tau(p, \vec{\tau}) = \tau'}{\Gamma \vdash_{\mathrm{e}} p(\vec{\texttt{e}}) : \tau'}\ \boxed{\textsc{E-PrimOp}} \qquad \frac{\Gamma \vdash_{\mathrm{e}} \texttt{e} : \tau}{\Gamma \vdash_{\mathrm{e}} \texttt{? e} : \texttt{?}\ \tau}\ \boxed{\textsc{E-?}}$$

$$\frac{\overrightarrow{\Gamma \vdash_{\mathrm{e}} \texttt{e} : \tau}}{\Gamma \vdash_{\mathrm{e}} m(\vec{\texttt{e}}) : m(\vec{\tau})}\ \boxed{\textsc{E-Tuple}}$$

Types suitable for asserting, $\texttt{assertable}(\tau)$:

$$\frac{}{\texttt{assertable}(\mathrm{B})}\ \boxed{\textsc{A-Base}} \qquad \frac{}{\texttt{assertable}(\star)}\ \boxed{\textsc{A-Wildcard}}$$

$$\frac{\texttt{assertable}(\tau)}{\texttt{assertable}(\texttt{?}\ \tau)}\ \boxed{\textsc{A-Wildcard}} \qquad \frac{\overrightarrow{\texttt{assertable}(\tau)}}{\texttt{assertable}(m(\vec{\tau}))}\ \boxed{\textsc{A-Tuple}}$$

Types suitable for patterns, $\texttt{pattern}(\tau)$:

$$\frac{}{\texttt{pattern}(\mathrm{B})}\ \boxed{\textsc{P-Base}} \qquad \frac{}{\texttt{pattern}(\star)}\ \boxed{\textsc{P-Wildcard}} \qquad \frac{}{\texttt{pattern}(\texttt{x}:\tau)}\ \boxed{\textsc{P-Bind}}$$

$$\frac{\texttt{pattern}(\tau)}{\texttt{pattern}(\texttt{?}\ \tau)}\ \boxed{\textsc{P-Wildcard}} \qquad \frac{\overrightarrow{\texttt{pattern}(\tau)}}{\texttt{pattern}(m(\vec{\tau}))}\ \boxed{\textsc{P-Tuple}}$$

The following metafunctions complete the definition of the type judgment.

The $\mathit{bindings}_{\mathrm{D}}$ metafunction creates a type environment from the binding variables in an event description:

$$
\begin{array}{rcl}
\mathit{bindings}_{\mathrm{D}} & : & \mathrm{D} \longrightarrow \Gamma \\
\mathit{bindings}_{\mathrm{D}}(\texttt{start}) & = & \cdot \\
\mathit{bindings}_{\mathrm{D}}(\texttt{stop}) & = & \cdot \\
\mathit{bindings}_{\mathrm{D}}(\texttt{asserted e}) & = & \mathit{bindings}_{\mathrm{P}}(\mathrm{e}) \\
\mathit{bindings}_{\mathrm{D}}(\texttt{retracted e}) & = & \mathit{bindings}_{\mathrm{P}}(\mathrm{e})
\end{array}
$$

It utilizes a helper function for patterns, $\mathit{bindings}_{\mathrm{P}}$:

$$
\begin{array}{rcl}
\mathit{bindings}_{\mathrm{P}} & : & \mathrm{e} \longrightarrow \Gamma \\
\mathit{bindings}_{\mathrm{P}}(\star) & = & \cdot \\
\mathit{bindings}_{\mathrm{P}}(\mathrm{x} : \tau) & = & \mathrm{x} : \tau \\
\mathit{bindings}_{\mathrm{P}}(m(\overrightarrow{\mathrm{e}})) & = & \overrightarrow{\mathit{bindings}_{\mathrm{P}}(\mathrm{e})} \\
\mathit{bindings}_{\mathrm{P}}(\texttt{?}\ \mathrm{e}) & = & \mathit{bindings}_{\mathrm{P}}(\mathrm{e}) \\
\mathit{bindings}_{\mathrm{P}}(\mathrm{e}) & = & \cdot
\end{array}
$$

Two metafunctions prune type environments to help keep track of facet and field names. The first, *prune*, removes all facet and field names from an environment, i.e.,variables that are not shared between actors:

$$
\begin{array}{rcl}
\mathit{prune} & : & \Gamma \longrightarrow \Gamma \\
\mathit{prune}(\cdot) & = & \cdot \\
\mathit{prune}(\Gamma, \mathrm{fn} : \texttt{FacetName}) & = & \mathit{prune}(\Gamma) \\
\mathit{prune}(\Gamma, \mathrm{x} : \tau) & = & \begin{cases} \mathit{prune}(\Gamma) & \text{if } \mathrm{x} = \texttt{Field}\ \tau' \\ \mathit{prune}(\Gamma), \mathrm{x} : \tau & \text{otherwise} \end{cases}
\end{array}
$$

The second, *prune-up-to*, removes all facet names that are children of a particular facet, giving an environment of names of active facets after that facet stops:

$$
\begin{array}{rcl}
\mathit{prune\text{-}up\text{-}to} & : & \mathrm{fn} \times \Gamma \longrightarrow \Gamma \\
\mathit{prune\text{-}up\text{-}to}(\mathrm{fn},\ \cdot) & = & \cdot \\
\mathit{prune\text{-}up\text{-}to}(\mathrm{fn},\ \Gamma, \mathrm{fn}' : \texttt{FacetName}) & = & \begin{cases} \Gamma & \text{if } \mathrm{fn} = \mathrm{fn}' \\ \mathit{prune\text{-}up\text{-}to}(\mathrm{fn},\ \Gamma) & \text{otherwise} \end{cases} \\
\mathit{prune\text{-}up\text{-}to}(\mathrm{fn},\ \Gamma, \mathrm{x} : \tau) & = & \mathrm{x} : \tau, \mathit{prune\text{-}up\text{-}to}(\Gamma,\ \mathrm{fn})
\end{array}
$$

E Dataspace Metafunctions

The semantics of Section 7 refers to several metafunctions. For completeness, this section provides their formal definitions. But, also see the work of Caldwell *et al.* (2020) plus Garnock-Jones & Felleisen (2016), which fully specify the semantics of dataspace systems as well. The primary difference between the semantics presented here and prior work is the aggregation of incoming patches performed by bc_{Δ} below.

The $\mathit{boot}_{\mathrm{DS}}$ metafunction initializes a dataspace program based on a collection of initial facet-based actor descriptions:

$$
\begin{array}{rcl}
\mathit{boot}_{\mathrm{DS}} & : & \overrightarrow{\mathrm{Pr}} \longrightarrow \mathrm{DS} \\
\mathit{boot}_{\mathrm{DS}}(\overrightarrow{\mathrm{Pr}}) & = & [\overrightarrow{(\ell, \pi/\emptyset)}; \emptyset; \overrightarrow{\mathrm{A}}] \\
 & & \text{where} \\
 & & \overrightarrow{\mathrm{P}} = \overrightarrow{\mathit{boot}_{\mathrm{P}}(\mathrm{Pr})} \\
 & & \overrightarrow{\Sigma, \pi} = \overrightarrow{\mathit{boot}_{\Sigma}(\mathrm{P})} \\
 & & \overrightarrow{\ell} = 0 \cdot 1 \cdot \ldots \cdot |\overrightarrow{\mathrm{Pr}}| - 1 \\
 & & \overrightarrow{\mathrm{A}} = \overrightarrow{\ell \mapsto \Sigma}
\end{array}
$$

The $boot_\Sigma$ function boots a process description to an actor state:

$$\begin{array}{rcl} boot_\Sigma & : & \mathrm{P} \longrightarrow \Sigma \times \pi \\ boot_\Sigma(\texttt{actor f v } \pi) & = & \emptyset/\emptyset \triangleright (\mathrm{f},\ \mathrm{v}), \pi \end{array}$$

The *update* metafunction is the work horse of dataspace event dispatch:

$$\begin{array}{rcl} update & : & (\mathrm{R} \times \overrightarrow{\mathrm{A}}) \times (\ell \times \Delta) \longrightarrow \mathrm{R} \times \overrightarrow{\mathrm{A}} \\ update((\mathrm{R},\ \overrightarrow{\mathrm{A}}),\ (\ell, \Delta)) & = & \mathrm{R}\ \oplus_{\mathrm{R}} (\Delta, \ell), \overrightarrow{bc_\Delta(\ell,\ \mathrm{R},\ \Delta,\ \mathrm{A})} \end{array}$$

The bc_Δ metafunction updates the pending event for an actor if it has an active interest:

$$\begin{array}{rcl} bc_\Delta & : & \ell \times \mathrm{R} \times \mathrm{Evt} \times \mathrm{A}_Q \longrightarrow \mathrm{A}_Q \\ bc_\Delta(\ell_{evt},\ \mathrm{R}_{old},\ \pi_{add}/\pi_{del},\ \ell \mapsto \Delta \triangleright \mathrm{B}\,\cdot) & = & \begin{cases} \ell \mapsto \Delta \circ \Delta_{fb} \triangleright \mathrm{B}\,\cdot & \text{if } \ell = \ell_{evt} \\ \ell \mapsto \Delta \circ \Delta_{other} \triangleright \mathrm{B}\,\cdot & \text{if } \ell \neq \ell_{evt} \end{cases} \end{array}$$

where

$$\begin{array}{rcl} \Delta_{fb} & = & \{\mathrm{u} \mid \mathrm{u} \in \pi_{\bullet add},\ (?\ \mathrm{u}, \ell) \in \mathrm{R}_{new}\} \cup \{\mathrm{u} \mid \mathrm{u} \in (\pi_\circ \cup \pi_{\bullet add} - \pi_{\bullet del}), ?\ \mathrm{u} \in \pi_{add}\}/ \\ & & \{\mathrm{u} \mid \mathrm{u} \in \pi_{\bullet del},\ (?\ \mathrm{u}, \ell) \in \mathrm{R}_{old}\} \cup \{\mathrm{u} \mid \mathrm{u} \in \pi_\circ, ?\ \mathrm{u} \in \pi_{del}\} \\ \Delta_{other} & = & \{\mathrm{u} \mid \mathrm{u} \in \pi_{\bullet add},\ (?\ \mathrm{u}, \ell) \in \mathrm{R}_{old}\}/\{\mathrm{u} \mid \mathrm{u} \in \pi_{\bullet del},\ (?\ \mathrm{u}, \ell) \in \mathrm{R}_{old}\} \\ \pi_\bullet & = & \{\mathrm{u} \mid (\mathrm{u}, \ell') \in \mathrm{R}_{old},\ \ell' \neq \ell_{evt}\} \\ \mathrm{R}_{new} & = & \mathrm{R}_{old}\ \oplus_{\mathrm{R}} (\pi_{add}/\pi_{del}, \ell) \\ \pi_{\bullet add} & = & \pi_{add} - \pi_\bullet \\ \pi_\circ & = & \{\mathrm{u} \mid (\mathrm{u}, \ell') \in \mathrm{R}_{old}\} \\ \pi_{del\bullet} & = & \pi_{del} - \pi_\bullet \end{array}$$

The $\circ$ operator applies two patches in sequence, maintaining disjointness:

$$\begin{array}{rcl} \circ & : & \Delta \times \Delta \longrightarrow \Delta \\ \dfrac{\pi_1^+}{\pi_1^-} \circ \dfrac{\pi_2^+}{\pi_2^-} & = & \dfrac{(\pi_1^+ - \pi_2^-) \cup \pi_2^+}{(\pi_1^- - \pi_2^+) \cup \pi_2^-} \end{array}$$

The helper function *patch* calculates the difference in assertions between two machine states:

$$\begin{array}{rcl} patch & : & \mathrm{M} \times \mathrm{M} \longrightarrow \Delta \\ patch(\mathrm{M},\ \mathrm{M}') & = & \dfrac{assertions\text{-}of_{\mathrm{M}}(\mathrm{M}') - assertions\text{-}of_{\mathrm{M}}(\mathrm{M})}{assertions\text{-}of_{\mathrm{M}}(\mathrm{M}) - assertions\text{-}of_{\mathrm{M}}(\mathrm{M}')} \end{array}$$

The *label-to-action* function translates a facet machine's internal transition label to a sequence of (zero or one) actions:

$$\begin{array}{rcl} label\text{-}to\text{-}action & : & l_\bullet \longrightarrow \overrightarrow{\mathrm{act}} \\ label\text{-}to\text{-}action(\bullet) & = & \cdot \\ label\text{-}to\text{-}action(\overline{\mathrm{Pr}}) & = & boot_{\mathrm{P}}(\mathrm{Pr}) \end{array}$$

The *assertions-of*$_{\mathrm{M}}$ function extends the *assertions-of* family of functions, determining the active assertions of a facet machine:

$$\begin{array}{rcl} assertions\text{-}of_{\mathrm{M}} & : & \mathrm{M} \longrightarrow \pi \\ assertions\text{-}of_{\mathrm{M}}(\texttt{error}) & = & \emptyset \\ assertions\text{-}of_{\mathrm{M}}(\langle \mathrm{FT};\ \overrightarrow{\mathrm{T}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma \rangle) & = & assertions\text{-}of(\mathrm{FT},\ \sigma) \end{array}$$

The $\oplus_{\mathrm{R}}$ operator updates the table of active assertions based on an actor's patch:

$$\begin{array}{rcl} \oplus_{\mathrm{R}} & : & \mathrm{R} \times (\Delta, \ell) \longrightarrow \mathrm{R} \\ \mathrm{R}\ \oplus_{\mathrm{R}} (\pi^+/\pi^-, \ell) & = & \mathrm{R} \cup \{(\mathrm{u}, \ell) \mid \mathrm{u} \in \pi^+\} - \{(\mathrm{u}, \ell) \mid \mathrm{u} \in \pi^-\} \end{array}$$

The $\mathit{dispatch}_{\mathrm{DS}}$ function invokes an actor's behavior on its pending event, when non-empty:

$$\begin{array}{rcl}
\mathit{dispatch}_{\mathrm{DS}} & : & \Sigma \longrightarrow \Sigma \times \Delta \times \overrightarrow{\mathrm{P}} \\
\mathit{dispatch}_{\mathrm{DS}}(\Delta \rhd (\mathrm{f},\ \mathrm{v})) & = & \begin{cases} \Delta \rhd (\mathrm{f},\ \mathrm{v}), \emptyset/\emptyset, \cdot & \text{if } \Delta = \emptyset/\emptyset \\ \emptyset/\emptyset \rhd (\mathrm{f},\ \mathrm{v}'), \Delta,\ \overrightarrow{\mathrm{P}} & \text{otherwise} \end{cases} \\
 & \text{where} & \\
 & & \mathrm{v}', \overrightarrow{\mathrm{act}} = \mathrm{f}(\Delta,\ \ \mathrm{v}) \\
 & & \Delta = \Delta_0 \circ \Delta_1 \circ \ldots \text{ for } \Delta_i \in \overrightarrow{\mathrm{act}} \\
 & & \overrightarrow{\mathrm{P}} = \mathrm{P}_0 \cdot \mathrm{P}_1 \ldots \text{ for } \mathrm{P}_i \in \overrightarrow{\mathrm{act}}
\end{array}$$

F Type Level Programs

Section 6.2 sketches a semantics for the behavioral types of facets, appealing to a type-level facet machine. In the same vein, Section 7.2 informally presents a semantics for the types of dataspaces. This section supplements these with formal definitions. The definitions of the metafunctions are just like those from Appendix B; their definitions are omitted.

Figure C2 defines the syntax for describing the states of a type-level facet machine and Figure C3 the transition relation.

F.1 Machine Typing

Theorem 1, soundness for the type-level facet machine, needs matching type judgments.

The type judgment for facet machines, $\vdash_{\mathrm{M}}\ \mathrm{M}\ :\ \mathrm{M_T}$:

$$\begin{array}{rcl}
\mathrm{M_T} \in \textbf{TyMachine} & = & \langle \mathrm{FT_T};\ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma_\tau \rangle \\
\mathrm{FT_T} \in \textbf{FctTreeTy} & = & \varepsilon \\
 & | & \mathrm{fn}[\mathrm{A}_\tau(\overrightarrow{\mathrm{D_T}\ \ \mathrm{T}})].\overrightarrow{\mathrm{FT_T}} \\
\mathrm{C_T} \in \textbf{TyContext} & = & \square \\
 & | & \mathrm{fid}[\mathrm{A}_\tau(\overrightarrow{\mathrm{D_T}\ \ \mathrm{T}})].\overrightarrow{\mathrm{FT_T}} \cdot \mathrm{C_T} \cdot \overrightarrow{\mathrm{FT_T}} \\
\mathrm{PS_T} \in \textbf{PScriptTy} & = & (\mathrm{fid},\ \mathrm{T}) \\
\mathrm{I_T} \in \textbf{InstrTy} & = & \mathtt{start}\ \mathrm{fn}\ \mathrm{A}_\tau\ (\overrightarrow{\mathrm{D_T}\ \ \mathrm{T}})\ @\ \mathrm{fid} \\
 & | & \mathtt{stop}\ \mathrm{fn} \\
 & | & \mathtt{spawn}\ \overrightarrow{\mathrm{T}} \\
\sigma_\tau \in \textbf{StoreTy} & = & \mathrm{x} \xrightarrow{\mathit{fin}} \tau \\
\delta_\tau \in \textbf{TyInterp} & = & p \times \overrightarrow{\tau} \longrightarrow \tau \\
\gamma_\tau \in \textbf{TySub} & = & \mathrm{x} \xrightarrow{\mathit{fin}} \tau \\
\mathscr{S}_\tau \in \textbf{TySubSets} & = & \mathscr{P}(\textbf{TySub})
\end{array}
\qquad
\begin{array}{rcl}
\tau_{\mathrm{R}} \in \textbf{SpecType} & = & \{\mathrm{B} \mid == \mathrm{R}\} \\
 & | & ?\ \tau_{\mathrm{R}} \\
 & | & m(\overrightarrow{\tau_{\mathrm{R}}}) \\
\mathrm{R} \in \textbf{Refine} & = & \mathrm{x} \mid b \\
\pi_\tau \in \textbf{ASetTy} & = & \mathscr{P}(\tau_{\mathrm{R}}) \\
\Delta_\tau \in \textbf{PatchTy} & = & \pi_\tau^+ / \pi_\tau^- \\
 & & \text{where } \pi_\tau^+ \cap \pi_\tau^- = \emptyset \\
\Delta_\tau \in \textbf{ExtEventTy} & = & \Delta_\tau \\
\mathrm{Evt}_\tau \in \textbf{EventTy} & = & \Delta_\tau \\
 & | & \mathtt{start} \\
 & | & \mathtt{stop} \\
l_\tau \in \textbf{LabelTy} & = & \bullet \\
 & | & \Delta_\tau \\
 & | & \overline{\mathrm{T}}
\end{array}$$

Fig. C2. Type evaluation syntax.

$$
\begin{array}{|l|}
\hline
\langle \mathrm{FT_T};\ \cdot;\ \cdot;\ \pi_\tau;\ \sigma_\tau\rangle \qquad \xrightarrow{\Delta_\tau}\ \langle \mathrm{FT_T};\ \cdot;\ \overrightarrow{\mathrm{PS_T}};\ \pi'_\tau;\ \sigma_\tau\rangle \qquad \textsf{(injectT)}\\
\text{where}\\
\quad \begin{array}{lcl}
\pi'_\tau & = & \pi_\tau \oplus_{\mathrm{T}} \Delta_\tau\\
\overrightarrow{\mathrm{PS_T}} & = & \mathit{dispatch}_{\mathrm{T}}(\mathrm{FT_T},\ \Delta_\tau,\ \pi_\tau,\ \pi'_\tau,\ \sigma_\tau,\ \langle\cdot\rangle)
\end{array}\\
\hline
\langle \mathrm{FT_T};\ \cdot;\ (\mathrm{fid},\ \mathrm{T})\cdot\overrightarrow{\mathrm{PS'_T}};\ \pi_\tau;\ \sigma_\tau\rangle \qquad \xrightarrow{\bullet}\ \mathrm{M'_T} \qquad \textsf{(transferT)}\\
\text{where}\\
\quad \begin{array}{lcl}
\mathrm{M'_T} & = & \langle \mathrm{FT_T};\ (\mathrm{fid},\ \overrightarrow{\mathrm{I_T}});\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma'_\tau\rangle\\
\overrightarrow{\mathrm{I_T}}, \sigma'_\tau & = & \mathit{p\text{-}e}_{\mathrm{T}}(\mathrm{T},\ \sigma_\tau, \mathrm{fid})
\end{array}\\
\hline
\langle \mathrm{FT_T};\ \texttt{start}\ \mathrm{fn}\ \mathrm{A}_\tau\ \overrightarrow{(\mathrm{D_T}\ \ \mathrm{T})}\ @\ \mathrm{fid}\cdot\overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma_\tau\rangle \qquad \xrightarrow{\bullet}\ \mathrm{M'_T} \qquad \textsf{(startT)}\\
\\
\text{where}\\
\quad \begin{array}{lcl}
\mathrm{M'_T} & = & \langle \mathrm{FT'_T};\ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}}\cdot\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{start}}}\cdot\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{boot}}}\cdot\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{stop}}};\ \pi_\tau;\ \sigma_\tau\rangle\\
\multicolumn{3}{l}{\mathrm{fn}_{\mathit{new}}\ \ \text{fresh in}\ \ \mathrm{FT_T}}\\
\mathrm{FT}_{\mathrm{T}\ \mathit{new}} & = & \mathrm{fn}_{\mathit{new}}[\mathrm{A}_\tau\ \overrightarrow{(\mathrm{D_T}\ \mathrm{T}[\mathrm{fn}\mapsto\mathrm{fn}_{\mathit{new}}])}].\varepsilon\\
\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{start}}} & = & \mathit{dispatch}_{\mathrm{T}}(\mathrm{FT}_{\mathrm{T}\ \mathit{new}},\ \texttt{start},\ \emptyset,\ \emptyset,\ \sigma_\tau,\ \mathrm{fid})\\
\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{boot}}} & = & \mathit{dispatch}_{\mathrm{T}}(\mathrm{FT}_{\mathrm{T}\ \mathit{new}},\ \pi_\tau/\emptyset,\ \emptyset,\ \pi_\tau,\ \sigma_\tau,\ \mathrm{fid})\\
\mathrm{FT'_T} & = & \begin{cases} \mathrm{C_T}[\mathrm{FT}_{\mathrm{T}\ \mathit{new}}] & \text{if } \mathit{locate}_{\mathrm{T}}(\mathrm{FT_T},\ \mathrm{fid}) = \mathrm{C_T}\\ \mathrm{FT_T} & \text{otherwise}\end{cases}\\
\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{stop}}} & = & \begin{cases} \cdot & \text{if } \mathit{locate}_{\mathrm{T}}(\mathrm{FT_T},\ \mathrm{fid})\ \text{defined}\\ \mathit{dispatch}_{\mathrm{T}}(\mathrm{FT}_{\mathrm{T}\ \mathit{new}},\ \texttt{stop},\ \pi_\tau,\ \pi_\tau,\ \sigma_\tau,\ \mathrm{fid}) & \text{otherwise}\end{cases}
\end{array}\\
\hline
\langle \mathrm{FT_T};\ \texttt{stop}\ \mathrm{fn}\cdot\overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma_\tau\rangle \qquad \xrightarrow{\bullet}\ \langle \mathrm{FT'_T};\ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}}\cdot\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{stop}}};\ \pi_\tau;\ \sigma_\tau\rangle \qquad \textsf{(stopT)}\\
\text{where}\\
\quad \begin{array}{lcl}
\mathrm{FT'_T} & = & \begin{cases} \mathrm{C_T}[\varepsilon] & \text{if } \mathrm{FT_T} = \mathrm{C_T}[\mathrm{fn}[\ldots].\overrightarrow{\mathrm{FT_T}}]\\ \mathrm{FT_T} & \text{otherwise}\end{cases}\\
\mathrm{fid} & = & \mathit{facet\text{-}context}_{\mathrm{T}}(\mathrm{C_T})\\
\overrightarrow{\mathrm{PS}_{\mathrm{T}\ \mathit{stop}}} & = & \begin{cases} \mathit{dispatch}_{\mathrm{T}}(\mathrm{fn}[\ldots].\overrightarrow{\mathrm{FT_T}},\ \texttt{stop},\ \pi_\tau,\ \pi_\tau,\ \sigma_\tau,\ \mathrm{fid}) & \text{if } \mathrm{FT_T} = \mathrm{C_T}[\mathrm{fn}[\ldots].\overrightarrow{\mathrm{FT_T}}]\\ \cdot & \text{otherwise}\end{cases}
\end{array}\\
\hline
\langle \mathrm{FT_T};\ \texttt{spawn}\ \mathrm{T}\cdot\overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma_\tau\rangle \qquad \xrightarrow{\overline{\mathrm{T}}}\ \langle \mathrm{FT_T};\ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma_\tau\rangle \qquad \textsf{(spawnT)}\\
\hline
\end{array}
$$

Fig. C3. Type machine transitions.

$$
\dfrac{\begin{array}{c}\mathit{store\text{-}bindings}(\sigma_\tau) = \Gamma \qquad \Gamma \vdash_{\mathrm{FT}}\ \mathrm{FT}\ :\ \mathrm{FT_T}\\ \overrightarrow{\Gamma \vdash_{\mathrm{I}}\ \mathrm{I}\ :\ \mathrm{I_T}} \qquad \overrightarrow{\Gamma \vdash_{\mathrm{PS}}\ \mathrm{PS}\ :\ \mathrm{PS_T}} \qquad \vdash_\pi\ \pi\ :\ \pi_\tau \qquad \vdash_\sigma\ \sigma\ :\ \sigma_\tau\end{array}}{\vdash_{\mathrm{M}}\ \langle \mathrm{FT};\ \ \overrightarrow{\mathrm{I}}\ ;\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma\rangle\ :\ \langle \mathrm{FT_T};\ \ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_\tau;\ \sigma_\tau\rangle}\ \boxed{\textsc{T-Machine}}
$$

The *store-bindings* function creates a type environment from a store:

$$
\begin{array}{rcl}
\mathit{store\text{-}bindings} & : & \sigma_\tau \longrightarrow \Gamma\\
\mathit{store\text{-}bindings}(\emptyset) & = & \cdot\\
\mathit{store\text{-}bindings}(\sigma_\tau \uplus \mathrm{x}\mapsto\tau) & = & \mathit{store\text{-}bindings}(\sigma_\tau), \mathrm{x}:\tau
\end{array}
$$

The type judgment for facet trees, $\Gamma\ \vdash_{\mathrm{FT}}\ \mathrm{FT}\ :\ \mathrm{FT_T}$:

$$
\dfrac{}{\Gamma\ \vdash_{\mathrm{FT}}\ \epsilon\ :\ \epsilon}\ \boxed{\textsc{FT-Empty}}
$$

$$
\dfrac{\begin{array}{c}\Gamma \vdash_{\mathrm{A}}\ \mathrm{A}\ :\ \mathrm{A}_\tau \qquad \overrightarrow{\Gamma \vdash_{\mathrm{D}}\ \mathrm{D}\ :\ \mathrm{D_T}}\\ \overrightarrow{\Gamma, \mathrm{fn} : \texttt{FacetName}, \mathit{bindings}_{\mathrm{D}}(\mathrm{D}) \vdash_{\mathrm{Pr}}\ \mathrm{Pr}\ :\ \mathrm{T}} \qquad \overrightarrow{\Gamma \vdash_{\mathrm{FT}}\ \mathrm{FT}\ :\ \mathrm{FT_T}}\end{array}}{\Gamma\ \vdash_{\mathrm{FT}}\ \mathrm{fn}[\overrightarrow{e}\ \overrightarrow{(\mathrm{D}\ \ \mathrm{Pr})}].\overrightarrow{\mathrm{FT}}\ :\ \mathrm{fn}[\mathrm{A}_\tau\overrightarrow{(\mathrm{D_T}\ \ \mathrm{T})}].\overrightarrow{\mathrm{FT_T}}}\ \boxed{\textsc{FT-Tree}}
$$

The type judgment for machine instructions, $\Gamma \vdash_{\mathrm{I}} \mathrm{I} : \mathrm{I}_{\mathrm{T}}$:

$$\frac{\Gamma \vdash_{\mathrm{A}} \mathrm{A} : \mathrm{A}_\tau \qquad \overrightarrow{\Gamma, \mathit{bindings}_{\mathrm{D}}(\mathrm{D}) \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}}{\Gamma \vdash_{\mathrm{I}} \texttt{start fn } \vec{\mathrm{e}}\ (\overrightarrow{\mathrm{D}\ \mathrm{Pr}})\ @\ \mathrm{fid} : \texttt{start fn } \mathrm{A}_\tau\ (\overrightarrow{\mathrm{D}_{\mathrm{T}}\ \mathrm{T}})\ @\ \mathrm{fid}}\ \boxed{\text{I-START}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{I}} \texttt{stop fn} : \texttt{stop fn}}\ \boxed{\text{I-STOP}} \qquad \frac{\mathit{prune}(\Gamma) \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}{\Gamma \vdash_{\mathrm{I}} \texttt{spawn Pr} : \texttt{spawn } \vec{\mathrm{T}}}\ \boxed{\text{I-SPAWN}}$$

The type judgment for pending scripts, $\Gamma \vdash_{\mathrm{PS}} \mathrm{PS} : \mathrm{PS}_{\mathrm{T}}$:

$$\frac{\Gamma \vdash_{\mathrm{Pr}} \mathrm{Pr} : \mathrm{T}}{\Gamma \vdash_{\mathrm{PS}} (\mathrm{fid},\ \mathrm{Pr}) : (\mathrm{fid},\ \mathrm{T})}\ \boxed{\text{PS-SCRIPT}}$$

The type judgment for assertion sets, $\vdash_\pi \pi : \pi_\tau$:

$$\frac{\pi_\tau = \{\tau_{\mathrm{R}} \mid \mathrm{u} \in \pi, \vdash_{\tau_{\mathrm{R}}} \mathrm{u} : \tau_{\mathrm{R}}\}}{\vdash_\pi \pi : \pi_\tau}\ \boxed{\pi\text{-SET}}$$

The type judgment for assertions, $\vdash_{\tau_{\mathrm{R}}} \mathrm{u} : \tau_{\mathrm{R}}$:

$$\frac{}{\vdash_{\tau_{\mathrm{R}}} b : \mathrm{B}}\ \boxed{\text{R-BASE}} \qquad \frac{\overrightarrow{\vdash_{\tau_{\mathrm{R}}} \mathrm{u} : \tau_{\mathrm{R}}}}{\vdash_{\tau_{\mathrm{R}}} m(\vec{\mathrm{u}}) : m(\vec{\tau_{\mathrm{R}}})}\ \boxed{\text{R-TUPLE}} \qquad \frac{\vdash_{\tau_{\mathrm{R}}} \mathrm{u} : \tau_{\mathrm{R}}}{\vdash_{\tau_{\mathrm{R}}} ?\,\mathrm{u} : ?\,\tau_{\mathrm{R}}}\ \boxed{\text{R-?}}$$

The type judgment for patches, $\vdash_\Delta \Delta : \Delta_\tau$:

$$\frac{\vdash_\pi \pi^+ : \pi_\tau^+ \qquad \vdash_\pi \pi^+ : \pi_\tau^+}{\vdash_\Delta \pi^+/\pi^- : \pi_\tau^+/\pi_\tau^-}\ \boxed{\Delta\text{-PATCH}}$$

The type judgment for events, $\vdash_{\mathrm{Evt}} \mathrm{Evt} : \mathrm{Evt}_\tau$:

$$\frac{\vdash_\Delta \Delta : \Delta_\tau}{\vdash_{\mathrm{Evt}} \Delta : \Delta_\tau}\ \boxed{\text{EVT-PATCH}} \qquad \frac{}{\vdash_{\mathrm{Evt}} \texttt{start} : \texttt{start}}\ \boxed{\text{EVT-START}}$$

$$\frac{}{\vdash_{\mathrm{Evt}} \texttt{stop} : \texttt{stop}}\ \boxed{\text{EVT-STOP}}$$

The type judgment for stores, $\vdash_\sigma \sigma : \sigma_\tau$:

$$\frac{\sigma_\tau = \{\mathrm{x} \mapsto \tau \mid \mathrm{x} \mapsto \mathrm{v} \in \sigma, \cdot \vdash_{\mathrm{e}} \mathrm{v} : \tau\}}{\vdash_\sigma \sigma : \sigma_\tau}\ \boxed{\sigma\text{-SET}}$$

The type judgment for substitutions, $\vdash_\gamma \gamma : \gamma_\tau$:

$$\frac{\gamma_\tau = \{\mathrm{x} \mapsto \tau \mid \mathrm{x} \mapsto \mathrm{v} \in \gamma, \cdot \vdash_{\mathrm{e}} \mathrm{v} : \tau\}}{\vdash_\gamma \gamma : \gamma_\tau}\ \boxed{\gamma\text{-SUB}}$$

The type judgment for transition labels, $\vdash_l\ l\ :\ l_\tau$:

$$\frac{}{\vdash_l\ \bullet\ :\ \bullet}\ \boxed{\text{L-None}} \qquad \frac{\vdash_{\text{Evt}}\ \Delta\ :\ \Delta_\tau}{\vdash_l\ \Delta\ :\ \Delta_\tau}\ \boxed{\text{L-Evt}} \qquad \frac{\cdot\ \vdash_{\text{Pr}}\ \text{Pr}\ :\ \text{T}}{\vdash_l\ \overline{\text{Pr}}\ :\ \overline{\text{T}}}\ \boxed{\text{L-Spawn}}$$

F.2 Type-level Dataspaces

A type-level dataspace DS_T has nearly the same semantics as the one from Section 7, with two modifications. First, assertions range over types τ. Second, actor behaviors are defined in terms of facet machine types, defined above, with related metafunctions such as $interp_\text{M}$ lifted to the type level, mutatis mutandis.

G Proof Details

The following lemmas establish that the machine transitions for actors and types are related via typing, under certain conditions. The transitions fall roughly in the following categories:

- initialization (Lemmas 13);
- transitions in response to an external stimulus (Lemmas 15, 16, 17); and
- transitions for performing internal work (Lemmas 18, 19).

Finally, Lemma 20 establishes that related type and term machines make related assertions.

Lemma 13 (Boot)**.** *If*

- $\Gamma\ \vdash_{\text{Pr}}\ \text{Pr}\ :\ \text{T}$
- $boot_{\text{Pr}}(\text{Pr},\ \sigma) = \langle \text{FT};\ \overrightarrow{\text{I}};\ \overrightarrow{\text{PS}};\ \pi;\ \sigma\rangle$

then

- $boot_{\text{T}}(\text{T}) = \langle \text{FT}_\text{T};\ \overrightarrow{\text{I}_\text{T}};\ \overrightarrow{\text{PS}_\text{T}};\ \pi_\tau;\ \emptyset\rangle$
- $\vdash_\text{M}\ \langle \text{FT};\ \overrightarrow{\text{I}};\ \overrightarrow{\text{PS}};\ \pi;\ \sigma\rangle\ :\ \langle \text{FT}_\text{T};\ \text{S}_\text{T};\ \overrightarrow{\text{PS}_\text{T}};\ \pi_\tau;\ \textit{type-store}(\sigma)\rangle$

Proof By induction on the type derivation and the soundness of the *dispatch* metafunction (Lemma 16). □

Definition 14 (Store Typing)**.**

$$\begin{array}{rcl} \textit{type-store} & : & \sigma \longrightarrow \sigma_\tau \\ \textit{type-store}(\emptyset) & = & \emptyset \\ \textit{type-store}(\sigma[\text{x} \mapsto \text{v}]) & = & \textit{type-store}(\sigma)[\text{x} \mapsto \tau] \\ & & \text{where}\ \cdot\ \vdash_\text{e}\ \text{v}\ :\ \tau \end{array}$$

In other words, initialization of related actor terms and types yields related term and type machines.

Lemma 15 (Matching Events). *If*

- $\vdash_{\mathrm{e}} \ \mathrm{v} \ : \tau$
- $\vdash_{\mathrm{e}} \ \mathrm{u} \ : \tau'$

then $\mathit{match}_{\mathrm{v}}(\mathrm{v},\ \mathrm{u}) = \gamma$ *iff* $\mathit{match}_{\tau}(\tau,\ \tau') = \gamma_{\tau}$ *with* $\vdash_{\gamma}\ \gamma\ :\ \gamma_{\tau}$.

Proof By induction on the pattern type derivation. □

Essentially, if a pattern matches an assertion, then the pattern's type matches the assertion's type, yielding related substitutions, and vice versa.

Lemma 16 (Dispatch). *If*

- $\mathrm{fid} = \langle \overrightarrow{\mathrm{fn}} \rangle$
- $\Gamma \vdash_{\mathrm{D}} \ \mathrm{D} \ : \ \mathrm{D_T}$
- $\Gamma, \overrightarrow{\mathrm{fn} : \mathtt{FacetName}, \mathit{bindings}_{\mathrm{D}}}(\mathrm{D}) \vdash_{\mathrm{Pr}} \ \mathrm{Pr} \ : \ \mathrm{T}$
- $\vdash_{\mathrm{Evt}} \ \mathrm{Evt} \ : \ \mathrm{Evt}_{\tau}$
- $\vdash_{\pi} \ \pi \ : \ \pi_{\tau}$
- $\vdash_{\pi} \ \pi' \ : \ \pi'_{\tau}$

then

$$\mathit{dispatch1}(\mathrm{fid},\ \mathrm{D},\ \mathrm{Pr},\ \mathrm{Evt},\ \pi,\ \pi',\ \sigma) = \overrightarrow{\mathrm{PS}}$$

iff

$$\mathit{dispatch1}_{\mathrm{T}}(\mathrm{fid},\ \mathrm{D_T},\ \mathrm{T},\ \mathrm{Evt}_{\tau},\ \pi_{\tau},\ \pi'_{\tau}) = \overrightarrow{\mathrm{PS_T}}$$

with $\overrightarrow{\Gamma, \overrightarrow{\mathrm{fn} : \mathtt{FacetName}} \vdash_{\mathrm{PS}} \ \mathrm{PS} \ : \ \mathrm{PS_T}}$

Proof By induction on the type derivations. The dispatch lemma (Lemma 16) lifts to events in general, so the term and type applications of $\mathit{match}_{\mathrm{D}}$ yield related substitutions. When applied to related event-handler bodies, the related substitutions yield related scripts. □

In other words, dispatching related events yields related scripts to execute.

Lemma 17 (Partial Evaluation). *If* $\Gamma \ \vdash_{\mathrm{Pr}} \ \mathrm{Pr} \ : \ \mathrm{T}$ *and* $\mathit{safe}(\mathrm{Pr})$ *then* $\mathit{p\text{-}e}(\mathrm{Pr},\ \sigma,\ \mathrm{fid}) = \overrightarrow{\mathrm{I}}, \sigma'$ *iff* $\mathit{p\text{-}e}_{\mathrm{T}}(\mathrm{T},\ \sigma_{\tau}, \mathrm{fid}) = \overrightarrow{\mathrm{I_T}}, \sigma'_{\tau}$ *with* $\overrightarrow{\Gamma \ \vdash_{\mathrm{I}} \ \mathrm{I} \ : \ \mathrm{I_T}}$

Proof By induction on the type derivation. □

That is, partially evaluating a pending script and its type yields related instructions.

Lemma 18 (Preservation). *If* $\vdash_{\mathrm{M}} \ \mathrm{M} \ : \ \mathrm{M_T}$ *and* $\mathit{safe}(\mathrm{M})$ *then* $\mathrm{M} \longrightarrow \mathrm{M}' = \langle \mathrm{FT};\ \overrightarrow{\mathrm{I}};\ \overrightarrow{\mathrm{PS}};\ \pi;\ \sigma \rangle$ *iff* $\mathrm{M_T} \longrightarrow \langle \mathrm{FT_T};\ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_{\tau};\ \sigma_{\tau} \rangle$ *with* $\vdash_{\mathrm{M}} \ \mathrm{M}' \ : \langle \mathrm{FT_T};\ \overrightarrow{\mathrm{I_T}};\ \overrightarrow{\mathrm{PS_T}};\ \pi_{\tau};\ \mathit{type\text{-}store}(\sigma) \rangle$

Proof Following a standard approach (Wright & Felleisen, 1994), we can show that at most one of the machine transition rules can apply. Through the application of related lemmas, such as Lemmas 16 and 17, the updated parts of the machine state remain related via typing. □

In other words, if a term machine takes a transition, then its type can take a transition to a related type machine, up to the type of the store. Since type machines do not manipulate or depend on the store at all, the store from the destination term-level machine state is translated to a new store type for the type-level.

Lemma 19 (Progress)**.** *If* $\vdash_{\mathrm{M}}\ \mathrm{M} : \mathrm{M_T}$ *either* $\mathtt{inert}(\mathrm{M})$, $\mathtt{inert}(\mathrm{M_T})$ *or there exists* $\mathrm{M}', \mathrm{M}'_{\mathrm{T}}$ *such that* $\mathrm{M} \xrightarrow{l_\bullet} \mathrm{M}'$ *and* $\mathrm{M_T} \xrightarrow{l_{\tau\bullet}} \mathrm{M}'_{\mathrm{T}}$.

Proof By inspection of the machine state. □

Lemma 20 (Machine Assertions)**.** *If* $\vdash_{\mathrm{M}}\ \mathrm{M} : \mathrm{M_T}$ *then*

$$\vdash_\pi\ \textit{assertions-of}_{\mathrm{M}}(\mathrm{M}) : \textit{assertions-of}_{\mathrm{M_T}}(\mathrm{M_T})$$

Proof By induction on the type derivation and via similar properties for the *assertions-of* family of functions. □

This establishes that related term and type machines make the same assertions.